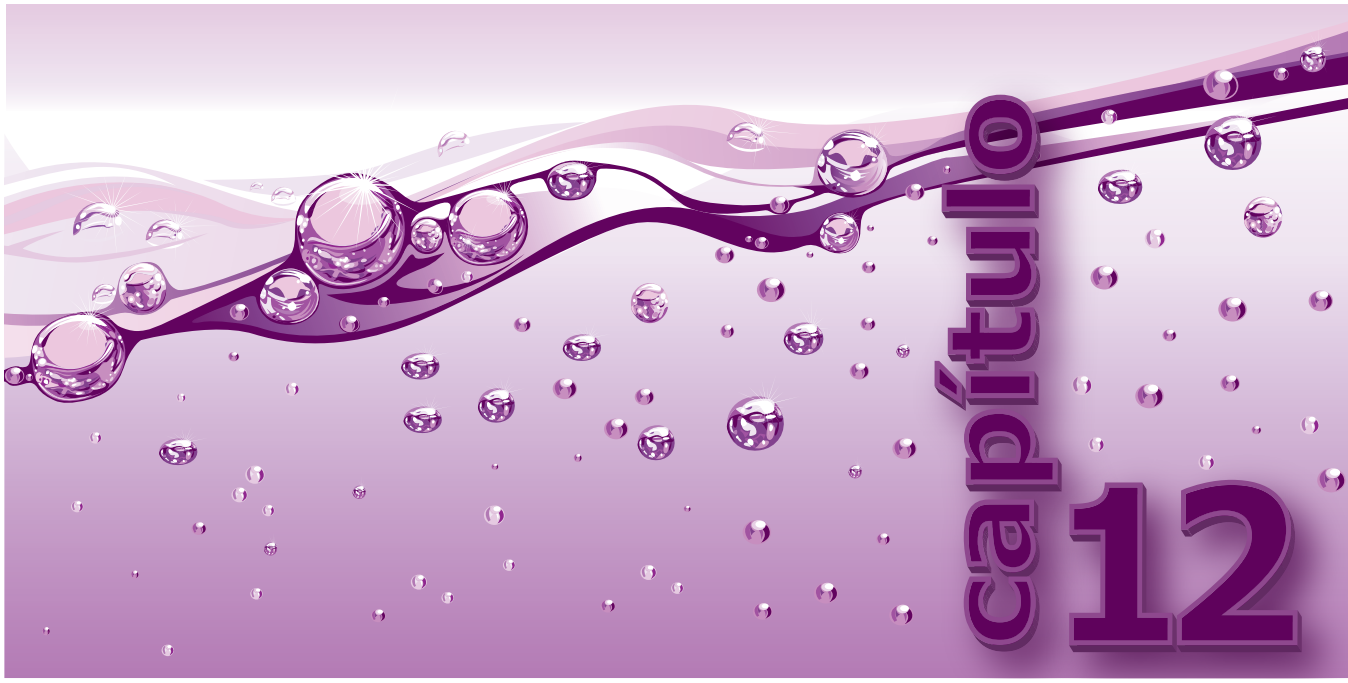

Ordenação em Memória Secundária e Bulkloading

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos no contexto de ordenação externa:
  - ❒ Série
  - ❒ Intercalação binária
  - ❒ Descarga de buffer
  - ❒ Recarga de buffer
  - ❒ Inserção massiva
  - ❒ Bulkloading
  - ❒ Buffers de entrada e de saída
  - ❒ Passagem de intercalação
  - ❒ Intercalação multíplice
- Explicar por que a maioria dos algoritmos usados em memória interna não são adequados para ordenação em memória externa
- Avaliar o desempenho de um algoritmo de ordenação externa
- Explicar por que nem sempre é possível efetuar intercalação numa única passagem quando se usa intercalação multíplice
- Determinar o tamanho do maior arquivo que pode ser ordenado usando intercalação multíplice
- Explicar o papel desempenhado por um heap em intercalação multíplice
- Expressar as vantagens que intercalação multíplice apresenta com respeito a intercalação binária
- Implementar os algoritmos de intercalação binária e de intercalação multíplice
- Descrever o processo de criação de árvore B+ usando inserção massiva
- Identificar o formato de uma árvore B+ criada por meio de inserção massiva
- Justificar o uso de bulkloading em detrimento do método ordinário de criação de árvores B+

objetivos

No Capítulo 11, todos os algoritmos examinados requerem que a tabela a ser ordenada esteja inteiramente em memória principal. Existem, entretanto, aplicações nas quais as tabelas são grandes demais e não podem ser contidas em memória interna. Este capítulo discute algoritmos de **ordenação externa** elaborados para lidar com volumes de dados muito grandes que devem residir em memória secundária relativamente lenta (usualmente, um HD). Este capítulo lida com situações nas quais deseja-se ordenar um arquivo sem ter que mantê-lo integralmente em memória principal.

Quando dados estão armazenados em memória secundária, algoritmos que são eficientes em memória principal podem não ser eficientes se o custo temporal do algoritmo é expresso como o número de operações de entrada e saída (v. **Capítulo 6**). O que norteia ordenação externa é minimização de acesso à memória secundária, visto que ler um bloco em disco leva cerca de um milhão de vezes mais tempo do que acessar um item em RAM (v. **Capítulo 1**). Portanto algoritmos de ordenação em memória externa são elaborados para minimizar o número de operações de entrada e saída.

A maioria dos algoritmos de ordenação externa é baseada em MergeSort. Eles tipicamente dividem um arquivo de dados grande em vários arquivos ordenados menores denominados *séries*. Essas séries são produzidas transferindo-se repetidamente uma porção do arquivo de dados para a memória principal, ordenando-a com um algoritmo de ordenação para memória principal (p. ex., QuickSort) e escrevendo os dados ordenados num arquivo em memória secundária. Depois de as séries ordenadas serem geradas, um algoritmo de intercalação é usado para combinar arquivos ordenados menores em arquivos ordenados maiores. A abordagem mais simples (v. **Seção 12.2**) consiste em intercalar arquivos ordenados dois a dois consecutivamente até obter apenas um grande arquivo ordenado. Entretanto a melhor abordagem é usar um algoritmo de intercalação multíplice (v. **Seção 12.4**) que pode intercalar várias séries mais curtas ao mesmo tempo.

Além de ordenação externa, este capítulo discute na **Seção 12.6** um processo de construção de árvores B+ a partir de um grande conjunto de registros armazenados em arquivo. Esse processo, denominado **inserção massiva** (*bulkloading*, em inglês), consiste na inserção de uma grande quantidade de registros ordenados numa árvore B+ por meio de uma única operação.

12.1 Conceitos Básicos

Ordenação externa tipicamente usa uma abordagem híbrida de ordenação por intercalação (v. **Seção 11.3.2**). Esse tipo de ordenação externa consiste em duas fases: (1) ordenação e (2) intercalação. Na **fase de ordenação**, parte dos registros são lidos (i.e., armazenados em memória principal), ordenados usando algum método de ordenação visto no **Capítulo 11** e escritos num arquivo temporário. Esse processo é repetido até que todos os registros do arquivo a ser ordenado sejam processados. Na **fase de intercalação**, os arquivos ordenados são combinados num único arquivo maior. Nessa fase de intercalação, cada par de arquivos resultante da fase de ordenação é intercalado de modo semelhante àquele mostrado na **Seção 11.3.2**, resultando num novo arquivo ordenado. Essa etapa é repetida até restar apenas um arquivo, que representa o arquivo original ordenado.

No contexto de ordenação externa, um conjunto de registros ordenados é denominado **série**. Num algoritmo de **intercalação binária** (v. **Seção 12.2**), as séries são intercaladas duas a duas. Um algoritmo de **intercalação multíplice** (v. **Seção 12.4**), por outro lado, intercala mais de duas séries de cada vez. De fato, o modo como a intercalação é efetuada é a principal diferença entre os métodos de ordenação externa estudados neste capítulo. Uma **passagem de intercalação** é uma operação na qual duas ou mais séries são completamente intercaladas e dão origem a uma nova série.

Na discussão sobre ordenação em memória secundária a ser apresentada neste capítulo, serão utilizados os parâmetros e respectivas interpretações apresentados na **Tabela 6–3** vista no **Capítulo 6**.

12.2 Intercalação Binária

12.2.1 Descrição

Suponha que E representa o arquivo que armazena os registros que precisam ser ordenados e S representa um arquivo vazio. O algoritmo de intercalação binária em memória secundária é dividido em duas fases principais. A **Fase 1**, mostrada na **Figura 12–1**, é responsável pela produção de séries iniciais. Ao final da **Fase 1**, tem-se, no arquivo S, $\lceil N/M \rceil$ séries de M registros, sendo N o número total de registros e M o número máximo de registros que cabem em memória principal. Contudo a última série pode ser mais curta. Por exemplo, se $N = 50000$ e $M = 1500$, o número de séries será $\lceil 50000/1500 \rceil = 34$, sendo que 33 séries terão 1500 registros e a última delas terá 500 registros.

ALGORITMO CRIASÉRIES

ENTRADA: Arquivo E contendo os registros que serão ordenados

ENTRADA: Arquivo S contendo as séries criadas

1. Repita os seguintes passos até que o final do arquivo E seja atingido:

- **1.1** Leia os próximos M registros do arquivo E em memória principal
- **1.2** Ordene-os em memória principal usando um algoritmo de ordenação usado em memória principal (v. **Capítulo 11**)
- **1.3** Escreva os registros ordenados ao final do arquivo S

FIGURA 12–1: ALGORITMO DE CRIAÇÃO DE SÉRIES

O algoritmo mostrado na **Figura 12–2** representa a **Fase 2** encarregada da intercalação dessas séries. A cada passo desse algoritmo, o número de séries é dividido por dois e o algoritmo para quando resta apenas uma série.

ALGORITMO INTERCALAÇÃOBINÁRIA

ENTRADA: Arquivo E contendo as séries que serão intercaladas

SAÍDA: Arquivo S contendo os registros ordenados

1. Enquanto houver mais de uma série no arquivo E, faça:

- **1.1** Esvazie o arquivo S
- **1.2** Se restar apenas uma série no arquivo E, copie-a para o final do arquivo S
- **1.3** Caso contrário:
  - **1.3.1** Intercale as próximas duas séries do arquivo E numa série
  - **1.3.2** Escreva a série resultante da intercalação ao final do arquivo S
- **1.4** Faça com que o arquivo S passe a ser o arquivo E
- **1.5** Faça com que o arquivo E passe a ser o arquivo S

2. Garanta que arquivo S seja o arquivo que contém o resultado

FIGURA 12–2: ALGORITMO DE INTERCALAÇÃO BINÁRIA

O último passo do algoritmo **INTERCALAÇÃOBINÁRIA** é necessário devido à frequente troca de papéis dos arquivos envolvidos no processo. Quer dizer, ora o arquivo E serve como entrada ora ele serve como saída e o mesmo ocorre com o arquivo S, como ilustra a **Figura 12–3**.

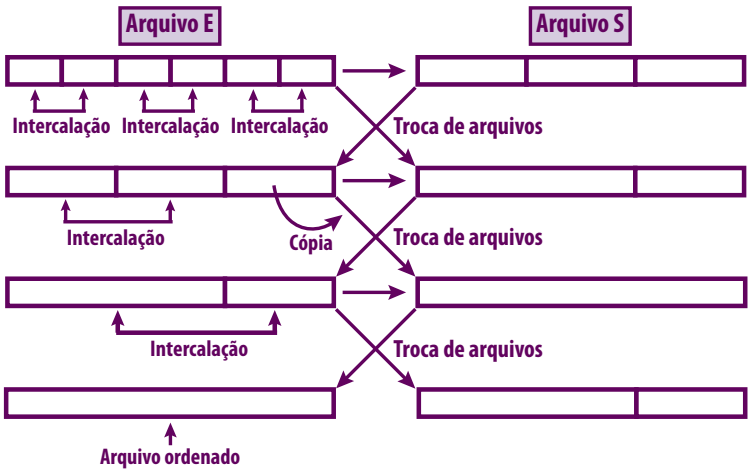

Figura 12–3: Troca de Papéis de Arquivos em Intercalação Binária

Qualquer implementação per si do algoritmo de intercalação binária apresentado na **Figura 12–2** está fadada à ineficiência devido ao fato de ele requerer acesso direto aos registros que constituem as séries que estão sendo intercaladas. Assim uma abordagem bem melhor para esse algoritmo mantém três arrays de tamanho B em memória principal (i.e., armazenando um bloco de disco cada). Os primeiros dois arrays, B_1 e B_2, são **buffers** (**de entrada**) para blocos de disco lidos da primeira e da segunda séries. O terceiro array é um **buffer de saída**. Esse algoritmo funciona como **MergeSort** em memória principal, intercalando os arrays B_1 e B_2 e resultando no array B_0. Se o final de B_1 ou B_2 for atingido, o próximo bloco da série correspondente é lido e armazenado em memória principal. Também, tão logo o array B_0 se torne repleto, ele é escrito no arquivo de saída. A **Figura 12–4** ilustra essa intercalação de blocos em memória principal.

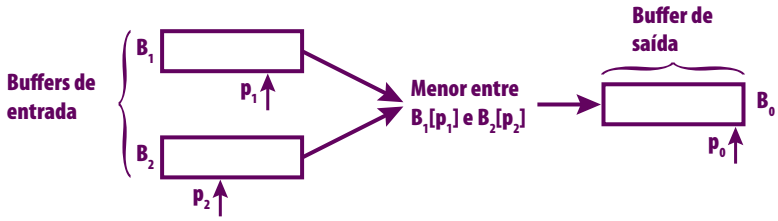

Figura 12–4: Intercalação Binária em Memória Principal

12.2.2 Implementação

A função `CriaSeriesMS()` lê os registros de um arquivo a ser ordenado e cria séries usando a função **qsort()** da biblioteca padrão de C (v. **Capítulo 11** do **Volume 1**). A função `CriaSeriesMS()` retorna o número de séries criadas e seus parâmetros são:

- `nomeE` (entrada) — nome do arquivo que origina as séries
- `nomeS` (entrada) — nome do arquivo que armazenará as séries

```
int CriaSeriesMS(const char *nomeE, const char *nomeS)
{
   tRegistroMEC *registros; /* Ponteiro para um array que armazena */
                            /* uma série em memória principal      */
   int           nRegistros,  /* Número de registros numa série */
                 nSeries = 0; /* Número de séries */
   FILE         *streamS, /* Stream associado ao arquivo que armazenará as séries */
                *streamE; /* Stream associado ao arquivo original */
```

```
      /* Aloca espaço para o array. Um array deste */
      /* tamanho não pode ser alocado na pilha     */
   registros = calloc(MAX_REGISTROS, TAM_REG);
   ASSEGURA( registros, "Buffer nao foi alocado" );

      /* Tenta criar o arquivo que conterá as séries */
   streamS = AbreArquivo(nomeS, "wb");

      /* Tenta abrir o arquivo original */
   streamE = AbreArquivo(NOME_ARQUIVO_BIN, "rb");

      /* Lê o arquivo a ser ordenado e cria as séries */
   while (1) {
         /* Lê os registros da série corrente */
      nRegistros = fread( registros, TAM_REG, MAX_REGISTROS, streamE );

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

         /* Encerra o laço se nenhum registro foi lido */
      if (!nRegistros)
         break;

         /* Ordena os registros lidos */
      qsort(registros, nRegistros, TAM_REG, ComparaInts2);

         /* Escreve a série corrente no arquivo */
      fwrite(registros, TAM_REG, nRegistros, streamS);

         /* Verifica se ocorreu erro de escrita */
      ASSEGURA( !ferror(streamS), "Erro de escrita em arquivo de series" );

      ++nSeries; /* Atualiza o número de séries */
   }

   FechaArquivo(streamS, nomeS); /* Fecha os arquivos */
   FechaArquivo(streamE, nomeE);

   free(registros); /* Libera o espaço ocupado pelo array */

   return nSeries;
}
```

A função `CriaSeriesMS()` usa as macros `MAX_REGISTROS` e `TAM_REG` definidas como:

```
   /* Número máximo de registros que (hipoteticamente) cabe em memória principal */
#define MAX_REGISTROS 80000

#define TAM_REG       sizeof(tRegistroMEC)  /* Tamanho de um registro  */
```

A constante `MAX_REGISTROS` representa, supostamente, em número de registros, a capacidade da memória principal disponível para execução do programa de ordenação. O tipo `tRegistroMEC` é definido no **Apêndice A**.

