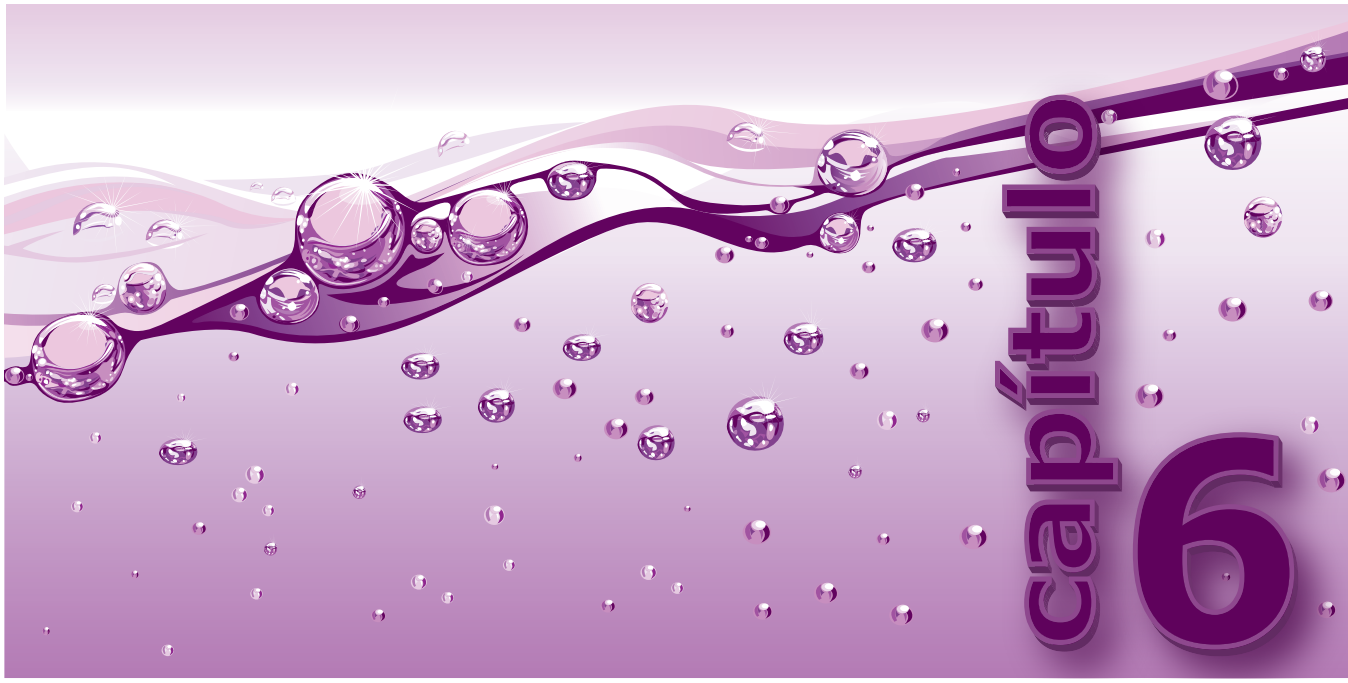


Busca Hierárquica em Memória Secundária

Após estudar este capítulo, você deverá ser capaz de:

objetivos

- Definir os seguintes conceitos relativos a árvores multidirecionais de busca:
  - ❒ Árvore descendente
  - ❒ Árvore ascendente
  - ❒ Ordem
  - ❒ Grau
  - ❒ Filho de chave
  - ❒ Folha
  - ❒ Semifolha
  - ❒ Nó completo
  - ❒ Árvore balanceada
  - ❒ Conjunto de índices
  - ❒ Conjunto sequencial
  - ❒ Preenchimento de estrutura
- Descrever o algoritmo de busca para árvores multidirecionais
- Explicar por que árvores multidirecionais descendentes de busca não são usadas na prática
- Definir transferência de disco e custo de transferência
- Dimensionar o grau de uma árvore multidirecional de busca implementada em arquivo
- Descrever os mecanismos de divisão e junção de nós de árvores B e B+
- Implementar árvore B em arquivo
- Comparar árvores B, B+ e B*
- Mostrar onde se encontram a maior e a menor chaves de uma árvore multidirecional de busca
- Calcular a altura de uma árvore multidirecional de busca
- Implementar busca de intervalo em árvores B e B+

EXISTEM MUITOS PROGRAMAS que precisam lidar com enormes quantidades de dados que não cabem inteiramente em memória principal. Um exemplo de tal situação seria um banco de dados de correntistas de uma grande instituição financeira. Às vezes, essa quantidade de dados é tão grande que mesmo uma tabela de busca que contenha apenas as chaves dos registros precisa ser mantida em memória secundária.

Este capítulo lida com algoritmos e estruturas de dados para busca e atualização de tabelas de busca armazenadas em memória secundária. Mais precisamente, este capítulo ensina técnicas que permitem encontrar um registro num arquivo com vários gigabytes armazenado em disco usando duas ou três operações de leitura. Esse é um tópico de enorme importância prática e para seu bom acompanhamento recomenda-se ao leitor uma releitura do **Capítulo 1**, notadamente as seções que tratam de hierarquias de memória e *caching*.

A medida de complexidade de algoritmos que operam sobre dados armazenados em memória secundária é o número de acessos a blocos, de modo que é melhor acessar uma grande quantidade de dados de uma vez do que acessar a memória secundária várias vezes para obter a mesma quantidade de dados. Logo os dados devem ser organizados de modo a minimizar o número de acessos. Resumindo, os algoritmos que operam sobre dados armazenados em arquivo (memória secundária) são avaliados de acordo com o número de vezes em que blocos são acessados (i.e., lidos ou escritos).

Transferência de disco é a leitura de um bloco em memória secundária e seu subsequente armazenamento em memória principal, ou vice-versa. O desempenho de algoritmos que lidam com tabelas de busca armazenadas em memória secundária é medido em termos do número de transferências de disco necessárias para efetuar uma busca ou atualização da tabela. Essa medida é denominada **custo de entrada e saída** ou **custo de transferência** do algoritmo em questão.

Este capítulo começa apresentando árvores multidirecionais descendentes e pode ser um tanto frustrante saber que elas não possuem nenhuma utilidade prática, pois são deveras ineficientes. Pior ainda, árvores multidirecionais não devem ser implementadas em memória principal. Então por que essas árvores são apresentadas aqui? A resposta a essa última questão é de natureza didática. Árvores multidirecionais descendentes são bem mais fáceis de implementar do que árvores multidirecionais ascendentes que serão discutidas mais adiante neste capítulo. Além disso, como o leitor já deve estar familiarizado com a implementação de árvores em memória principal, implementar inicialmente árvores multidirecionais em memória principal deve facilitar sua implementação em memória secundária.

Este capítulo é melhor apreciado utilizando-se programas-clientes com arquivos de registros realmente grandes, como o arquivo `CensoMEC.bin` descrito no **Apêndice A**, como fazem os programas que usam árvores multidirecionais encontrados no site deste livro na internet.

6.1 Árvores Multidirecionais Descendentes de Busca

6.1.1 Conceitos

Uma **árvore multidirecional de busca de ordem** (ou **grau) n** é uma árvore na qual cada nó possui n ou menos **filhos** e um número de chaves armazenadas igual ao número de filhos menos um (p. ex., se um nó tiver *3* filhos, ele terá *2* chaves). Se os m filhos de um nó forem representados por $f_0, f_1, ..., f_{m-1}$ e suas **chaves** forem representadas por $c_0, c_1, ..., c_{m-2}$, em ordem crescente, então todas as chaves em f_0 serão menores do que ou iguais a c_0, todas as chaves em f_j, sendo $1 \leq j \leq m-2$, serão maiores do que c_{j-1} e menores do que ou iguais a c_j e todas as chaves em f_{m-1} serão maiores do que c_{m-2}. O filho f_j é chamado **filho esquerdo da chave** c_j e **filho direito da chave** c_{j-1}. Um nó pode ter um ou mais filhos vazios.