A função `IntercalaBinMS()` efetua intercalação binária de séries armazenadas num arquivo e seus parâmetros são:

- `nomeE` (entrada) — nome do arquivo que inicialmente contém as séries que serão intercaladas
- `nomeS` (entrada) — nome do arquivo que inicialmente conterá o resultado de intercalação das séries do primeiro arquivo
- `nSeries` (entrada) — número inicial de séries
- `tamSerie` (entrada) — tamanho (número de registros) máximo de cada série inicial

A função `IntercalaBinMS()` retorna o nome do arquivo que contém o resultado da intercalação (v. adiante).

```
const char *IntercalaBinMS( const char *nomeE, const char *nomeS,
                            int nSeries, int tamSerie )
{
   FILE       *streamS, /* Stream associado ao arquivo que contém */
                        /* o resultado de concatenação de séries  */
              *streamE; /* Stream associado ao arquivo que contém */
                        /* séries que serão concatenadas          */
   int         ns = nSeries, /* Número corrente de séries */
               ts = tamSerie; /* Tamanho máximo de cada série */
   int         i1, i2; /* Índices de duas séries que serão intercaladas */
   const char *aux;

   while (ns > 1) {
         /* Tenta abrir o arquivo contendo as séries que serão intercaladas */
      streamE = AbreArquivo(nomeE, "rb");

         /* Tenta criar o arquivo que conterá as séries resultantes */
      streamS = AbreArquivo(nomeS, "wb");

      i1 = 0; /* Índice inicial da primeira série */
      i2 = 1; /* Índice inicial da segunda série  */

         /* As séries são indexadas de 0 a ns - 1. */
         /* Portanto o maior valor de i2 é ns - 1. */
      while (i2 < ns) {
            /* Intercala as duas séries correntes */
         IntercalaDuasSeriesMS(streamE, streamS, i1, i2, ts);

         i1 += 2; /* Passa para as próximas séries */
         i2 += 2;
      }

         /* Se o número corrente de séries for ímpar, resta */
         /* copiar a última série para o arquivo resultante */
      if (i1 < ns) {
            /* Verificação de consistência */
         ASSEGURA(ns%2, "O numero de series deveria ser impar");
         ASSEGURA(i1 == ns - 1, "Deveria ser a ultima serie");

            /* Move apontador para o primeiro byte a ser copiado */
            /* e copia a última série para o arquivo resultante  */
         MoveApontador( streamE, (long)i1*ts*TAM_REG, SEEK_SET );
         CopiaRestoArquivo(streamE, streamS, 0);
      }

      FechaArquivo(streamE, nomeE); /* Fecha os arquivos */
      FechaArquivo(streamS, nomeS);

      ns = ns/2 + ns%2; /* O número de séries foi reduzido à metade */

         /* O número máximo de registros de cada série dobra de tamanho,   */
         /* sendo que o número de registros da última série pode ser menor */
      ts *= 2;

      aux = nomeE; /* Troca os papéis dos arquivos */
      nomeE = nomeS;
      nomeS = aux;
   }

   return nomeE; /* Retorna o nome do arquivo que contém o resultado */
}
```

Na função `IntercalaBinMS()`, os nomes associados aos arquivos de entrada e saída mudam durante sua execução (i.e., eles trocam de papéis, como mostra a **Figura 12–3**). Assim, ao final da operação, o resultado pode estar armazenado no arquivo representado pelo primeiro ou pelo segundo parâmetro dessa função. Essa é a razão pela qual ela retorna o nome do arquivo que contém o resultado, mas é fácil alterá-la de modo que o resultado seja sempre armazenado no segundo parâmetro.

A função `IntercalaDuasSeriesMS()`, chamada por `IntercalaBinMS()`, intercala duas séries armazenadas num arquivo e armazena o resultado noutro arquivo, de acordo com o que foi ilustrado na **Figura 12–4**. Os parâmetros de `IntercalaDuasSeriesMS()` são:

- `streamE` (entrada) — stream associado ao arquivo que contém as séries
- `streamS` (entrada) — stream associado ao arquivo que conterá o resultado da intercalação
- `is1` (entrada) — índice da primeira série
- `is2` (entrada) — índice da segunda série
- `tamSerie` (entrada) — tamanho máximo de uma série

```
static void IntercalaDuasSeriesMS( FILE *streamE, FILE *streamS,
                                   long is1, long is2, int tamSerie )
{
   tRegistroMEC *buf1, *buf2, /* Buffers de entrada */
                *bufS; /* Buffer de saída */
   int           ir1 = 0, /* Índice do próximo registro a ser lido no buffer 1 */
                 ir2 = 0, /* Índice do próximo registro a ser lido no buffer 2 */
                 irs = 0, /* Índice do próximo registro a   */
                          /* ser escrito no buffer de saída */
                 tamBuffer, /* Tamanho de um buffer */
                 n1, n2, /* Números de registros nos buffers associados */
                         /* respectivamente às séries 1 e 2           */
                 resto1 = tamSerie,/* Registros restantes na série 1 */
                 resto2 = tamSerie;/* Registros restantes na série 2 */
   long          pos1, /* Posição de leitura de um byte da série 1 */
                 pos2; /* Posição de leitura de um byte da série 2 */

      /* A primeira série deve preceder a segunda série */
   ASSEGURA(is1 < is2, "Indice da 1a serie e' maior do que indice da 2a");

      /* Divide a memória supostamente disponível pelos três buffers */
   tamBuffer = MAX_REGISTROS/3;

      /* Aloca espaço para cada buffer */
   buf1 = calloc(tamBuffer, TAM_REG);
   buf2 = calloc(tamBuffer, TAM_REG);
   bufS = calloc(tamBuffer, TAM_REG);
   ASSEGURA( buf1 && buf2 && bufS, "Pelo menos um buffer nao foi alocado" );

      /* Calcula a posição do primeiro byte de cada série */
   pos1 = is1*tamSerie*TAM_REG;
   pos2 = is2*tamSerie*TAM_REG;

      /* Efetua os primeiros carregamentos de buffers */
   n1 = CarregaBufferMS(streamE, buf1, MIN(tamBuffer, resto1), &pos1);
   n2 = CarregaBufferMS(streamE, buf2, MIN(tamBuffer, resto2), &pos2);

      /* Calcula o número de registros restantes em cada série */
   resto1 -= n1;
   resto2 -= n2;

      /* Intercala as duas séries */
```

```
  while (1) {
        /* Se o buffer de saída estiver cheio descarrega-o para o arquivo de saída */
     if (irs >= tamBuffer) {
        fwrite(bufS, TAM_REG, tamBuffer, streamS);
        ASSEGURA(!ferror(streamS), "Erro de escrita");

        irs = 0; /* Buffer ficou novinho em folha */
     }

        /* Verifica se o buffer 1 está esgotado */
     if (ir1 >= n1) {
           /* Se ainda houver registros na série 1, recarrega esse buffer. */
           /* Caso contrário, o processamento dessa série está concluído   */
        if (resto1 > 0) {
              /* Ainda há registros da série 1 em arquivo */
           n1 = CarregaBufferMS( streamE, buf1, MIN(tamBuffer, resto1), &pos1 );
           ir1 = 0;
           resto1 -= n1;
        } else {
           /*                                                              */
           /* Não há mais registros na série 1. Copia o restante da        */
           /* segunda série para o arquivo de saída e encerra o laço.      */
           /*                                                              */

              /* Primeiro, copia os registros que se encontram no buffer de */
              /* saída e depois copia aqueles que se encontram no buffer 2  */
           fwrite(bufS, TAM_REG, irs, streamS);
           fwrite(buf2 + ir2, TAM_REG, n2 - ir2, streamS);
           ASSEGURA(!ferror(streamS), "Erro de escrita");

              /* Se houver registros da série 2 restantes no */
              /* arquivo, copia-os para o arquivo de saída   */
           if (resto2 > 0) {
              MoveApontador(streamE, pos2, SEEK_SET);
              CopiaRestoArquivo(streamE, streamS, (long) resto2*TAM_REG);
           }
           break;
        }
     }

        /* Verifica se o buffer 2 está esgotado */
     if (ir2 >= n2) {
           /* Se ainda houver registros na série 2, recarrega esse buffer. */
           /* Caso contrário, o processamento da série 2 está concluído.   */
        if (resto2 > 0) {
              /* Talvez, ainda haja registros da série 2 em arquivo, mas o   */
              /* número inicial de registros dessa série pode ter sido menor */
           n2 = CarregaBufferMS(streamE, buf2, MIN(tamBuffer, resto2), &pos2);
           ir2 = 0;
           resto2 = n2 ? resto2 - n2 : 0;
        } else {
           /*                                                          */
           /* Não há mais registros na série 2. Copia o restante da */
           /* série 1 para o arquivo de saída e encerra o laço.     */
           /*                                                          */

              /* Primeiro, copia os registros que se encontram no buffer de */
              /* saída e depois copia aqueles que se encontram no buffer 1  */
           fwrite(bufS, TAM_REG, irs, streamS);
```

```
            fwrite(buf1 + ir1, TAM_REG, n1 - ir1, streamS);
            ASSEGURA(!ferror(streamS), "Erro de escrita");

               /* Se houver registros da série 1 restantes no */
               /* arquivo, copia-os para o arquivo de saída   */
            if (resto1 > 0) {
               MoveApontador(streamE, pos1, SEEK_SET);
               CopiaRestoArquivo( streamE, streamS, (long) resto1*TAM_REG );
            }
            break;
         }
      }

         /* O teste a seguir é necessário, pois a série 2 pode */
         /* ser a última do arquivo e seu tamanho inicial pode */
         /* ser menor do que o tamanho das demais séries       */
      if (n2 > 0)
            /* Copia o menor dos dois registros correntes nos  */
            /* buffers de entrada 1 e 2 para o buffer de saída */
         bufS[irs++] = ComparaInts2(buf1 + ir1, buf2 + ir2) < 0 ? buf1[ir1++]
                                                                : buf2[ir2++];
   }

   free(buf1); /* Libera os buffers */
   free(buf2);
   free(bufS);
}
```

A função `IntercalaDuasSeriesMS()` considera que as séries são indexadas de *0* a *ns – 1*, em que *ns* é o número de séries. Em cada série, os registros são indexados de *0* a *ts – 1*, sendo *ts* o número máximo de registros numa série, como ilustra a **Figura 12–5**. De fato, todas as séries, com a possível exceção da última série, têm exatamente *ts* registros. A função `ComparaInts2()` utilizada por `CriaSeriesMS()` e `IntercalaDuasSeriesMS()` compara duas chaves inteiras de registros do tipo `tRegistroMEC` e é relativamente fácil de implementar.

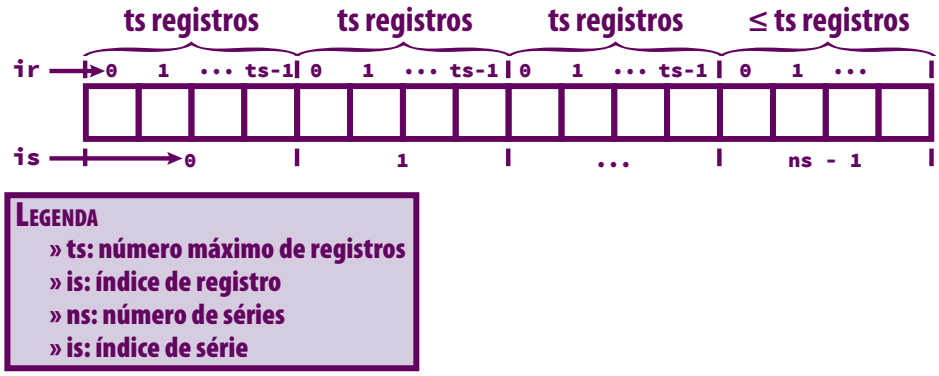

FIGURA 12–5: RELAÇÃO ENTRE ÍNDICE DE REGISTRO E ÍNDICE DE SÉRIE

A função `CarregaBufferMS()` preenche o conteúdo de um buffer com registros ordenados lidos do arquivo que contém as séries. Essa função retorna o número de registros lidos e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que contém os registros ordenados
- `buffer` (entrada) — o buffer que será preenchido
- `maxRegs` (entrada) — número máximo de registros que serão lidos
- `*pos` (entrada/saída) — posição da próxima leitura no arquivo

```
static int CarregaBufferMS(FILE *stream, tRegistroMEC *buffer, int maxRegs, long *pos)
{
   int nRegistros; /* Número de registros lidos */

      /* Move o apontador de posição para o local de leitura */
   MoveApontador(stream, *pos, SEEK_SET);

      /* Lê os registros no arquivo */
   nRegistros = fread(buffer, TAM_REG, maxRegs, stream);
   ASSEGURA(!ferror(stream), "Erro de leitura");

   *pos = ObtemApontador(stream); /* Obtém a posição da próxima leitura */

   return nRegistros;
}
```

A função `CopiaRestoArquivo()` chamada por `IntercalaDuasSeriesMS()` é semelhante à função `CopiaArquivo()` definida na **Seção 2.15.2**. Uma diferença entre elas é que a função utilizada aqui copia um arquivo para outro a partir da posição em que se encontram seus apontadores de posição. Além disso, função `CopiaRestoArquivo()` permite limitar o número de bytes copiados. As funções `AbreArquivo()` e `FechaArquivo()` chamadas pelas funções exibidas acima foram discutidas na **Seção 2.3**. As funções `MoveApontador()` e `ObtemApontador()` chamadas pelas funções `IntercalaBinMS()` e `IntercalaDuasSeriesMS()` foram definidas na **Seção 2.11.2**.

O leitor encontrará o código completo do programa de intercalação binária no site dedicado ao livro na internet. Além disso, nesse site, aparecem duas implementações alternativas de ordenação de arquivos por intercalação binária. A única diferença entre a primeira implementação alternativa e aquela apresentada aqui é que a função `IntercalaDuasSeriesMS()` da implementação alternativa não usa buffers (i.e., ela segue rigorosamente o algoritmo da **Figura 12–2**). O uso de buffers faz com que a implementação explorada acima seja bem mais eficiente do que essa implementação alternativa, como mostra a **Tabela 12–1**.

		Implementação Corrente		Implementação Alternativa 1		Implementação Alternativa 2	
# Registros	# Series	Fase 1	Fase 2	Fase 1	Fase 2	Fase 1	Fase 2
600.000	*8*	*3,00 s*	*2,00 s*	*3,00 s*	*33,0 s*	*3,00 s*	*26,0 s*
1.200.000	*15*	*8,00 s*	*6,00 s*	*8,00 s*	*1,48 m*	*9,00 s*	*1,13 m*
2.400.000	*30*	*14,00 s*	*1,12 m*	*14,00 s*	*3,58 m*	*20,00 s*	*3,02 m*
4.800.000	*60*	*28,00 s*	*4,08 m*	*28,00 s*	*8,90 m*	*37,00 s*	*6,97 m*
9.565.483	*120*	*58,00 s*	*8,75 m*	*58,00 s*	*20,28 m*	*1,23 m*	*16,17 m*

Tabela 12–1: Implementações Alternativas de Intercalação Binária de Arquivos

Na segunda implementação alternativa, cada série criada é armazenada num arquivo próprio. Essa implementação é relativamente mais fácil de entender do que aquela apresentada acima, mas ela não segue os algoritmos apresentados na **Figura 12–1** e na **Figura 12–2**. Examinando a **Tabela 12–1**, note que a implementação exibida acima apresenta uma ligeira vantagem com relação a essa segunda implementação alternativa, pois, além do uso de buffers, suas séries são armazenadas num mesmo arquivo. Por outro lado, essa segunda implementação alternativa leva uma sensível vantagem com relação à primeira implementação alternativa. Isso ocorre porque, apesar do ônus envolvido na criação e remoção de vários arquivos, durante a fase de intercalação, a segunda implementação alternativa lê os registros usando um padrão de acesso sequencial. A primeira implementação

alternativa, por sua vez, efetua esse processamento usando um padrão de acesso direto, que, como foi visto no **Capítulo 1** (v. **Seção 1.1.3**), é bem mais lento do que acesso sequencial.

12.2.3 Análise

Na análise de um algoritmo baseado em entrada e saída, leva-se em conta a quantidade de operações de acesso à memória secundária que são efetuadas, sendo que a unidade de acesso é o bloco. Assim o custo temporal de um algoritmo de ordenação externa é estimado pelo número de operações de entrada e saída efetuadas durante o processo de ordenação. Isso significa que eliminar uma única dessas operações pode melhorar significativamente o desempenho de tal algoritmo. Como se supõe que o custo temporal de um método de ordenação externa é dominado por entrada e saída, pode-se estimar o custo temporal de um método de ordenação externa multiplicando-se o número de acessos à memória secundária pelo tempo requerido para ler e escrever um bloco. De modo semelhante ao que ocorre com busca em memória secundária (v. **Capítulo 6**), supõe-se que ordenar dados em memória principal é desprezível em face a esse número de acessos.

Teorema 12.1: O custo de transferência do algoritmo de intercalação binária é $\theta(n \cdot log_2(n/m))$.