A **Figura 6–1** ilustra uma árvore multidirecional de busca de ordem (grau) *4*:

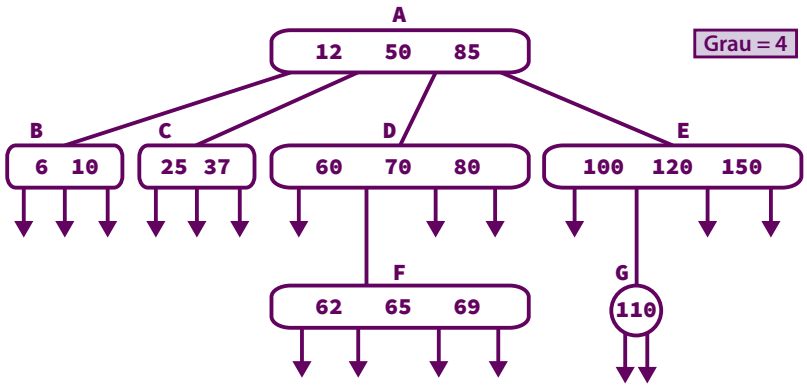


Figura 6–1: Árvore Multidirecional Descendente de Busca de Ordem 4

Na **Figura 6–1**, os nós rotulados com *A*, *D*, *E* e *F* possuem o número máximo de filhos (i.e., *4*) e chaves (i.e., *3*) e, assim, são chamados **nós completos**. As setas representam filhos vazios dos respectivos nós. Nessa mesma figura, os nós *B*, *C* e *G* são **incompletos**. Note ainda que as chaves do nó *B* são todas menores do que a primeira chave de *A* e as chaves do nó *E* são todas maiores do que a última chave de *A*. Finalmente, note que os valores das chaves de *C* estão entre a primeira e a segunda chave de *A*, as chaves de *D* se encontram entre os valores da segunda e da terceira chaves de *A*, e assim por diante.

Uma **folha** de uma árvore multidirecional é um nó cujos filhos são todos nulos. Numa árvore multidirecional de busca **descendente**, qualquer nó incompleto é uma folha. Uma **semifolha** é um nó com pelo menos um filho vazio, de modo que toda folha é também uma semifolha. Por exemplo, na árvore de grau 3 da **Figura 6–2**, os nós de *B* a *G* e de *I* a *R* são semifolhas. De fato, nessa figura, os únicos nós que não são semifolhas são *A* e *H*.

Numa árvore multidirecional de busca descendente, cada semifolha é completa ou é folha, o que significa que um nó só terá seu primeiro filho não vazio após possuir o número máximo de chaves. Por exemplo, na árvore da **Figura 6–2**, o nó *G* só poderá ter seu primeiro filho não vazio após possuir duas chaves.

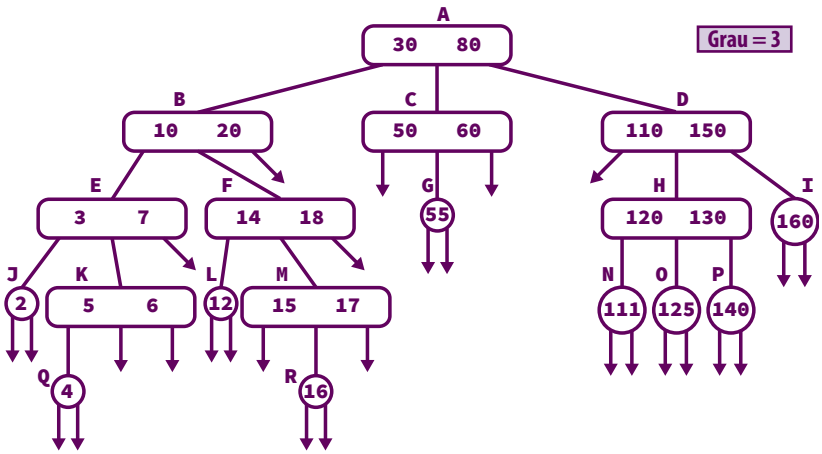


Figura 6–2: Árvore Multidirecional Descendente de Busca de Ordem 3

Numa **árvore multidirecional de busca balanceada**, todas as semifolhas se encontram no mesmo nível, o que implica no fato de todas as semifolhas serem folhas. Árvores B, que serão discutidas na **Seção 6.4**, constituem exemplos de árvores multidirecionais balanceadas.

A **Figura 6–3** mostra uma árvore multidirecional de busca balanceada de ordem *3*. Note que essa árvore não é *descendente*, pois, por exemplo, sua raiz não é completa e, mesmo assim, possui dois filhos que não são vazios.

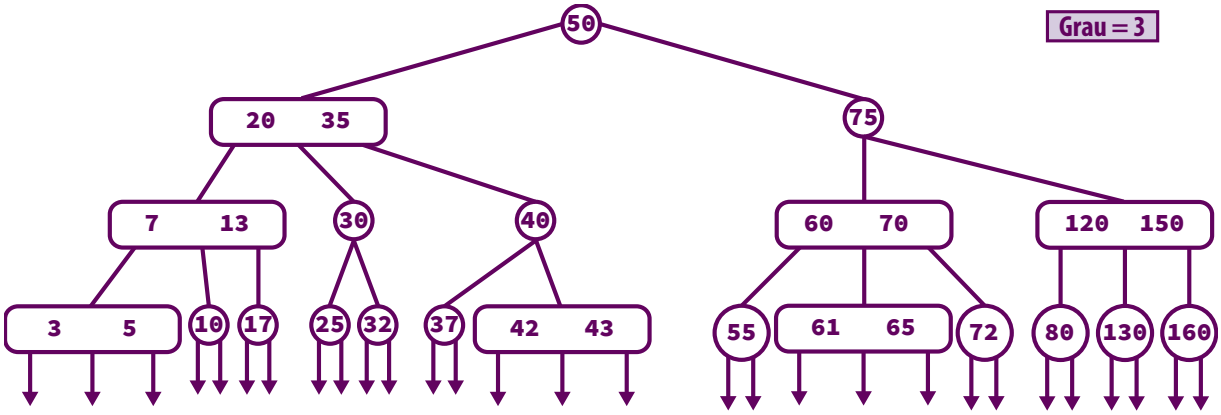


Figura 6–3: Árvore Multidirecional Balanceada de Busca de Ordem 3

6.1.2 Busca

Busca em árvores multidirecionais é uma generalização de busca em árvores binárias. Quer dizer, essa operação começa sempre na raiz da árvore até que se encontre a menor chave que é maior do que ou igual à chave de busca. Se a chave de busca for igual à chave encontrada, a busca estará encerrada. Caso contrário, desce-se na árvore seguindo o filho esquerdo da chave encontrada.

O algoritmo de busca para árvores multidirecionais é apresentado na **Figura 6–4**.

Algoritmo BuscaEmÁrvoreMultidirecional

Entrada: Uma árvore multidirecional de busca e uma chave de busca

Saída: O valor associado à chave que casa com a chave de busca ou um valor informando que a chave não foi encontrada

1. Faça um ponteiro *p* apontar para a raiz da árvore
2. Obtenha o índice *i* da menor chave que é maior do que ou igual à chave de busca no nó apontado por *p*; se tal chave não for encontrada, torne *i* igual ao número de chaves desse nó
3. Se a chave na posição *i* do nó apontado por *p* for igual à chave de busca, retorne o valor associado a essa chave
4. Caso contrário, se o filho na posição *i* desse nó for nulo, retorne um valor que indique o fracasso da operação
5. Caso contrário, faça com que *p* aponte para esse filho e volte para o **Passo 2**

Figura 6–4: Algoritmo de Busca em Árvore Multidirecional de Busca

A **Figura 6–5** apresenta exemplos de aplicação do algoritmo de busca descrito na **Figura 6–4** numa árvore multidirecional de busca de ordem *4*.