Prova: A **Fase 1** do algoritmo de intercalação binária (v. **Figura 12–1**) lê todos os blocos de um arquivo E e escreve a mesma quantidade de blocos num arquivo S. Assim $2n$ operações de entrada e saída são efetuadas, em que n é o número de blocos de disco no arquivo inicial. Numa iteração do laço principal da **Fase 2** (v. **Figura 12–2**), blocos de todas as séries no arquivo E são lidos e essa mesma quantidade de blocos é escrita no arquivo S. Novamente, $2n$ [i.e., $\theta(n)$] operações de entrada ou saída são efetuadas. Nessa fase, cada iteração reduz o número de séries pela metade. Como começa-se com $\lceil N/M \rceil = \lceil n/m \rceil$ séries e termina-se com uma única série, há $log_2(n/m)$ iterações, cada uma das quais executando $\theta(n)$ operações de entrada e saída. Somando os custos da **Fase 1** e da **Fase 2**, obtém-se o custo temporal total $\theta(n \cdot log_2(n/m))$. ■

12.3 Intercalação Multíplice de Arrays

12.3.1 Descrição do Problema

A compreensão da próxima técnica de ordenação externa a ser discutida e de sua análise serão bastante facilitadas se o leitor estudar previamente o problema a ser descrito na presente seção. Esse problema é relativamente trivial e consiste em ordenar k arrays, sendo que cada um dos quais contém n elementos já ordenados. O resultado da ordenação deve ser armazenado em outro array.

Uma solução óbvia para esse problema seria armazenar os elementos dos arrays no array que conterá os itens ordenados e, então, ordená-los usando um método de ordenação com o melhor custo temporal possível, que, em geral, é $\theta(n \cdot k \cdot log\ n \cdot k)$, como foi visto no **Capítulo 11**. Mas essa é uma abordagem ingênua que não leva em consideração o fato de os k arrays já estarem ordenados.

Uma abordagem melhor para o problema em questão consiste em intercalar os k arrays contendo n elementos ordenados usando um heap de mínimo (v. **Capítulo 10**). Usando essa abordagem, pode-se mostrar que o custo da ordenação cai para $\theta(n \cdot k \cdot log\ k)$, que, à primeira vista, parece ser um ganho irrelevante com relação à abordagem ingênua, mas que, de fato, não o é, quando essa mesma abordagem é usada em ordenação externa, como será visto na próxima seção.

12.3.2 Algoritmo

Precisamente, a técnica de **intercalação multíplice de arrays** por intermédio de heap consiste dos passos apresentados na **Figura 12–6**.

Algoritmo IntercalaçãoMultíplicеDeArrays

Entrada: k arrays ordenados, cada um dos quais com tamanho n

Saída: Um array de tamanho $n \cdot k$ contendo os elementos dos k arrays de entrada

1. Aloque um array com capacidade suficiente para conter o resultado (i.e., com tamanho igual a $n \cdot k$)
2. Crie um heap de mínimo de tamanho igual a k
3. Insira o primeiro elemento de cada array no heap
4. Enquanto os k arrays não estiverem exauridos, faça o seguinte:
   - **4.1** Obtenha o menor elemento do heap (que se encontra em sua raiz)
   - **4.2** Armazene esse elemento no array que conterá o resultado
   - **4.3** Se ainda houver algum elemento a ser considerado no array ao qual a raiz do heap pertencia, substitua-a pelo próximo elemento desse array
   - **4.4** Caso contrário, substitua a raiz do heap por seu último elemento
   - **4.5** Reorganize o heap de modo que ele continue satisfazendo sua propriedade de ordenação

Figura 12–6: Algoritmo de Intercalação Múltiplice de Arrays

12.3.3 Implementação

A implementação do algoritmo descrito na **Figura 12–6** será apresentada adiante. As seguintes definições de tipos serão utilizadas:

```
typedef int tElemArray; /* Tipo de cada elemento dos arrays */

   /* Tipo de nó de um heap */
typedef struct {
           tElemArray elemento; /* Elemento do array */
           int        i;        /* Índice do elemento em seu array */
           int        iArray;   /* Índice do array ao qual o elemento pertence */
        } tNoHeapIM_Arr;

   /* Tipo que representa um heap */
typedef struct {
           tNoHeapIM_Arr *itens;  /* Array de elementos */
           int            nItens; /* Número de elementos do heap */
        } tHeapIM_Arr;

   /* Tipo de ponteiro para função de comparação que compara elementos do array */
typedef int (*tFCompara) (const void *, const void *);
```

A função `IniciaHeapIM_Arr()` inicia um heap e usa como parâmetros:

- `*heap` (saída) — heap que será iniciado
- `nElementos` (entrada) — número de elementos do heap

Essa função retorna seu primeiro parâmetro.

```
tHeapIM_Arr *IniciaHeapIM_Arr(tHeapIM_Arr *heap, int nElementos)
{
   heap->nItens = nElementos;
   heap->itens = calloc(nElementos, sizeof(tNoHeapIM_Arr));

   ASSEGURA(heap->itens, "Impossivel alocar heap");

   return heap;
}
```

A função `ObtemMinimoIM_Arr()` retorna o menor elemento de um heap de mínimo e seu único parâmetro é um ponteiro para o heap.

```
tNoHeapIM_Arr ObtemMinimoIM_Arr(const tHeapIM_Arr *heap)
{
   return heap->itens[0]; /* O menor elemento de um heap de mínimo se acha na raiz */
}
```

A função `SubstituiMinimoIM_Arr()` substitui o menor elemento de um heap de mínimo e tem como parâmetros:

- `heap` (entrada e saída) — ponteiro para o heap que terá seu menor elemento substituído
- `item` (entrada) — ponteiro para o conteúdo do elemento que substituirá o menor elemento do heap
- `Compara` (entrada) — ponteiro para uma função que compara dois elementos de array armazenados no heap

```
void SubstituiMinimoIM_Arr(tHeapIM_Arr *heap, const tNoHeapIM_Arr *item, tFCompara Compara)
{
   heap->itens[0] = *item; /* Efetua a substituição */

      /* Talvez a substituição da raiz faça com que o heap */
      /* deixe de satisfazer sua propriedade de ordenação  */
   OrdenaHeapIM_Arr(heap, 0, Compara);
}
```

A função `OrdenaHeapIM_Arr()`, chamada por `SubstituiMinimo()`, restaura a propriedade de ordenação de um heap de mínimo após ele ter sido alterado. Essa função é semelhante à função `OrdenaHeap()` definida na **Seção 10.2.4**.

A função `ReduzHeapIM_Arr()` reduz o tamanho de um heap e retorna seu último elemento. O único parâmetro (de entrada e saída) dessa função é um ponteiro para o heap.

```
tNoHeapIM_Arr ReduzHeapIM_Arr(tHeapIM_Arr *heap)
{
   return heap->itens[--heap->nItens];
}
```

A função `IntercalaNArrays()` intercala *k* arrays ordenados de *n* elementos cada usando um heap de mínimo e tem como parâmetros:

- `ar[][]` (entrada) — os arrays ordenados
- `k` (entrada) — o número de arrays ordenados

Essa função retorna o endereço do array que contém o resultado da intercalação

```
tElemArray *IntercalaNArrays(tElemArray ar[][N_ELEMENTOS], int k)
{
   tElemArray   *resultado;
   tHeapIM_Arr   heap;
   int           i;
   tNoHeapIM_Arr raiz;

      /* Aloca espaço para o array que conterá o resultado */
   resultado = calloc(N_ELEMENTOS*k, sizeof(tElemArray));
   ASSEGURA(resultado, "Impossivel alocar array");

   IniciaHeapIM_Arr(&heap, k); /* Inicia o heap */

      /* Associa o primeiro elemento de cada array a um nó do heap */
```

```
   for (i = 0; i < k; i++) {
      heap.itens[i].elemento = ar[i][0];
      heap.itens[i].i = 0;
      heap.itens[i].iArray = i;
   }

      /* Ordena o heap */
   for (i = (k - 1)/2; i >= 0; --i)
      OrdenaHeapIM_Arr(&heap, i, ComparaInts);

      /* Obtém o menor elemento do heap e o substitui pelo próximo elemento */
      /* do array que o contém até que todos os arrays estejam esgotados    */
   for (i = 0; i < N_ELEMENTOS*k; ++i) {
      raiz = ObtemMinimoIM_Arr(&heap); /* Obtém o menor valor armazenado no heap */

         /* Armazena o menor valor no array que */
         /* conterá o resultado da intercalação */
      resultado[i] = raiz.elemento;

         /* O elemento que substituirá a raiz pertence ao mesmo array ao */
         /* qual pertence raiz, a não ser que esse array esteja esgotado */
      if (++raiz.i < N_ELEMENTOS)
            /* Substitui a raiz por um novo elemento */
            /* do array ao qual ela pertencia        */
         raiz.elemento = ar[raiz.iArray][raiz.i];
      else
            /* Os elementos desse array estão esgotados. A nova raiz será o   */
            /* último elemento do heap, que passará a ter um elemento a menos */
         raiz = ReduzHeapIM_Arr(&heap);

         /* Substitui a raiz do heap pelo novo elemento */
      SubstituiMinimoIM_Arr(&heap, &raiz, ComparaInts);
   }

   return resultado; /* Retorna o array resultante da intercalação */
}
```

12.3.4 Análise

Teorema 12.2: O custo temporal do algoritmo de intercalação multíplice de arrays é $\theta(n \cdot k \cdot \log k)$.

Prova: Examinando-se o algoritmo apresentado na **Seção 12.4.1**, nota-se que seu custo temporal corresponde ao custo temporal de seu **Passo 4**, que é um laço executado nk vezes, que é igual ao número de elementos nos k arrays. Por outro lado, o custo do corpo desse laço é determinado pelo **Passo 4.4**, que, conforme foi visto na **Seção 10.2.5**, é $\theta(\log k)$. Logo o custo temporal do algoritmo é $\theta(n \cdot k \cdot \log k)$. ■

12.4 Intercalação Multíplice de Arquivos

12.4.1 Descrição

Para tornar o algoritmo de intercalação binária mais eficiente, deve-se notar que, enquanto se usa toda a memória principal disponível na **Fase 1**, apenas três de cada m blocos de memória principal disponível são usados na **Fase 2**. É fácil ver que alocar arrays maiores B_1, B_2 e B_0 não altera o número de operações de entrada ou saída disponíveis. Em vez disso, mais séries precisam ser intercaladas ao mesmo tempo.

Um algoritmo de **intercalação multíplice** (ou **de múltiplas vias**) recebe como entrada vários arquivos ordenados e intercala-os num único arquivo ordenado usando, tipicamente, um número de vias de intercalação bem maior do que dois. Além disso, a ideia que norteia intercalação multíplice é a mesma que norteia intercalação binária. Entretanto, em vez de usar dois arrays de entrada de B elementos, usam-se $\lfloor N/M \rfloor$ arrays de entrada, cada um deles com B elementos. Cada array corresponde a uma série não concluída (ou **ativa**).

Inicialmente, todas as séries estão ativas e cada array tem um ponteiro[1] para seu primeiro elemento que ainda não foi levado em consideração. A **Figura 12–7** mostra como ocorre intercalação multíplice em memória principal.

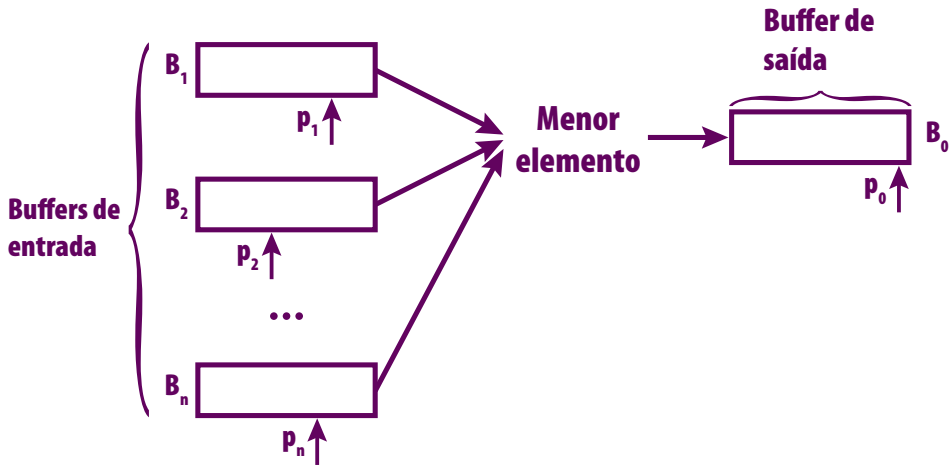

Figura 12–7: Intercalação Multíplice em Memória Principal

Uma intercalação multíplice permite que os arquivos em memória secundária sejam intercalados com um menor número de operações de entrada ou saída do que numa intercalação binária. Se houver seis séries que precisam ser intercaladas, uma intercalação binária usaria três intercalações, em oposição a uma única passagem de intercalação sêxtupla. Essa redução de passagens de intercalação é importante considerando a grande quantidade de dados que está usualmente sendo ordenada, permitindo maior rapidez ao mesmo tempo que também reduz a quantidade de acessos à memória secundária.

A intercalação multíplice é levada a efeito seguindo o algoritmo da **Figura 12–8**.

Algoritmo IntercalaçãoMultípliceDeArquivos

Entrada: Séries contendo registros ordenados

Saída: Arquivo contendo todos os registros das séries ordenados

1. Preencha os buffers de entrada com blocos de registros de suas respectivas séries
2. Enquanto houver buffer de entrada ativo, faça o seguinte:
   - **2.1** Encontre o menor registro que se encontra nos buffers de entrada
   - **2.2** Mova esse registro para a primeira posição disponível no buffer de saída
   - **2.3** Se o buffer de saída estiver cheio:
     - **2.3.1** Escreva o conteúdo do buffer de saída no arquivo resultante
     - **2.3.2** Reinicie o buffer de saída
   - **2.4** Se o buffer de entrada do qual o menor registro foi obtido ficar vazio:
     - **2.4.1** Se ainda houver bloco disponível na série correspondente, preencha esse buffer de entrada com o próximo bloco de registros
     - **2.4.2** Caso contrário, se não houver bloco disponível na série correspondente, considere desativado esse buffer de entrada
3. Se houver registros remanescentes no buffer de saída, escreva o conteúdo desse buffer no arquivo resultante

Figura 12–8: Algoritmo de Intercalação Multíplice de Arquivos

[1] No presente contexto, ponteiro é uma variável que armazena um índice de array.

12.4.2 Exemplo

Considere um exemplo fictício de ordenação externa por intercalação multíplice no qual deseja-se ordenar um arquivo contendo *24* registros, mas apenas *8* deles cabem inteiramente na memória principal disponível[2]. Os passos seguidos para ordenação desse arquivo serão descritos abaixo.

1. São lidos *8* registros do arquivo em memória principal. Em seguida, esses registros são ordenados por algum método convencional de ordenação visto no **Capítulo 11**, tal como **QuickSort**.
2. Os registros ordenados no passo anterior constituem a primeira série do processo. Essa série é escrita num novo arquivo em disco. A **Figura 12–9** mostra o resultado dos dois primeiros passos. A despeito de essa figura parecer mostrar que em memória principal há dois arrays, de fato, em qualquer instante, há apenas um array.

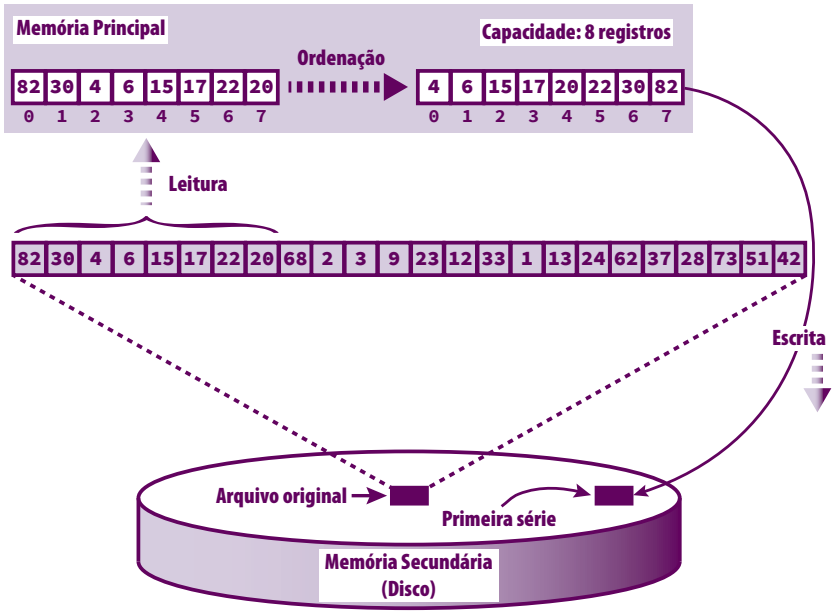

Figura 12–9: Exemplo de Intercalação Multíplice: Ordenação de Série

3. Os **Passos 1** e **2** são repetidos até que todos os registros estejam ordenados em novos arquivos de *8* registros. Como há *24* registros, haverá *3* novos arquivos ordenados (séries), como mostra a **Figura 12–10**.