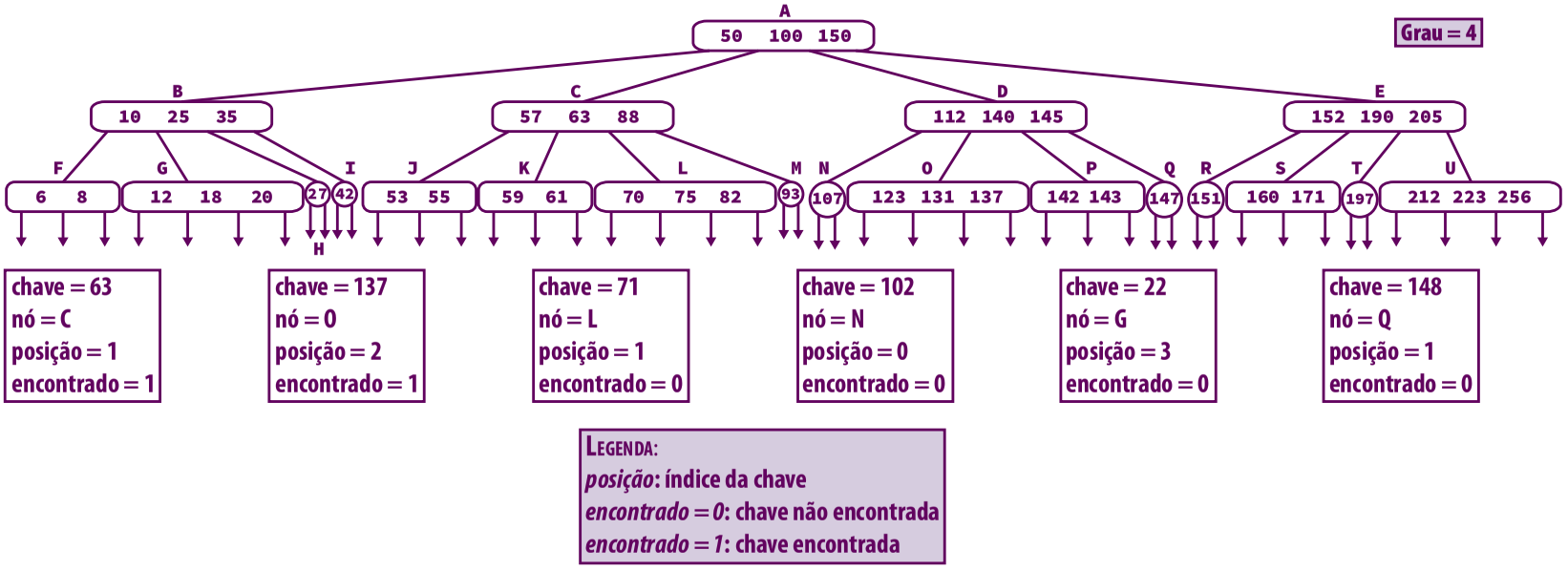


FIGURA 6–5: EXEMPLOS DE BUSCA EM ÁRVORE MULTIDIRECIONAL DESCENDENTE DE BUSCA

6.1.3 Inserção

No algoritmo de inserção apresentado na **Figura 6–6**, assume-se que chaves duplicadas não são permitidas na árvore (i.e., as chaves são consideradas primárias).

ALGORITMO INSEREEMÁRVOREMULTIDIRECIONALDESCENDENTE

ENTRADA: Uma nova chave e seu valor associado

ENTRADA/SAÍDA: Uma árvore multidirecional descendente de busca

1. Se a árvore estiver vazia, crie um nó, acrescente a chave a esse nó, torne-o raiz da árvore e retorne informando o sucesso da operação
2. Encontre o nó que contém ou deveria conter a chave a ser inserida
3. Se a chave foi encontrada, retorne um valor informando o fracasso da operação (pois a chave é considerada primária)
4. Se a chave não foi encontrada, o nó encontrado no **Passo 2** é aquele que acomodará a nova chave. Então faça o seguinte:
   - **4.1** Se o nó não estiver completo, insira a nova chave no nó de modo que suas chaves permaneçam ordenadas
   - **4.2** Caso contrário, crie um nó contendo a nova chave e torne esse nó filho esquerdo da menor chave do nó encontrado no **Passo 2** que é maior do que a nova chave
5. Retorne um valor indicando que a operação foi bem-sucedida