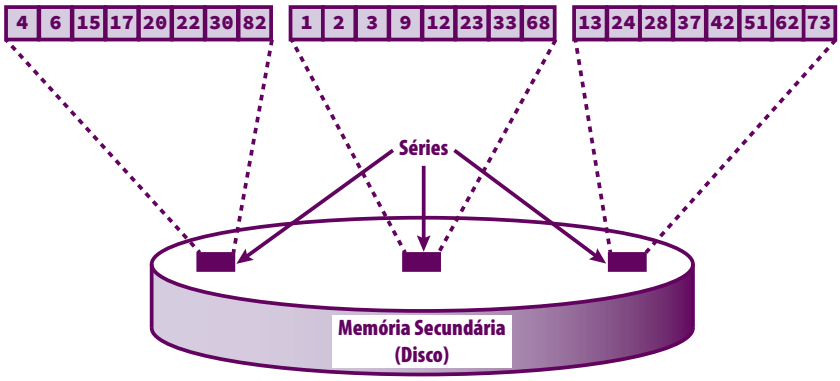

Figura 12–10: Exemplo de Intercalação Multíplice: Arquivos Contendo as Séries

[2] Não esqueça que esse é um exemplo fictício que não tem nenhum compromisso com as dimensões envolvidas. Portanto não tente imaginar uma situação real correspondente a esse exemplo. Em vez disso, concentre-se em entender o funcionamento do processo, que é real.

4. Para efetuar a intercalação das séries, são alocados *4* arrays em memória principal. Três desses arrays são buffers de entrada enquanto o outro array é um buffer de saída. Cada buffer de entrada está associado a uma série armazenada em arquivo. Por sua vez, o buffer de saída estará associado a um novo arquivo que conterá o resultado da intercalação dessas séries. A **Figura 12–11** mostra os buffers de entrada *E1*, *E2* e *E3* e o buffer de saída *S*. Como, por hipótese, a capacidade total da memória principal é de *8* registros, cada buffer tem capacidade para conter apenas dois registros.

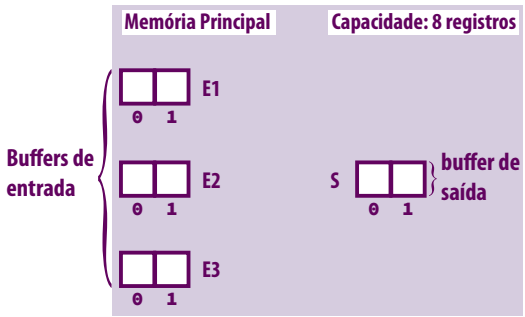

Figura 12–11: Exemplo de Intercalação Multíplice: Buffers de Entrada e Saída

5. Os primeiros dois registros de cada série são lidos e armazenados no respectivo buffer de entrada associado à série. A **Figura 12–12** ilustra a situação após a execução desse passo. Note que acima de cada buffer existe uma seta para baixo. Cada uma dessas setas representa a posição do **elemento ativo** corrente do respectivo buffer. O menor elemento ativo corrente nos buffers de entrada é aquele que será escrito na posição do elemento ativo corrente do buffer de saída. Cada seta para baixo que aparece sobre uma série na **Figura 12–12** indica a posição do apontador de arquivo dessa série.

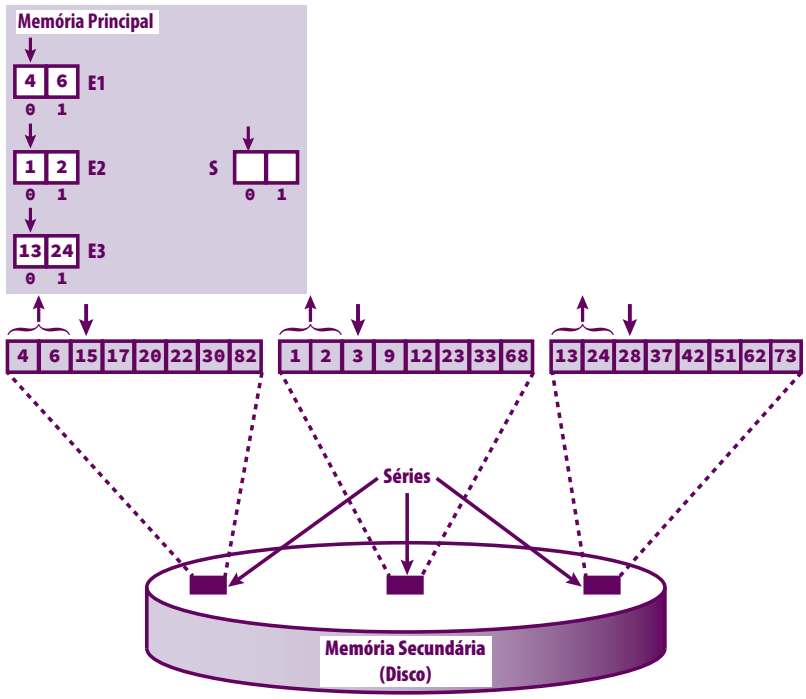

Figura 12–12: Exemplo de Intercalação Multíplice: Leitura de Registros de Séries

6. Efetua-se uma intercalação de três vias e armazena-se o resultado no buffer de saída. A **Figura 12–13** ilustra a intercalação dos registros contendo as chaves *4*, *1* e *13*. Nesse caso, como a menor chave é *1*, o registro que contém essa chave é copiado para o buffer de saída.

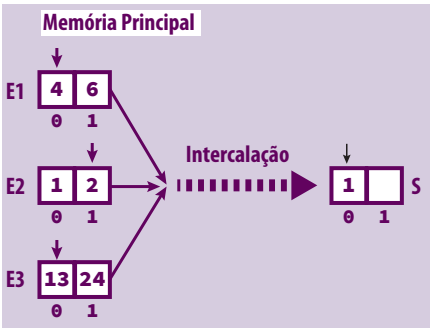

Figura 12–13: Exemplo de Intercalação Multíplice: Intercalação de Três Registros

7. Sempre que um elemento de um buffer de entrada é copiado para o buffer de saída, o indicador de elemento ativo desse buffer é incrementado, e o mesmo ocorre com o indicador de elemento ativo do buffer de saída (v. **Figura 12–13**). Quando o indicador de um buffer de entrada ultrapassa o limite do buffer (array), ele é considerado um **buffer de entrada vazio**, visto que ele não possui mais nenhum elemento ativo. Por outro lado, quando o indicador do buffer de saída ultrapassa o limite desse buffer, ele é considerado **repleto**, pois todas as suas posições já foram preenchidas. Sempre que o buffer de saída se torna repleto, ele é escrito no arquivo que conterá o resultado da intercalação, como mostra a **Figura 12–14**. Como no presente exemplo essa é a primeira escrita efetuada após o buffer se tornar repleto, um novo arquivo é criado. Após a **descarga do buffer de saída** em arquivo, seu indicador de elemento ativo volta para o início do buffer, de modo que ele é considerado vazio novamente (v. **Figura 12–15**).

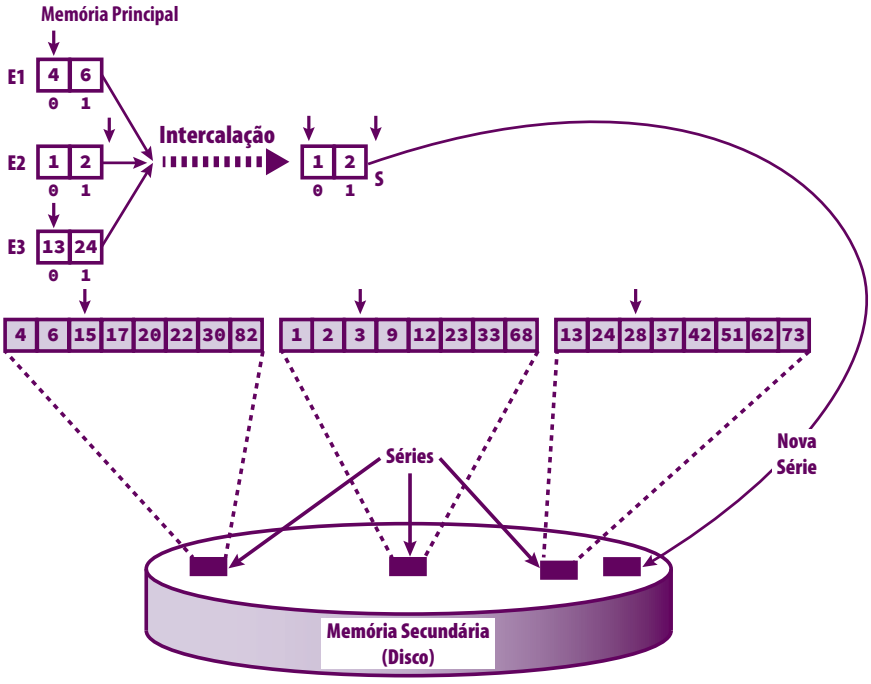

Figura 12–14: Exemplo de Intercalação Multíplice: Escrita de Nova Série

8. Sempre que qualquer buffer de entrada ficar vazio, ele será preenchido com os próximos registros de sua série de entrada associada até que não haja mais dados disponíveis nessa série. Na **Figura 12–15**, os registros com chaves *3* e *9* da segunda série de entrada foram lidos e usados para **recarregar o buffer de entrada** *E2* associado a essa série.

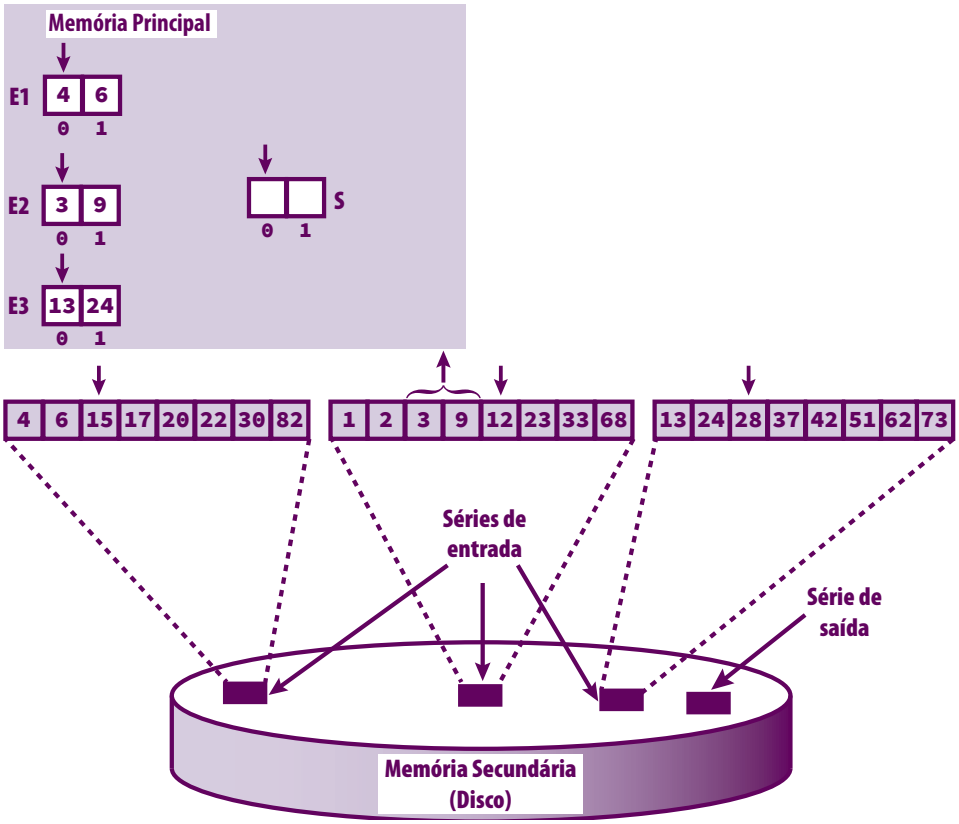

Figura 12–15: Exemplo de Intercalação Multíplice: Recarga de Buffer de Entrada

9. A **Figura 12–16** mostra a próxima descarga do buffer de saída após duas intercalações de registros.

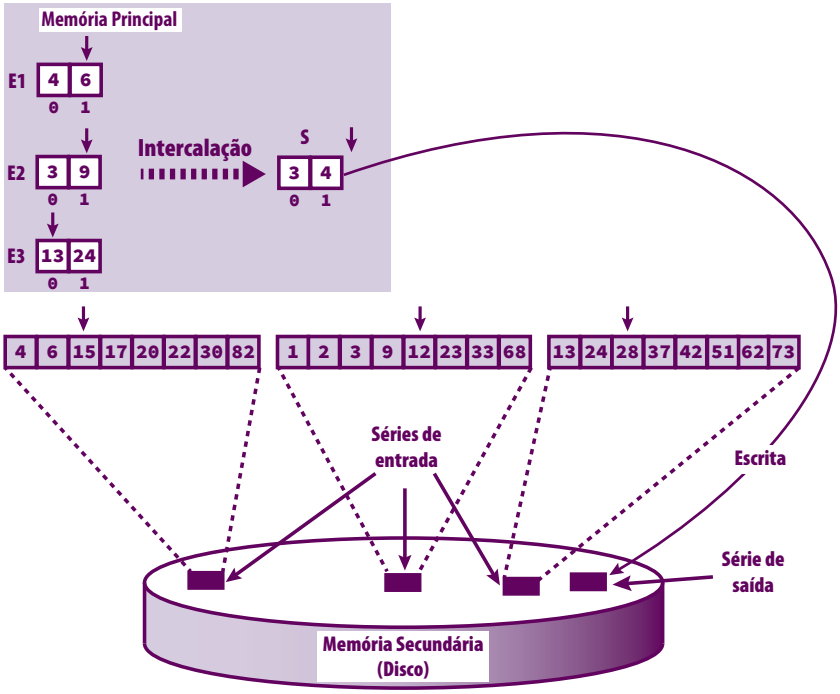

Figura 12–16: Exemplo de Intercalação Multíplice: Descarga de Buffer de Saída

A partir deste ponto, a história se repetirá e espera-se que o leitor tenha adquirido cabedal suficiente para completo entendimento do exemplo. Ao final desse exemplo, todas as séries terão sido intercaladas e esses arquivos podem ser removidos do disco. A série de saída, por outro lado, conterá o arquivo original ordenado.

O exemplo acima ilustra uma ordenação de dois passos: (1) ordenação (i.e., criação das séries ordenadas de entrada) e (2) intercalação, como ocorre com intercalação binária. Note, contudo, que, aqui, houve uma única passagem de intercalação múltipla em três vias. Se intercalação binária tivesse sido usada, seriam necessárias duas passagens de intercalação. O ganho nesse exemplo foi pequeno, mas, na prática, é comum ocorrerem várias intercalações simultâneas e não apenas três como foi o caso nesse exemplo.

A título de comparação, considere uma intercalação de seis séries ordenadas. Numa intercalação binária haveria 5 passagens [v. **Figura 12–17 (a)**], enquanto numa intercalação multíplice haveria apenas uma passagem [v. **Figura 12–17 (b)**].

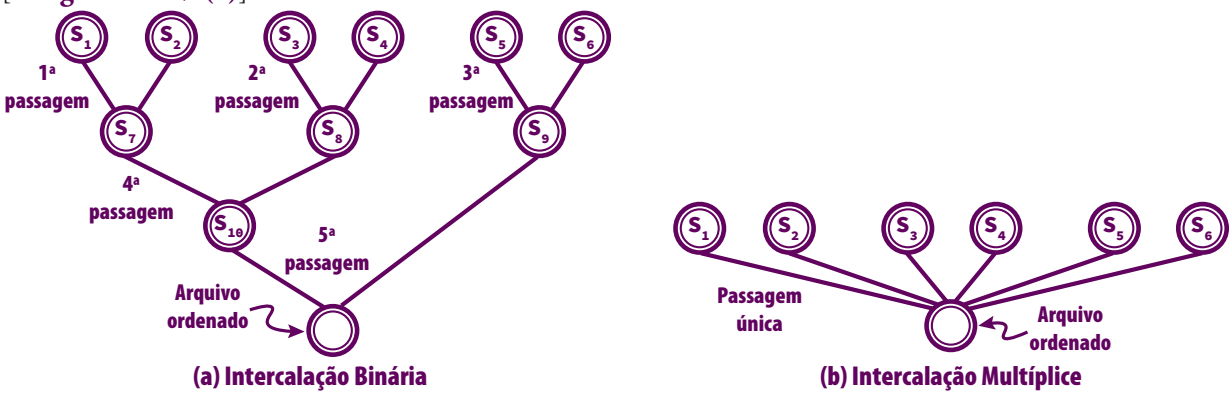

Figura 12–17: Passagens de Intercalação Binária e Multíplice

Existe uma limitação para intercalação numa única passagem. Isto é, quando o número de séries cresce muito, divide-se o espaço disponível em memória principal num número maior de buffers de tamanhos cada vez menores, de modo que um número maior de leituras menores precisa ser feito. Assim, por exemplo, ordenar 50 GiB usando 100 MiB de RAM com uma única passagem de intercalação não é eficiente, pois os acessos a disco requeridos para preencher os buffers de entrada com dados de cada uma das 500 séries de 200 KiB de cada vez consome muito tempo. Agora usando-se duas passagens de intercalação, resolve-se o problema.