FIGURA 6–6: ALGORITMO DE INSERÇÃO EM ÁRVORE MULTIDIRECIONAL DESCENDENTE DE BUSCA

A árvore construída de acordo com o algoritmo de inserção da **Figura 6–6** é descendente, pois um novo nó só é criado quando seu pai está completo. Assim nenhum nó incompleto possui filho não vazio e é, portanto, uma folha.

A **Figura 6–7** mostra a inserção da chave *102* na folha *N* e *148* na folha *Q* do exemplo ilustrado na **Figura 6–5**. Em ambos os casos, o nó no qual ocorre a inserção é incompleto.

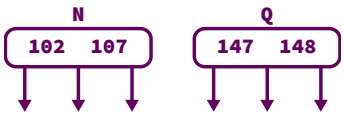


FIGURA 6–7: INSERÇÃO EM FOLHA DE ÁRVORE MULTIDIRECIONAL DESCENDENTE DE BUSCA

A **Figura 6–8** apresenta o resultado obtido com a inserção da chave *71* no nó *L* e da chave *22* no nó *G* da árvore ilustrada na **Figura 6–5**. Nesses casos, cada nó no qual ocorre inserção é completo, de modo que é necessário criar um novo nó (folha) para conter a chave inserida.

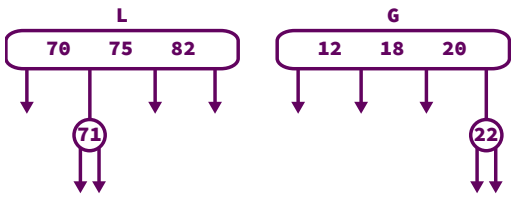


FIGURA 6–8: INSERÇÃO EM ÁRVORE MULTIDIRECIONAL DESCENDENTE DE BUSCA 1

Exercício: Para certificar-se que realmente entendeu o processo de inserção em árvores multidirecionais descendentes de busca, desenhe o estado intermediário da árvore da **Figura 6–8** após a inserção de cada chave.

A **Figura 6–9** ilustra as inserções no nó *L* das chaves com valores *86*, *77*, *87*, *84*, *85* e *73*, nessa ordem. Nessa figura, o estado inicial do nó *L* antes dessas inserções é aquele da **Figura 6–8**.

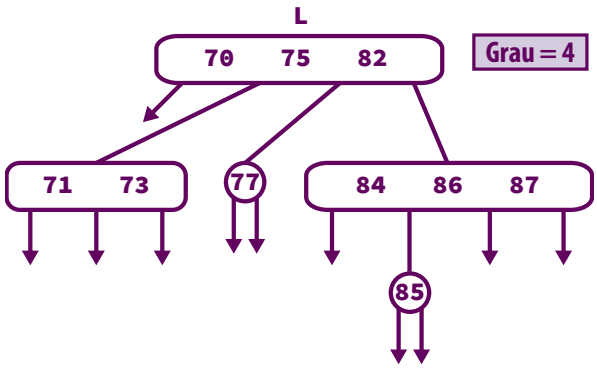


Figura 6–9: Inserção em Árvore Multidirecional Descendente de Busca 2

6.1.4 Remoção

O algoritmo de remoção de árvores multidirecionais descendentes é uma generalização daquele apresentado para árvores de busca binárias e segue os passos descritos na **Figura 6–10**.