12.4.3 Implementação

A implementação de intercalação multíplice usa os seguintes tipos:

```
   /* Tipo de variável que contém informação sobre uma série */
typedef struct {
          long          pos; /* Posição da próxima leitura */
          tRegistroMEC *buffer; /* Buffer associado à série */
          int           nRegs; /* Número de registros que se encontram no buffer */
          int           resto; /* Número de registros da série  */
                               /* que restam em arquivo       */
       } tSerieMS;

   /* Tipo de nó de um heap */
typedef struct {
          tRegistroMEC *elemento; /* Ponteiro para um registro de uma série */
          int           ir; /* Índice do registro em seu buffer */
          int           is; /* Índice da série à qual o registro pertence */
       } tNoHeapIM_Arq;
```

```
   /* Tipo de um heap */
typedef struct {
          tNoHeapIM_Arq *itens;  /* Array de elementos do heap   */
          int            nItens; /* Número de elementos do array */
       } tHeapIM_Arq;
```

O tipo `tRegistroMEC` usado nas definições acima é definido no **Apêndice A**. Além desses tipos, usa-se ainda nesta implementação o tipo `tFCompara` definido na **Seção 12.3.3**.

A implementação da **Fase 1** de intercalação multíplice é idêntica àquela que cria as séries iniciais da intercalação binária vista na **Seção 12.2.2**. Por sua vez, a **Fase 2** de intercalação multíplice é implementada pela função `IntercalaMultiArq()` cujos parâmetros são:

- `nomeE` (entrada) — nome do arquivo que contém as séries
- `nomeS` (entrada) — nome do arquivo resultante da intercalação
- `nSeries` (entrada) — o número de séries que serão intercaladas
- `tamSerie` (entrada) — tamanho máximo de uma série

```
void IntercalaMultiArq(const char *nomeE, const char *nomeS, int nSeries, int tamSerie)
{
   tSerieMS     *series; /* Ponteiro para um array contendo */
                         /* informações sobre as séries     */
   int           tamBuffer; /* Tamanho de um buffer de entrada ou saída */
   FILE         *streamS, /* Stream associado ao arquivo que */
                          /* conterá os registros ordenados  */
                *streamE; /* Stream associado ao arquivo que contém as séries */
   tHeapIM_Arq   heap; /* Heap usado na ordenação dos elementos */
                       /* dos buffers de entrada                */
   int           i,
                 ibs = 0, /* Índice no qual o próximo registro  */
                          /* do buffer de saída será armazenado */
                 iHeap = 0; /* Índice de um elemento do heap */
   tNoHeapIM_Arq raiz; /* Uma raiz do heap */
   tRegistroMEC *bufferS; /* Apontará para o buffer de saída */

   tamBuffer = MAX_REGISTROS/(nSeries + 1); /* Calcula o tamanho de cada buffer */

      /* O tamanho de um buffer não pode ser zero */
   ASSEGURA(tamBuffer, "O tamanho de cada buffer e' zero");

      /* Aloca espaço para o buffer de saída */
   bufferS = calloc(tamBuffer, TAM_REG);
   ASSEGURA(bufferS, "Impossivel alocar buffer de saida");

   IniciaHeapIM_Arq(&heap, nSeries); /* Inicia o heap */

      /* Tenta abrir o arquivo contendo as séries que serão intercaladas */
   streamE = AbreArquivo(nomeE, "rb");

      /* Aloca um array que contém informações sobre as séries */
   series = calloc(nSeries, sizeof(tSerieMS));
   ASSEGURA(series, "Impossivel alocar array de series");

      /* Aloca espaço para cada buffer de entrada, preenche cada um */
      /* com o conteúdo inicial de sua respectiva série e associa o */
      /* primeiro registro de cada buffer a um nó do heap           */
   for (i = 0; i < nSeries; ++i) {
         /* Aloca espaço para o buffer de entrada corrente */
      series[i].buffer = calloc(tamBuffer, TAM_REG);
```

```
   ASSEGURA( series[i].buffer, "Impossivel alocar um buffer de entrada" );

      /* Obtém a posição do primeiro byte da série */
   series[i].pos = i*tamSerie*TAM_REG;

      /* Preenche o conteúdo do buffer de entrada corrente */
   series[i].nRegs = CarregaBufferMS( streamE, series[i].buffer,
                                      tamBuffer, &series[i].pos );

      /* Obtém o número de registros restantes na série */
   series[i].resto = tamSerie - series[i].nRegs;

      /* Associa dados do primeiro registro do buffer corrente a um nó do heap */
   heap.itens[iHeap].elemento = series[i].buffer;/* Endereço do registro */
   heap.itens[iHeap].ir = 0; /* Seu índice no buffer */
   heap.itens[iHeap].is = i; /* Índice de sua série */

   ++iHeap; /* Passa para o próximo elemento do heap */
}

   /* Ordena o heap */
for (i = (nSeries - 1)/2; i >= 0; --i)
   OrdenaHeapIM_Arq(&heap, i, ComparaInts2);

   /* Abre o arquivo que conterá o resultado da ordenação */
streamS = AbreArquivo(nomeS, "wb");

   /* Obtém o menor elemento do heap e o substitui pelo próximo elemento */
   /* da série que o contém até que todas as séries estejam esgotadas    */
while (!SeriesEstaoVaziasMS(series, nSeries, &heap)) {
   raiz = ObtemMinimoIM_Arq(&heap); /* Obtém o menor valor armazenado no heap */

      /* Verifica se o buffer de saída está repleto */
   if (ibs > tamBuffer - 1) {
         /* O buffer de saída está repleto e é necessário  */
         /* descarregá-lo no stream que contém o resultado */
      fwrite( bufferS, TAM_REG, tamBuffer, streamS );
      ASSEGURA(!ferror(streamS), "Erro de escrita 1");

      ibs = 0; /* Reinicia o buffer de saída */
   }

      /* Armazena o menor valor do heap no buffer */
      /* que conterá o resultado da intercalação  */
   bufferS[ibs++] = *raiz.elemento;

      /* O elemento que substituirá a raiz pertence */
      /* à mesma série à qual pertencia a raiz      */
   if (++raiz.ir < series[raiz.is].nRegs) {
         /* Obtém o próximo elemento da série à qual a raiz pertencia */
      raiz.elemento = &series[raiz.is].buffer[raiz.ir];
   } else { /* Esse buffer está esgotado */
         /* Tenta renovar o buffer de entrada  */
      series[raiz.is].nRegs = CarregaBufferMS(streamE, series[raiz.is].buffer,
                                              MIN(tamBuffer, series[raiz.is].resto),
                                              &series[raiz.is].pos);

         /* Calcula o restante da série em arquivo */
      series[raiz.is].resto = series[raiz.is].nRegs
                            ? series[raiz.is].resto - series[raiz.is].nRegs
                            : 0;
```

```
            /* Se o buffer foi renovado, obtém os dados do seu primeiro registro */
         if (series[raiz.is].nRegs > 0) {
            raiz.elemento = series[raiz.is].buffer;
            raiz.ir = 0;
         } else {
            /* O buffer não foi renovado porque todos os registros de sua série */
            /* já foram consumidos. A nova raiz será o último elemento do heap, */
            /* que passará a ter um elemento a menos.                           */

               /* Teste de consistência */
            ASSEGURA(!series[raiz.is].resto, "Nao deveria haver mais registros");

            raiz = ReduzHeapIM_Arq(&heap); /* O heap terá um elemento a menos */
         }
      }

         /* Substitui a raiz do heap pelo novo elemento e reordena o heap */
      SubstituiMinimoIM_Arq(&heap, &raiz, CompараInts2);
   }

      /* Verifica se restam registros no buffer de saída */
   if (ibs > 0) {
         /* Escreve os registros restantes no stream que contém o resultado */
      fwrite( bufferS, TAM_REG, ibs, streamS );
      ASSEGURA(!ferror(streamS), "Erro de escrita 2");
   }

      /* Libera os buffers de entrada */
   for (i = 0; i < nSeries; ++i)
      free(series[i].buffer);

   free(series); /* Libera array com informações sobre as séries */

   free(bufferS); /* Libera o buffer de saída */

   FechaArquivo(streamE, nomeE); /* Fecha os arquivos */
   FechaArquivo(streamS, nomeS);
}
```

A função `IntercalaMultiArq()` chama as funções `CarregaBufferMS()`, discutida na **Seção 12.2.2**, e `SeriesEstaoVaziasMS()`. Essa última função verifica se todos os registros já foram processados e seus parâmetros são:

- `series` (entrada) — lista indexada contendo informações sobre as séries
- `nSeries` (entrada) — número de séries
- `*heap` (entrada) — o heap usado na intercalação

A função `SeriesEstaoVaziasMS()` retorna `1`, se os registros já foram todos processados, ou `0`, em caso contrário.

```
static int SeriesEstaoVaziasMS(tSerieMS *series, int nSeries, tHeapIM_Arq *heap)
{
   int i;

   for (i = 0; i < nSeries; ++i)
      if (series[i].nRegs > 0 && series[i].resto > 0)
         return 0;

   return HeapVazioIM_Arq(heap) ? 1 : 0;
}
```

12.4.4 Análise

A **Tabela 12–2** mostra dados comparativos de desempenhos das implementações de intercalações binária e multíplice apresentadas acima. Essa tabela leva em consideração apenas a **Fase 2** de cada respectiva implementação, que é aquela que efetivamente efetua as intercalações. Note como, em termos de tempo de processamento, a intercalação multíplice é de fato bem mais eficiente do que a intercalação binária.

# Registros	# Series	Intercalação Binária	Intercalação Multíplice
600.000	*8*	*2,00 s*	*1,00 s*
1.200.000	*15*	*6,00 s*	*1,00 s*
2.400.000	*30*	*1,12 m*	*8,00 s*
4.800.000	*60*	*4,08 m*	*36,00 s*
9.565.483	*120*	*8,75 m*	*2,73 m*

Tabela 12–2: Intercalação Binária versus Intercalação Multíplice

Teorema 12.3: O custo de transferência do algoritmo de intercalação multíplice é $\theta(n)$.

Prova: A **Fase 2** do algoritmo de intercalação multíplice descrito na **Figura 12–8** executa $\theta(n)$ operações de entrada ou saída e esse é o custo temporal do algoritmo inteiro. ■

O algoritmo de intercalação multíplice tem uma limitação: ele não pode ordenar arquivos arbitrariamente grandes. Ou seja, se a **Fase 1** do algoritmo produzir um número demasiadamente grande de séries, não será possível intercalar todas elas de uma vez na **Fase 2**. Isso ocorre porque cada série requer um buffer de entrada e, além disso, um buffer de saída também é requerido. Idealmente, o tamanho de cada um desses buffers deve ser, pelo menos, igual ao tamanho de um bloco de disco. Logo, supondo que o tamanho de um bloco seja B, o número de séries seja ns e o tamanho de um buffer seja tb, deve-se ter:

$$tb = \frac{M}{ns+1} \geq B \Rightarrow ns \leq \frac{M}{B} - 1 \qquad \textbf{Fórmula 12–1}$$

Como $ns = \lceil N/M \rceil \geq N/M$, tem-se que o tamanho máximo de um arquivo que pode ser ordenado pelo algoritmo da **Figura 12–8** é dado por:

$$\frac{N}{M} \leq ns \leq \frac{M}{B} - 1 \Rightarrow N \leq \frac{M^2}{B} - M \approx \frac{M^2}{B} \qquad \textbf{Fórmula 12–2}$$

Uma maneira óbvia de estender o algoritmo de intercalação multíplice de duas fases para suportar arquivos de qualquer tamanho é fazer várias iterações na **Fase 2** do algoritmo, em vez de apenas uma. Isto é, usa-se o algoritmo de intercalação em memória secundária descrito na **Figura 12–2**, mas, em vez de usar intercalação binária, usa-se intercalação multíplice (como foi descrito nesta seção) para intercalar $M/B - 1$ séries. Então, após cada iteração do laço principal da **Fase 2**, o número de séries é reduzido de $M/B - 1$.

A **Fase 1** executa $\theta(n)$ operações de entrada ou saída, e o mesmo ocorre com cada iteração do laço principal da **Fase 2**. Depois da **Fase 1**, inicia-se com $\lceil N/M \rceil = \lceil n/m \rceil$ séries, cada iteração do laço principal da **Fase 2** reduz de $m - 1$ o número de séries e encerra-se quando se tem apenas uma série. Assim há $\log_{m-1}(n/m)$ iterações do laço principal da **Fase 2**. Consequentemente o custo temporal total do algoritmo é:

$$\theta(n \cdot \log_{m-1}(n/m)) = \theta(n \cdot \log_m n - n \cdot \log_m m) = \theta(n \log m\, n - n) = \theta(n \cdot \log_m n)$$

Outra abordagem similar para ordenação de arquivos arbitrariamente grandes consiste em utilizar um número de fases (ou **passos**) maior do que dois. Por exemplo, suponha que sejam produzidas tantas séries na **Fase 1**, que não se pode alocar pelo menos um bloco para cada série na **Fase 2**. Então, primeiro, produzem-se séries muito maiores usando a intercalação multíplice de duas fases descrita na **Figura 12–8**. Nesse caso, cada série produzida terá comprimento máximo $M \times M/B$. Na **Fase 3**, intercalam-se essas séries, sendo que o número máximo de séries que podem ser processadas nessa fase é M/B, cada uma delas com tamanho M^2/B. Assim, usando um algoritmo com três passos, arquivos de tamanho M^3/B^2 bytes podem ser ordenados.

Em geral, pode-se ordenar um arquivo de tamanho N em p passos quando o valor de N é dado por:

$$N = M^p/B^{p-1}$$

Aplicando-se log_B em ambos os lados dessa última equação, obtém-se uma estimativa para o número de passos necessários p:

$$log_B N = p \cdot log_B M - (p-1) \cdot log_B B \Rightarrow p \approx log_B N / log_B M = log_M N$$

Assim o custo de transferência é $\theta(N \cdot log_M N)$.

Como exemplo, uma intercalação multíplice usando *0,5 MiB* de RAM e tamanho de bloco de *4 KiB* é capaz de conter *128* blocos em memória principal de uma vez. Isso permitiria que *127* séries fossem intercaladas num único passo. Um arquivo com tamanho igual a *128 MiB* poderia ser ordenado em dois passos: um para construir as séries iniciais e outro para intercalá-las. Por outro lado, um arquivo com *16 GiB* poderia ser ordenado em apenas três passos. (Como exercício, verifique a validade dessas afirmações.)

12.5 Limite Inferior para Ordenação em Memória Secundária

O **Teorema 12.4** define o limite inferior para qualquer algoritmo de ordenação externa.

Teorema 12.4: Ordenar n registros armazenados em memória secundária requer pelo menos:

$$\Omega\left(\frac{n}{B} \cdot \frac{\log(n/B)}{\log(M/B)}\right)$$

transferências de bloco, em que M é o tamanho da memória interna e B é o tamanho de um bloco.

Prova: Se for possível executar o processo de intercalação usando apenas $O(n/B)$ transferências de disco, então para valores suficientemente grandes de n, o número total de transferências executadas por esse algoritmo satisfaz a seguinte relação de recorrência:

$$T(n) = d \cdot T(n/d) + c \cdot n/B$$

para alguma constante $c \geq 1$ e sendo $d = M/B - 1$. Pode-se encerrar a recursão quando $n \leq B$, pois, nesse ponto, pode-se executar uma única transferência de bloco, obter todos os itens que se encontram em memória principal e, então, ordenar o conjunto com um algoritmo eficiente para memória interna. Assim o caso base da relação de recorrência é:

$$T(n) = 1 \text{ se } n/B \leq 1$$

O resultado dessa relação de recorrência, cuja resolução está além do escopo deste livro, implica que $T(n)$ é $O((n/B)\ log_d (n/B))$, que é:

$\Omega((n/B)\ log(n/B)/\ log\ M/B)$ ■

A razão *M/B* é o número de blocos de memória secundária que podem caber em memória interna (v. **Tabela 6–3**). Assim o **Teorema 12.4** diz que o melhor desempenho que se pode obter para ordenação externa equivale àquele obtido acessando todos os dados de entrada, o que consome $\theta(n/B)$ transferências, pelo menos um número logarítmico de vezes, em que a base desse logaritmo é o número de blocos que cabe em memória interna.

12.6 Inserção Massiva em Árvores B+ (Bulkloading)

12.6.1 Conceito

Frequentemente, tem-se um grande conjunto de registros e deseja-se construir uma árvore B+ para indexá-los. Evidentemente, pode-se começar com uma árvore B+ vazia e inserir um registro de cada vez usando algoritmo de inserção em árvores B+ descrito na **Figura 6–37**. Entretanto essa abordagem não é eficiente, pois, inserindo um par chave/índice de cada vez na árvore, é bem provável que cada folha acessada numa inserção esteja num bloco diferente no arquivo que contém a árvore, o que pode requerer uma operação de leitura para cada inserção. Como provavelmente essa folha não permanecerá em memória cache, uma operação de escrita via acesso direto poderá ser requerida para atualizá-la antes da leitura da próxima folha. Desse modo uma operação de leitura e outra de escrita por meio de acesso direto podem ser necessárias para a inserção de cada chave. Por exemplo, se um arquivo contém 10 milhões de registros e cada operação de leitura ou escrita via acesso direto leva cerca de 10 milissegundos, a construção de uma árvore B+ inserindo chaves individualmente poderia demorar mais de 27 horas.

Inserção massiva (*bulk loading* ou *bulkloading*, em inglês) é o processo de inserção numa tabela de busca de uma grande quantidade de chaves numa única operação. Esse termo é usado principalmente quando a tabela de busca é implementada usando árvores B+, pois, para tais implementações, existem algoritmos específicos. Muitos sistemas de gerenciamento de bancos de dados implementam inserção massiva para reduzir o custo de criação de índices.

Em resumo, o mecanismo de inserção massiva para criação de uma árvore B+ consiste em três passos:

1. Criação de um arquivo temporário contendo os pares chave/índice que serão inseridos nas folhas da árvore B+
2. Ordenação desse arquivo usando a chave de cada par como chave de ordenação
3. Construção da árvore B+

Como o segundo passo de inserção massiva é a ordenação do arquivo contendo os pares chave/índice que serão inseridos numa árvore, esse tema foi adiado para este capítulo, em vez de ter sido discutido no **Capítulo 6**.

Usando *bulkloading* na construção de uma árvore B+, cada folha precisa ser escrita apenas uma vez. Além disso, nenhum nó, interno ou folha, precisa ser lido no arquivo que contém a árvore. Portanto muitas chaves podem ser inseridas numa árvore B+ ao custo de apenas uma operação de escrita. Suponha, por exemplo, que cada folha de uma árvore B+ contém *200* pares chave/índice e que o número total desses pares é *10* milhões. Então haverá *50.000* folhas que poderão ser escritas sequencialmente, o que é bem mais rápido do que via acesso direto (v. **Seção 1.1.3**). Considerando que uma operação de acesso sequencial a um bloco de disco leva *1* milissegundo (em vez de *10* ms para acesso direto), a escrita das folhas consumiria cerca de *50* segundos. Suponha ainda que os nós internos têm grau *200*. Como quase todos os nós internos são preenchidos pela metade (v. adiante) e a primeira chave de cada folha aparece no índice (exceto aquela mais à esquerda — v. **Seção 6.5.4**), o número de nós internos neste exemplo será aproximadamente *500* (*50.000/100*), o que requer cerca de *0,5* segundo de operações de escrita via acesso sequencial para armazenar esses nós internos em arquivo. Concluindo, o tempo consumido na construção da árvore B+ desse exemplo usando *bulkloading* seria aproximadamente *50* segundos.

12.6.2 Algoritmos

Como foi visto acima, a primeira fase de inserção massiva é relativamente trivial, pois consiste apenas em ler o arquivo de registros e obter os pares chave/índice que serão inseridos nas folhas da árvore B+. A segunda etapa dessa implementação é a ordenação do arquivo de pares que será usado na construção da árvore. Essa operação pode ser efetuada de modo eficiente usando a técnica de intercalação multíplice descrita na **Seção 12.4**. Resta a ser discutida a terceira etapa, o que será feito a seguir.

O algoritmo **CriaÁrvoreB+**, que representa a terceira etapa do mecanismo de inserção massiva para árvores B+, é apresentado na **Figura 12–18**.

Algoritmo CriaÁrvoreB+ComBulkloading

Entrada: Arquivo contendo pares chave/índice ordenados

Saída: Arquivo contendo a árvore B+ construída

1. Leia no arquivo ordenado de pares um número de pares suficiente para preencher a primeira folha da árvore mantida em memória principal (essa folha será denominada *folha esquerda*)
2. Escreva essa folha no arquivo da árvore
3. Se o final do arquivo de entrada foi atingido, torne essa folha a raiz da árvore e encerre
4. Crie um nó interno (raiz) em memória principal e faça com que seu primeiro filho aponte para a primeira folha
5. Crie uma lista indexada para armazenar os nós internos mantidos em memória principal (doravante denominada *lista de nós ativos*)
6. Enquanto o final do arquivo de pares não for atingido, faça o seguinte:
   - 6.1 Efetue uma leitura no arquivo de pares, preencha outra folha da árvore em memória principal (essa folha será denominada *folha direita*)
   - 6.2 Acrescente a folha direita ao arquivo da árvore e obtenha sua posição nesse arquivo
   - 6.3 Faça com que a folha esquerda aponte para a folha direita
   - 6.4 Atualize a folha esquerda no arquivo
   - 6.5 Insira a primeira chave da folha direita juntamente com a posição dessa folha em arquivo usando o algoritmo **InsereAcimaEmÁrvoreB+**
   - 6.6 Faça com que a folha esquerda passe a ser a folha direita
7. Atualize no arquivo da árvore todos os nós ativos

Figura 12–18: Algoritmo de Inserção Massiva em Árvore B+

O algoritmo **InsereAcimaEmÁrvoreB+** invocado pelo algoritmo **CriaÁrvoreB+** é apresentado na **Figura 12–19**.

Algoritmo InsereAcimaEmÁrvoreB+

Entrada: Lista de nós ativos, arquivo que contém a árvore, a chave a ser inserida e seu filho direito

Saída: Arquivo contendo a árvore B+

1. Para cada nó da lista de nós ativos, faça o seguinte:
   - 1.1 Se o nó não estiver repleto:
     - 1.1.1 Acrescente a chave e seu filho direito ao final do nó
     - 1.1.2 Encerre

Continua

Figura 12–19: Algoritmo InsereAcimaEmÁrvoreB+ de Bulkloading de Árvores B+

Algoritmo InsereAcimaEmÁrvoreB (Continuação)

1.2 Caso contrário:
  1.2.1 Divida o nó usando o algoritmo **DivideNoInternoB+**
  1.2.2 Armazene o nó dividido no arquivo
  1.2.3 Substitua o nó dividido pelo novo nó resultante da divisão na lista de nós ativos
2. Crie uma nova raiz tendo como filho esquerdo a antiga raiz (que foi dividida), como filho direito o novo nó resultante da divisão e como chave aquela que subiu para o próximo nível superior
3. Acrescente a nova raiz ao final da lista de nós ativos

Figura 12–19 (Cont.): Algoritmo InsereAcimaEmÁrvoreB+ de Bulkloading de Árvores B+

O algoritmo **DivideNoInternoB+** invocado pelo algoritmo **InsereAcimaEmÁrvoreB+** é semelhante ao algoritmo **DivideNo** para árvores B apresentado na **Seção 6.4.3**.

Note que, quando um nó é dividido, nenhuma chave é anexada ao nó que deu origem à divisão. Em outras palavras, essa chave sempre é inserida no novo nó resultante dessa divisão (i.e., no nó mais à direita de cada nível). Isso faz com que a maioria dos nós internos sejam preenchidos apenas pela metade.

12.6.3 Exemplo

Suponha que se tenha um arquivo de registros já ordenado. Esta seção apresentará um exemplo (fictício, como sempre) de criação de uma árvore B+ usando a técnica de inserção massiva descrita acima.

Depois de criar a primeira folha, aloca-se um novo nó para servir de raiz e faz-se o filho esquerdo desse nó apontar para essa folha. Em seguida, para cada nova folha acrescentada à árvore, inclui-se a menor chave dessa folha à raiz e faz-se o filho direito dessa chave apontar para essa última folha. Esse procedimento prossegue até que a raiz da árvore esteja completa. Nesse caso, a próxima chave a ser inserida nessa raiz causa uma divisão de nós com a consequente criação de uma nova raiz.

É interessante notar que, como as chaves estão ordenadas, o nó interno esquerdo resultante de uma divisão não terá mais chaves acrescidas a ele. Portanto esses nós podem ser definitivamente escritos no arquivo que armazena a árvore. O mesmo ocorre com folhas que já foram acrescentadas à árvore, de modo que, durante o processo de bulkloading, em memória principal haverá apenas um nó para cada nível da árvore, incluindo a última folha acrescentada à árvore. Esse fato é mostrado nas figuras dos exemplos a seguir. Nessas figuras, o sombreamento de alguns nós indica que eles já foram escritos em arquivo, enquanto os demais nós (um por nível) permanecem (temporariamente) em memória principal.

A **Figura 12–20** mostra a criação da primeira folha e do primeiro nó interno, que, por enquanto, é a raiz da árvore. Essa figura corresponde à execução dos **Passos 2** e **4** do algoritmo da **Figura 12–18**.

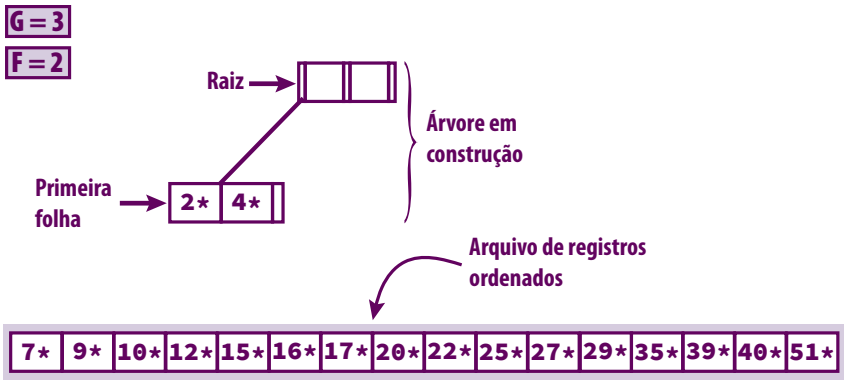

Figura 12–20: Exemplo de Inserção Massiva em Árvores B+ 1

A **Figura 12–21** mostra a criação da segunda folha que corresponde aos **Passos 6.1** a **6.4** do algoritmo da **Figura 12–18**. Nessa e nas demais figuras desse exemplo os números indicam valores de chaves, sendo que aqueles que são acompanhados com asterisco indicam que essas chaves estão acompanhadas de seus respectivos registros, mas apenas as chaves aparecem. Também, dados armazenados em memória secundária são circundados por um sombreado, ao passo que aqueles que se encontram em memória principal não apresentam esse sombreado.

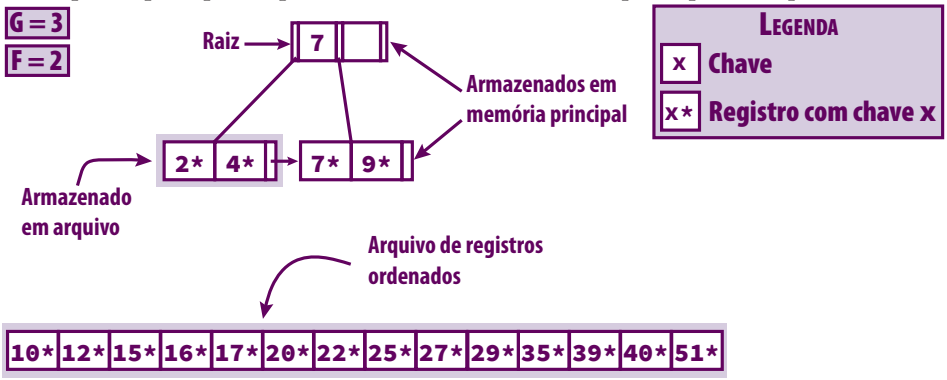

Figura 12–21: Exemplo de Inserção Massiva em Árvores B+ 2

A **Figura 12–22** mostra a criação da terceira folha seguida da inserção de sua primeira chave na raiz da árvore.

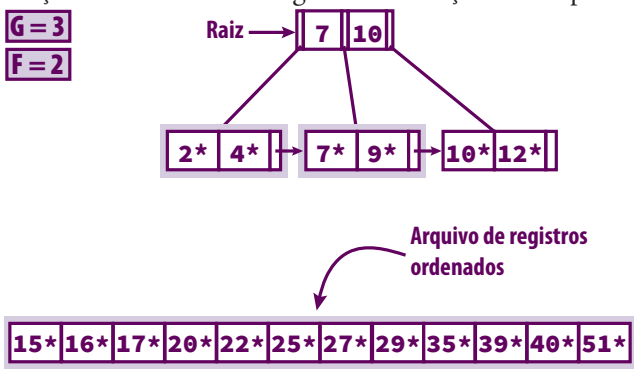

Figura 12–22: Exemplo de Inserção Massiva em Árvores B+ 3

Na criação da quarta folha da árvore do presente exemplo, mostrada na **Figura 12–23**, ocorre uma divisão da raiz dessa árvore. Note que, nesse instante, há três nós armazenados em memória principal e quatro nós em arquivo.

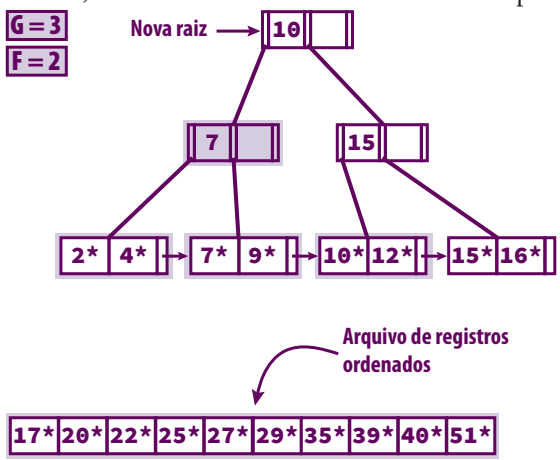

Figura 12–23: Exemplo de Inserção Massiva em Árvores B+ 4

As demais figuras exibidas a seguir ilustram o complemento do exemplo. Em caso de dúvida, o leitor é convidado a estudar novamente o **Capítulo 6** para dirimi-las.

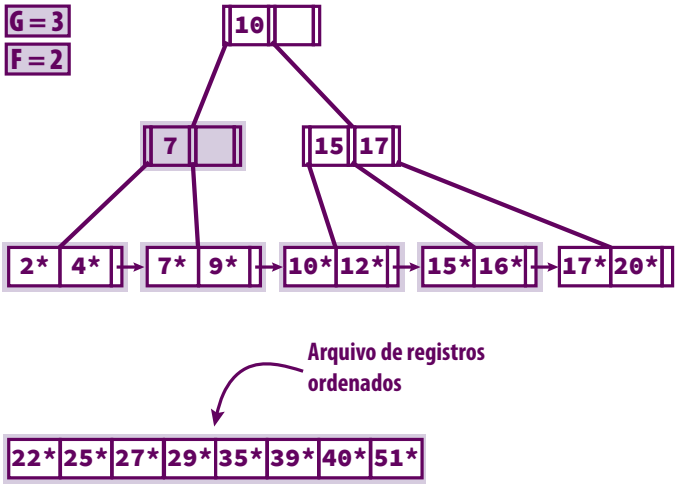

Figura 12–24: Exemplo de Inserção Massiva em Árvores B+ 5

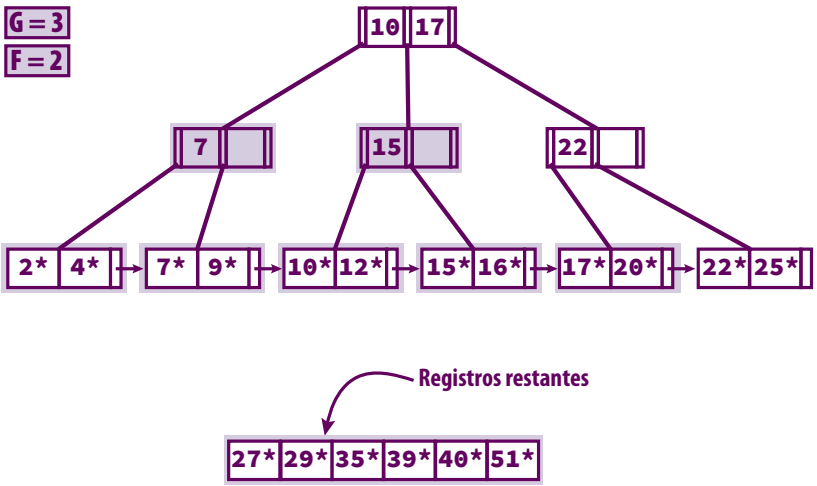

Figura 12–25: Exemplo de Inserção Massiva em Árvores B+ 6

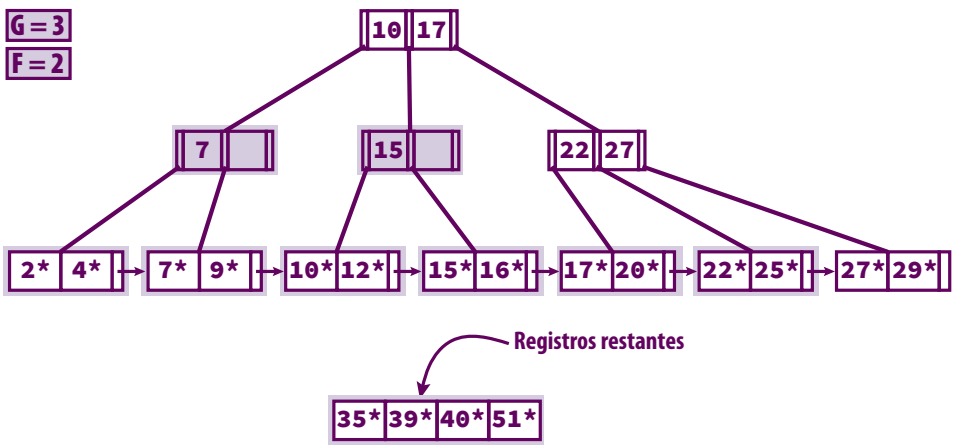

Figura 12–26: Exemplo de Inserção Massiva em Árvores B+ 7

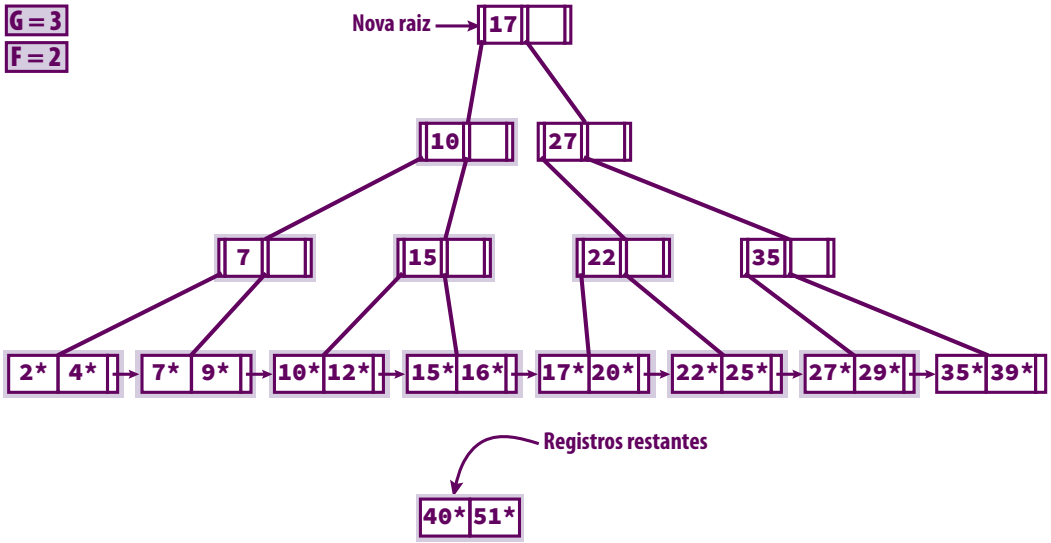

Figura 12–27: Exemplo de Inserção Massiva em Árvores B+ 8

Os dois últimos registros desse exemplo serão inseridos sem qualquer divisão de nós, de modo que todos os nós podem ser armazenados no arquivo que armazena a árvore.

12.6.4 Implementação

Além dos tipos usados na implementação de árvores B+ (v. **Seção 6.5.6**), a implementação de inserção massiva apresentada nesta seção utiliza o seguinte tipo de lista usada para guardar nós ativos durante o processo de bulkloading.

```
   /* Tipo de variável usada para armazenar um nó e sua posição em arquivo */
typedef struct {
          tNoBM  no;  /* O nó */
          int    pos; /* Posição do nó no arquivo que o contém */
       } tNoPos;

   /* Tipo de lista indexada usada para guardar    */
   /* nós ativos durante o processo de bulkloading */
typedef struct {
          tNoPos *nosPos; /* Array contendo nós ativos e suas */
                          /* respectivas posições em arquivo  */
          int     nNos;   /* Número de nós ativos */
          int     tamArray; /* Tamanho corrente do array */
       } tListaNosAtivos;
```

A função `CriaArvoreBM()` insere um conjunto de pares chave/índice ordenados numa árvore B+ usando o algoritmo descrito na **Figura 12–18**. Essa função tem como parâmetros:

- `streamOrd` (entrada) — stream associado ao arquivo contendo os pares ordenados
- `streamArv` (entrada) — stream associado ao arquivo que conterá a árvore
- `raiz` (saída) — ponteiro para a raiz da árvore em memória principal

A função `CriaArvoreBM()`, apresentada adiante, assume os seguintes pressupostos:

1. Ela não checa se há chaves repetidas
2. O arquivo que conterá a árvore deverá estar aberto num modo que permita escrita
3. Esse arquivo deve estar inicialmente vazio

```
static void CriaArvoreBM( FILE *streamOrd, FILE *streamArv, tNoBM *raiz )
{
   tChave          chaveQSobe; /* Chave a ser inserida num nó interno da árvore */
   tNoBM           folhaEsq, /* Folha esquerda ativa */
                   folhaDir; /* Folha direita ativa  */
   int             posFolhaEsq, /* Posição no arquivo de 'folhaEsq' */
                   posFolhaDir, /* Posição no arquivo de 'folhaDir' */
                   i;
   tListaNosAtivos nosAtivos; /* Lista de nós ativos */

      /* Garante que a leitura de pares começa no início do arquivo que os contém */
   MoveApontador(streamOrd, 0L, SEEK_SET);

   IniciaNoBM(FOLHA, &folhaEsq); /* Inicia a primeira folha da árvore */

      /* Copia pares para a primeira folha */
   folhaEsq.conteudo.noFolha.nChaves =  fread(folhaEsq.conteudo.noFolha.chaves,
                                           sizeof(tChaveIndice), F, streamOrd);

    ASSEGURA(!ferror(streamOrd), "CriaArvoreBM: erro de leitura");

      /* Se o final do arquivo foi atingido, a árvore terá */
      /* apenas um nó, que é a única folha preenchida      */
   if (feof(streamOrd)) {
         /* Torna a folha a raiz da árvore */
      *raiz = folhaEsq;

      EscreveNoBM(streamArv, 0, raiz); /* Escreve a folha (raiz) no arquivo */

      return; /* Serviço concluído */
   }

   IniciaNoBM(INTERNO, raiz); /* Inicia a raiz da árvore */

      /* A raiz da árvore será o primeiro nó do arquivo */
   EscreveNoBM(streamArv, 0, raiz);

      /* O primeiro filho da raiz apontará para a primeira folha */
   raiz->conteudo.noInterno.filhos[0] = posFolhaEsq =
         AcrescentaNoBM(streamArv, &folhaEsq);
   raiz->conteudo.noInterno.nFilhos = 1;

      /* Aloca um array para armazenar os nós internos ativos.  */
      /* O número mágico 5 é uma boa estimativa inicial, já que */
      /* não se espera que a árvore tenha altura maior do que 4 */
   nosAtivos.tamArray = 5;
   nosAtivos.nosPos = calloc(nosAtivos.tamArray, sizeof(tNoPos));
   ASSEGURA(nosAtivos.nosPos, "CriaArvoreBM(): Impos'sivel alocar lista de nos");

      /* Por enquanto, a raiz corrente é o único nó ativo */
   nosAtivos.nosPos[0].no = *raiz;
   nosAtivos.nosPos[0].pos = 0;
   nosAtivos.nNos = 1;

   IniciaNoBM(FOLHA, &folhaDir); /* Inicia a segunda folha da árvore */

      /* Lê os demais pares e acrescenta-os à árvore */
   while (1) {
         /* Copia pares para a folha direita */
      folhaDir.conteudo.noFolha.nChaves = fread(folhaDir.conteudo.noFolha.chaves,
                                              sizeof(tChaveIndice), F, streamOrd);
       ASSEGURA(!ferror(streamOrd), "Erro de leitura em CriaArvoreBM()");
```

```
         /* Se não foi lido nenhum par, encerra o laço */
      if (!folhaDir.conteudo.noFolha.nChaves)
         break;

         /* Acrescenta a folha direita ao arquivo */
         /* e obtém sua posição nesse arquivo     */
      posFolhaDir = AcrescentaNoBM(streamArv, &folhaDir);

         /* Efetua o encadeamento das folhas */
      folhaEsq.conteudo.noFolha.proximaFolha = posFolhaDir;
      folhaDir.conteudo.noFolha.proximaFolha = POSICAO_NULA;

         /* Atualiza a folha esquerda no arquivo */
      EscreveNoBM(streamArv, posFolhaEsq, &folhaEsq);

         /* A primeira chave do nó da direita será inserida num nó interno */
      chaveQSobe = folhaDir.conteudo.noFolha.chaves[0].chave;

         /* Insere a chave em algum nó interno da árvore */
      InsereAcimaBM(&nosAtivos, chaveQSobe, posFolhaDir, streamArv);

         /* A folha direita passará a ser a folha esquerda */
      folhaEsq = folhaDir;
      posFolhaEsq = posFolhaDir;
   }

      /* A raiz é o último nó da lista de nós ativos */
   *raiz = nosAtivos.nosPos[nosAtivos.nNos - 1].no;

      /* Atualiza no arquivo todos os nós ativos */
   for (i = 0; i < nosAtivos.nNos; ++i)
      EscreveNoBM(streamArv, nosAtivos.nosPos[i].pos, &nosAtivos.nosPos[i].no);

   free(nosAtivos.nosPos); /* A lista de nós ativos não é mais necessária */
}
```

A função `CriaArvoreBM()` chama a função `InsereAcimaBM()` que insere a primeira chave de uma folha num nó interno de uma árvore B+ e usa como parâmetros:

- `*nosAtivos` (entrada/saída) — lista de nós internos ativos (i.e., armazenados em memória principal)
- `chave` (entrada) — chave a ser inserida num nó do último nível de nós internos
- `pFilhoDireito` (entrada) — posição no arquivo do nó que será filho direito da chave a ser inserida
- `streamArvore` (entrada) — stream associado ao arquivo que contém a árvore

```
static void InsereAcimaBM( tListaNosAtivos *nosAtivos, tChave chave,
                           int pFilhoDireito, FILE *streamArvore )
{
   int    iChave, /* Índice da chave no nó no qual ela será inserida */
          i;
   tChave chaveQSobe; /* Chave a ser inserida um nível acima */
   tNoBM  noDireito, /* Nó direito que conterá as chaves */
                     /* superiores de um nó dividido     */
          antigaRaiz, /* Antiga raiz, se foi criada uma nova */
          novaRaiz;   /* Uma nova raiz */

      /* A lista não pode estar vazia */
   ASSEGURA(nosAtivos->nNos > 0, "InsereAcimaBM: lista de nos esta vazia");

   for (i = 0; i < nosAtivos->nNos; ++i) {
         /* O nó corrente deve ser um nó interno */
      ASSEGURA( nosAtivos->nosPos[i].no.tipoDoNo == INTERNO,
                "InsereAcimaBM: No' deveria ser interno" );
```

```
      /* A chave a ser inserida será a última chave do nó */
   iChave = nosAtivos->nosPos[i].no.conteudo.noInterno.nFilhos - 1;

      /* O índice de inserção da chave não pode ser negativo */
   ASSEGURA(iChave >= 0, "InsereAcimaBM: Indice de insercao e' negativo");

      /* Se o nó corrente não estiver repleto, insere nele */
   if (iChave < G - 1) {
         /* Efetua a inserção */
      nosAtivos->nosPos[i].no.conteudo.noInterno.chaves[iChave] = chave;
      nosAtivos->nosPos[i].no.conteudo.noInterno.filhos[iChave+1]=pFilhoDireito;

         /* O número de filhos do nó aumentou */
      nosAtivos->nosPos[i].no.conteudo.noInterno.nFilhos++;

         /* Atualiza o nó no arquivo */
      EscreveNoBM( streamArvore, nosAtivos->nosPos[i].pos,
                   &nosAtivos->nosPos[i].no );

      return; /* Serviço completo */
   }

       /* O nó corrente é completo e precisa ser dividido */
   DivideNoInternoBM( &nosAtivos->nosPos[i].no, iChave, chave,
                      pFilhoDireito, &noDireito, &chaveQSobe );

      /* Atualiza o nó dividido no arquivo */
   EscreveNoBM(streamArvore, nosAtivos->nosPos[i].pos, &nosAtivos->nosPos[i].no);

      /* O filho direito da chave que subirá para o nível superior será o */
      /* novo nó criado no nível corrente. Armazena este nó no arquivo e  */
      /* guarda sua posição neste arquivo.                                */
   pFilhoDireito = AcrescentaNoBM(streamArvore, &noDireito);

      /* Substitui na lista de nós ativos o nó */
      /* que foi dividido pelo novo nó direito */
   nosAtivos->nosPos[i].no = noDireito;
   nosAtivos->nosPos[i].pos = pFilhoDireito;
}
/****                                                                ****/
/* Neste ponto, sabe-se que o último nó divido no laço for era uma raiz   */
/* completa. Agora cria-se uma nova raiz contendo a chave que subiu tendo */
/* como filho esquerdo o nó dividido e como filho direito o novo nó       */
/****                                                                ****/

IniciaNoBM(INTERNO, &novaRaiz); /* Inicia a nova raiz */

novaRaiz.conteudo.noInterno.chaves[0] = chaveQSobe; /* Copia a chave */

   /* Lê a antiga raiz em arquivo, pois ela  */
   /* foi substituída na lista de nós ativos */
LeNoBM(streamArvore, 0, &antigaRaiz);

   /* O filho da esquerda da nova raiz é a antiga raiz que */
   /* precisa ser armazenada em outra posição no arquivo   */
   /* porque a raiz deve sempre ocupar a primeira posição  */
novaRaiz.conteudo.noInterno.filhos[0] =
     AcrescentaNoBM(streamArvore, &antigaRaiz);

   /* O filho direito da raiz já foi inserido antes no arquivo e sua */
   /*  posição neste arquivo está guardada em 'pFilhoDireito'        */
novaRaiz.conteudo.noInterno.filhos[1] = pFilhoDireito;
```

```
      /* A nova raiz tem dois filhos */
   novaRaiz.conteudo.noInterno.nFilhos = 2;

      /* Armazena a raiz da árvore a posição 0 do arquivo */
   EscreveNoBM(streamArvore, 0, &novaRaiz);

      /* Verifica se lista de nós ativos ainda tem espaço */
   if (nosAtivos->nNos == nosAtivos->tamArray) {
         /* Array está repleto. Dobra sua capacidade */
      nosAtivos->tamArray *= 2;
      nosAtivos->nosPos = realloc( nosAtivos->nosPos, nosAtivos->
                                   tamArray* sizeof(tNoPos) );
      ASSEGURA( nosAtivos->nosPos, "InsereAcimaBM: "
                "Impossivel redimensionar lista de nos ativos" );
   }

      /* Acrescenta a nova raiz à lista de nós ativos */
   nosAtivos->nosPos[nosAtivos->nNos].no = novaRaiz;
   nosAtivos->nosPos[nosAtivos->nNos].pos = 0;
   nosAtivos->nNos++;
}
```

12.6.5 Análise

Em resumo, o algoritmo visto na **Figura 12–18** apresenta as seguintes vantagens:

- Não é necessário descer a árvore para efetuar inserções
- Esse algoritmo requer um menor número de operações de entrada e saída durante a construção do que seria o caso se as inserções fossem individuais
- Folhas são armazenadas sequencialmente (e encadeadas, evidentemente), o que favorece o princípio de localidade de referência (v. **Capítulo 1**)
- Todas as folhas são completamente preenchidas, com exceção, talvez, da última delas, o que faz com que a árvore assim criada ocupe menos espaço

Se os registros já estiverem ordenados, o algoritmo de inserção massiva tem custo de transferência de $\theta(n/B)$. Caso contrário, esse algoritmo tem o mesmo custo da ordenação externa, que é $\theta(n/B \cdot \log_B n)$. Assim, mesmo que o algoritmo de inserção massiva execute uma ordenação primeiro, ele ainda é G vezes mais rápido do que inserir chaves individualmente numa árvore B+, sendo G o grau da árvore.

O primeiro passo de *bulkloading* consiste em ler todos os registros e escrever os pares chave/índice correspondentes em arquivo. O custo desse passo é $\theta(R + P)$, em que R é o número de blocos contendo os registros e P é o número de blocos contendo esses pares. Como foi visto na **Seção 12.4.4**, o custo de ordenação usando intercalação multíplice é $\theta(n/B \cdot \log_B n)$. O custo do terceiro passo é igual ao custo de escrita de todos os nós da árvore mais o custo de leitura de cada folha.

A única desvantagem aparente do método de inserção massiva descrito aqui é que os nós internos têm 50% de ocupação, com exceção do nó mais à direita de cada nível, que pode ter uma ocupação maior. Esse índice de ocupação está um pouco acima da média para árvores B e B+ (v. **Seção 6.4.8**) e isso faz com que uma árvore B+ criada dessa maneira possa ter uma altura um pouco maior do que poderia ter se ela fosse criada do modo tradicional.

12.7 Exemplos de Programação

12.7.1 (Pseudo–) Ordenação de Arquivos por Indexação

Preâmbulo: Um método antigo de ordenação de arquivos é denominado **ordenação por indexação**. Usando-se esse método, leem-se apenas as chaves de ordenação e emparelha-se cada chave com um índice que representa a posição no arquivo do registro que contém a respectiva chave. A vantagem desse método com relação à ordenação interna é a economia de memória, pois ordenar uma coleção de chaves e índices é mais econômico do que ordenar uma coleção de registros inteiros. Quanto maior for o tamanho de cada registro em relação à chave, maior será a economia de memória. A grande desvantagem é que o arquivo não é realmente ordenado, de forma que, se outra exibição ordenada se fizer necessária, o processo terá que ser repetido.

A título de ilustração, suponha que o arquivo `Tudor.bin` (v. **Apêndice A**) seja um arquivo tão grande que justifique o uso do método de ordenação externa por indexação. Os registros desse arquivo podem ser indexados de `0` até o número de registros menos um, como se vê na **Tabela 12–3**.

Índice	Registro			
0	Henrique VIII	1029	9.5	9.0
1	Catarina Aragon	1014	5.5	6.5
2	Ana Bolena	1012	7.8	8.0
3	Joana Seymour	1017	7.7	8.7
4	Ana de Cleves	1022	4.5	6.0
5	Catarina Howard	1340	6.0	7.7
6	Catarina Parr	1440	4.0	6.0

Tabela 12–3: Registros Indexados num Arquivo

Suponha que a chave de ordenação escolhida seja o campo `nome` (primeiro campo de cada registro). Então o primeiro passo da ordenação por indexação consiste em associar cada chave ao seu respectivo índice, conforme ilustrado na **Tabela 12–3**.

Índice	Chave
0	Henrique VIII
1	Catarina Aragon
2	Ana Bolena
3	Joana Seymour
4	Ana de Cleves
5	Catarina Howard
6	Catarina Parr

Tabela 12–4: Ordenação de Arquivos por Indexação: Pares Desordenados

Após serem ordenados pela chave, esses pares de chaves e índices apresentar-se-iam na ordem vista na **Tabela 12–5**.

Índice	Chave
2	Ana Bolena
4	Ana de Cleves
1	Catarina Aragon
5	Catarina Howard
6	Catarina Parr
0	Henrique VIII
3	Joana Seymour

Tabela 12–5: Ordenação de Arquivos por Indexação: Pares Ordenados

Com esses pares ordenados, sabe-se que o primeiro registro a ser exibido é o de índice *2* (i.e., aquele cujo nome é `Ana Bolena`), depois o de índice *4* (`Ana de Cleves`), depois o de índice *1* (`Catarina Aragon`) e assim por diante.

Problema: Supondo que o arquivo `Tudor.bin` seja tão grande que justifique o uso do método de ordenação externa por indexação, crie um programa que implemente esse método e teste seu funcionamento com o arquivo mencionado.

Solução: Para armazenamento de cada par chave/índice, será utilizada uma estrutura do tipo `tParChaveIndice` definido como:

```
typedef  struct {
            int  indice;
            char chave[MAX_NOME + 1];
         } tParChaveIndice;
```

O primeiro passo na ordenação por indexação é a leitura sequencial das chaves, com o subsequente emparelhamento de cada chave com seu índice. Essa missão é cumprida pela função `ColetaPares()` que lê cada registro no arquivo e armazena seu índice e seu campo `nome` (i.e., a chave de ordenação) num elemento do tipo `tParChaveIndice` de um array. Os parâmetros dessa função são:

- `stream` (entrada) — stream no qual será feita a leitura
- `pares` (saída) — array que armazenará os pares chave/índice
- `maxPares` (entrada) — número de elementos do array

A função `ColetaPares()` retorna o número de registros lidos, se não houver erro, ou um valor negativo em caso contrário. O tipo `tAluno` usado por essa função é definido no **Apêndice A**.

```
int ColetaPares(FILE *stream, tParChaveIndice pares[], int maxPares)
{
   int    i;
   tAluno registro;

   rewind(stream); /* Move o indicador de posição para o início do arquivo */

      /* Lê cada registro do arquivo e armazena sua chave e seu índice no arquivo */
   for (i = 0; i < maxPares; i++) {
         /* Lê um registro */
      fread(&registro, sizeof(registro), 1, stream);

         /* Se o final de arquivo for atingido ou ocorrer */
         /* algum erro de leitura, encerra o laço         */
      if ( feof(stream) || ferror(stream) )
         break;
```

```
         /* Armazena o par chave/índice do registro */
         /* lido num elemento do array pares[]      */
      strcpy(pares[i].chave, registro.nome);
      pares[i].indice = i;
   }

   return ferror(stream) ? -1 : i;
}
```

A próxima tarefa é ordenar a lista de estruturas do tipo `tPares`. Essa tarefa pode ser efetivada com uma única chamada da função **qsort()** da biblioteca padrão de C. Depois disso, resta apenas exibir os registros do arquivo em ordem. Essa é a tarefa da função `ExibeRegistrosOrdenados()` definida abaixo e cujos parâmetros são:

- `stream` (entrada) — stream binário no qual será feita a leitura de registros
- `pares[]` (entrada) — array que armazena os pares chave/índice ordenados pela chave
- `nElementos` (entrada) — número de elementos do array

A função `ExibeRegistrosOrdenados()` retorna zero, se não houver erro, ou um valor diferente de zero em caso contrário.

```
int ExibeRegistrosOrdenados(FILE *stream, const tParChaveIndice pares[], int nElementos)
{
   tAluno registro;
   int    i;

      /* Verifica se há elementos a serem escritos */
   if (!nElementos)
      return 1; /* Não há */

      /* Escreve o cabeçalho da tabela */
   printf( "\n\tNome\t\tMatricula\tNota 1\tNota 2"
           "\n\t====\t\t=========\t======\t======\n\n" );

      /* Acessa cada registro do arquivo na ordem   */
      /* indicada pela sua posição no array pares[] */
   for (i = 0;  i < nElementos; i++) {
         /* Move o indicador de posição do arquivo para  */
         /* o local calculado utilizando pares[i].indice */
      MoveApontador(stream, sizeof(tAluno)*pares[i].indice, SEEK_SET);

      fread(&registro, sizeof(tAluno), 1, stream); /* Lê o registro */

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

         /* Escreve o registro lido com algum embelezamento */
      printf( "    %-15s\t%6s\t\t %3.1f\t %3.1f\n", registro.nome,
              registro.matr, registro.n1, registro.n2 );
   }

      /* Se não ocorreu erro de leitura, o retorno é 0; */
      /* caso contrário, o retorno é diferente de 0     */
   return ferror(stream);
}
```

A função **main()** que completa o programa é relativamente fácil de implementar e o leitor a encontrará no site dedicado ao livro na internet. Na chamada de **qsort()** para ordenação dos pares pode-se usar a função `ComparaStr()` definida como:

```
int ComparaStr(const void *p1, const void *p2)
{
   return strcmp(((tParChaveIndice *)p1)->chave, ((tParChaveIndice *)p2)->chave);
}
```

12.7.2 Testando Ordenação em Memória Secundária

Problema: Escreva uma função que verifica se um arquivo está ordenado em ordem crescente. Suponha que a chave de ordenação seja um campo de um tipo, denominado `tRegistro`, que representa cada registro do arquivo e que a chave de ordenação desses registros é um string denominado `chave`.

Solução: A função `TestaOrdenacaoMS()`, apresentada a seguir, recebe um stream associado ao arquivo que se deseja testar e retorna `1`, se o arquivo estiver ordenado ou `0`, em caso contrário. Note que o stream recebido como parâmetro por essa função deve estar aberto em formato binário que permita leitura.

```
int TestaOrdenacaoMS(FILE *stream)
{
   tRegistro registroAtual, registroAnterior;
   int       teste;

   rewind(stream);

      /* Lê o primeiro registro do arquivo */
   fread(&registroAnterior, sizeof(registroAnterior), 1, stream);
   ASSEGURA( !ferror(stream), "Erro de leitura" );

      /* Lê os demais registros e testa se eles estão ordenados */
   while (1) {
         /* Lê o próximo registro */
      fread(&registroAtual, sizeof(registroAtual), 1, stream);
      ASSEGURA( !ferror(stream), "Erro de leitura" );

      if (feof(stream))
         break; /* Leitura acabou */

         /* Compara a chave do registro atual */
         /* com a chave do registro anterior  */
      teste = memcmp( registroAtual.chave, registroAnterior.chave, TAM_CHAVE );

         /* Verifica se as duas últimas chaves estão em ordem crescente */
      if (teste < 0)
         return 0;

         /* A chave anterior passa a ser a chave corrente */
      memcpy( &registroAnterior, &registroAtual, sizeof(registroAtual) );
   }

      /* Se ainda não houve retorno, o arquivo está ordenado */
   return 1;
}
```

12.8 Exercícios de Revisão

Conceitos Básicos (Seção 12.1)

1. O que é ordenação externa?
2. Por que QuickSort não é adequado para ordenação externa?
3. Como se avalia o desempenho de um algoritmo de ordenação externa?
4. O que é uma passagem de intercalação?

Intercalação Binária (Seção 12.2)

5. (a) Como são denominadas as duas fases tipicamente usadas em ordenação externa? (b) Descreva cada uma delas.

6. Descreva o algoritmo de intercalação binária em memória secundária.

7. Quais são as diferenças entre intercalação binária em memória secundária e o método de ordenação **MergeSort** estudado no **Capítulo 11**?

8. Qual é a diferença básica entre avaliar um algoritmo de ordenação em memória principal, como aqueles vistos no **Capítulo 11**, e avaliar um algoritmo de ordenação em memória secundária?

9. Qual é o custo de uma ordenação externa por intercalação binária?

10. Suponha que se tenha um arquivo ocupa 30.000 blocos em disco e que o tamanho da memória principal disponível corresponda a 6 desses blocos. Se o algoritmo de ordenação por intercalação binária for usado para ordenar esse arquivo, apresente o número de séries que serão produzidas na segunda passagem de intercalação. (b) Quantas passagens de intercalação serão necessárias para ordenar esse arquivo completamente? (c) Quantas operações de entrada ou saída serão realizadas?

Intercalação Multíplice de Arrays (Seção 12.3)

11. (a) Descreva o problema de intercalação multíplice de arrays. (b) Descreva as duas técnicas para resolução desse problema que foram discutidas na **Seção 12.3**.

12. Qual é o papel desempenhado pelo heap na solução do problema de intercalação multíplice de arrays?

13. O que intercalação multíplice de arrays tem a ver com ordenação externa?

14. Qual é o custo do algoritmo de intercalação multíplice de arrays apresentado na **Seção 12.3**?

15. Suponha que os k arrays do problema discutido na **Seção 12.3** não estejam ordenados. (a) Qual é o melhor custo que se pode obter para que seus dados resultem num outro array ordenado? (b) Usar o algoritmo da referida seção é compensador?

Intercalação Multíplice de Arquivos (Seção 12.4)

16. O que é intercalação multíplice?

17. O que é intercalação de vias múltiplas?

18. Descreva o algoritmo de intercalação multíplice.

19. Que vantagens intercalação multíplice apresenta com respeito a intercalação binária?

20. O que é uma série no algoritmo de intercalação multíplice?

21. Por que a escolha do método de ordenação utilizado para ordenar séries no algoritmo de intercalação multíplice não é tão importante?

22. Descreva os seguintes conceitos relacionados a intercalação multíplice:

(a) Buffer de entrada
(b) Buffer de saída
(c) Descarga de buffer
(d) Recarga de buffer

23. (a) O que é um elemento ativo num buffer de entrada? (b) O que é um elemento ativo num buffer de saída?

24. O que ocorre quando o indicador de posição de um buffer de entrada ultrapassa seu limite?

25. O que ocorre quando o indicador de posição de um buffer de saída ultrapassa seu limite?

26. Por que nem sempre é possível efetuar intercalação numa única passagem quando se usa intercalação multíplice?

27. Apresente diagramas semelhantes àqueles da **Seção 12.4.2** que ilustrem o complemento do exemplo apresentado na referida seção.

28. (a) Por que o algoritmo de ordenação multíplice não é capaz de ordenar arquivos arbitrariamente grandes? (b) Como se determina o tamanho do maior arquivo que pode ser ordenado por esse método?

29. Suponha que um computador possua 512 MiB de memória principal disponível, o tamanho de bloco de disco seja igual a 8 KiB e que o tamanho de cada registro seja 256 KiB. Determine o número máximo de registros que podem ser ordenados usando ordenação multíplice.

30. Determine o tamanho máximo de um arquivo que pode ser ordenado pelo algoritmo apresentado na **Seção 12.4.1** considerando um computador com 1 GiB de memória principal disponível (i.e., $M = 2^{30}$ bytes) e um tamanho de bloco de memória secundária igual a 4 KiB (i.e., $B = 2^{12}$ bytes).

31. Suponha que se tenham *10.000.000* registros, cada um dos quais com *200* bytes, constituindo um arquivo de *2* GB. Esse arquivo é armazenado em disco com blocos de *20* KB, cada um dos quais contendo 100 registros. O arquivo inteiro ocupa *100.000* blocos (número de bytes/tamanho de bloco = $200 \times 10^7 / 20 \times 10^3$). Supondo ainda que haja *100* MB disponíveis em memória principal e que o tempo de transferência médio por bloco seja *11,5 ms*, quanto tempo levará a ordenação?

32. Qual é o número máximo de registros se podem ordenar em dois passos na questão **31**?

33. Suponha que se tenham os seguintes parâmetros numa operação de ordenação externa: $M = 100$ MB $= 100.000.000$ bytes $= 10^8$ bytes, $B = 20.000$ bytes, $R = 100$ bytes (número de registros). Qual é o número máximo de registros que podem ser ordenados em dois passos?

34. Quantas séries serão necessárias na ordenação da questão **33**?

35. Suponha que se tenha uma memória disponível $M = 100\ MB = 10^8$ bytes e blocos com tamanho igual a $20\ KB = 2 \times 10^4$ bytes. Qual é o tamanho máximo de arquivo que se pode ordenar usando três passos?

Limite Inferior para Ordenação em Memória Secundária (Seção 12.5)

36. Qual é o menor custo que se pode obter com um algoritmo de ordenação em memória secundária?

Inserção Massiva em Árvores B+ (Bulkloading) (Seção 12.6)

37. (a) O que é inserção massiva? (b) Quando inserção massiva deve ser utilizada?

38. (a) Quais são as estruturas de dados propícias para inserção massiva? (b) Por que essas estruturas favorecem inserção massiva?

39. Descreva o algoritmo de inserção massiva.

40. Qual é o percentual de preenchimento de cada folha de uma árvore B+ criada usando *bulkloading*?

41. (a) Qual é o percentual de preenchimento de cada nó interno de uma árvore B+ criada usando *bulkloading*? (b) Como esse percentual poderia ser aumentado?

42. Apresente um algoritmo de inserção massiva (*bulkloading*) para árvores B.

43. (a) Qual é o percentual de preenchimento de cada folha de uma árvore B criada usando o algoritmo apresentado como resposta da questão **42**? (b) Qual é o percentual de preenchimento de cada nó interno de uma árvore B criada usando esse mesmo algoritmo?

44. Apresente uma representação gráfica que represente uma árvore B+ com grau (G) igual a *5* construída por intermédio de inserção massiva das chaves *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*, *11*, *12*, *13*, *14*, *15* e *16*. Considere o número máximo de chaves numa folha (F) igual a 5.

45. Realize a mesma tarefa da questão **44** considerando agora que as chaves são inseridas uma a uma na ordem em que se encontram.

46. Qual é o número máximo de nós mantidos em memória principal durante uma operação de inserção massiva numa árvore B+?

47. (a) Por que usando a técnica descrita na **Seção 12.6** a maioria dos nós internos é preenchida apenas pela metade? (b) Qual é a consequência desse fato?

48. Desenhe a árvore B+ mais alta possível que se pode obter com as chaves da questão **44**.

49. Como seria a árvore B+ do exemplo discutido na **Seção 12.6.3** se os registros fossem inseridos um a um na ordem em que eles se encontram?

Exemplos de Programação (Seção 12.7)

50. (a) Descreva o método de ordenação por indexação. (b) Quais são as vantagens e desvantagens desse método de ordenação?

51. Avalie o custo temporal do algoritmo de ordenação por indexação.

12.9 Exercícios de Programação

EP12.1 Reimplemente a função `IntercalaBinMS()` apresentada na **Seção 12.2.2** de tal maneira que o arquivo ordenado seja aquele cujo nome é representado pelo segundo parâmetro da função. Nessa nova implementação, a função não retorna nada (pois não é mais necessário).

EP12.2 Escreva uma função que lê em dois arquivos contendo strings em ordem alfabética e intercala-os criando um terceiro arquivo contendo todos os strings dos dois arquivos originais em ordem alfabética.

EP12.3 Implemente funções para inserção massiva de chaves em árvores B semelhantes àquelas apresentadas na **Seção 12.6**.