Algoritmo RemoveEmÁrvoreMultidirecionalDescendente

Entrada: A chave a ser removida

Entrada/Saída: Uma árvore multidirecional de busca

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

1. Tente encontrar a chave a ser removida usando o algoritmo de busca apresentado na **Figura 6–4**
2. Se a chave não for encontrada, encerre anunciando o fracasso da operação
3. Se a chave a ser removida tiver um filho esquerdo ou direito vazio:
   - **3.1** Remova a chave
   - **3.2** Se houver outras chaves no nó, compacte-o (v. adiante)
   - **3.3** Se a chave removida era a única chave do nó
     - **3.3.1** Libere o nó
     - **3.3.2** Atualize o pai desse nó
4. Se a chave a ser removida tiver filhos esquerdo e direito não vazios
   - **4.1** Encontre a chave sucessora da chave a ser removida (essa chave sucessora deve ter uma subárvore esquerda vazia — v. **Seção 4.1.2**)
   - **4.2** Substitua a chave a ser removida pela chave sucessora
   - **4.3** Remova a chave sucessora usando o **Passo 3** acima

Figura 6–10: Algoritmo de Remoção em Árvore Multidirecional Descendente de Busca

A compactação de nós a que se refere o **Passo 3.2** do algoritmo acima consiste em mover todas as chaves a partir daquela que imediatamente segue a chave removida para uma posição anterior, como ilustra a **Figura 6–11**. Observe nessa figura que alguns filhos dos nós também são movidos para trás, mas de modo diferente dependendo de qual dos filhos da chave removida está vazio. Quer dizer, se o filho esquerdo dessa chave for vazio, o primeiro filho a ser movido é filho direito da chave removida [v. **Figura 6–11 (a)**]. Por outro lado, se o filho esquerdo da chave removida não for vazio, o primeiro filho a ser movido é o filho direito da próxima chave à direita da chave removida [v. **Figura 6–11 (b)**].

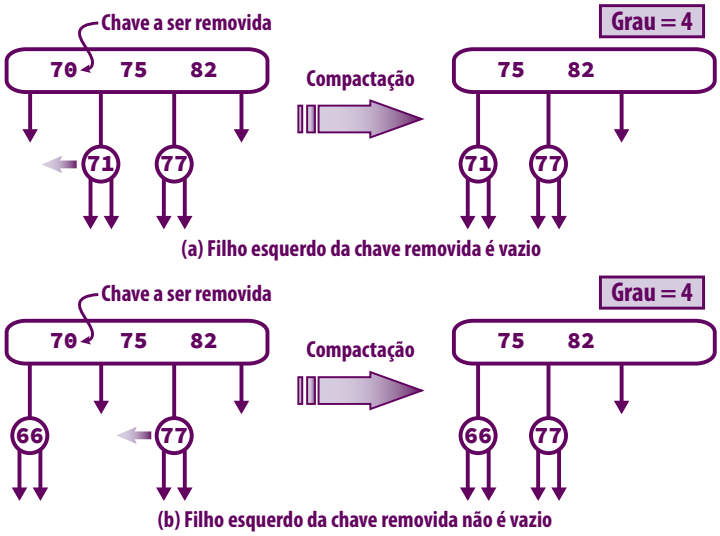


Figura 6–11: Exemplos de Compactação de Nó após Remoção

Seguindo o algoritmo descrito acima, uma árvore multidirecional de busca pode deixar de ser considerada como tal de acordo com a definição apresentada na **Seção 6.1.1**. Isto é, após uma operação de remoção, pode-se obter uma árvore com um nó incompleto que não é folha. Esse problema pode ser resolvido fazendo-se algumas alterações no algoritmo de remoção descrito anteriormente. Mas não compensa lidar com essa complicação adicional, visto que árvores multidirecionais descendentes não têm muita utilidade prática, o que não ocorre com as árvores B e B+, que serão estudadas adiante.

6.1.5 Implementação em Memória Principal

Conforme foi afirmado, árvores multidirecionais de busca são dirigidas para a implementação de tabelas de busca residentes em memória secundária. Todavia, por razões meramente didáticas, será parcialmente mostrado a seguir como essas árvores podem ser implementadas em memória principal. A implementação completa dessas árvores em memória principal encontra-se no site dedicado ao livro na internet (v. **Prefácio**).

O tipo de nó de uma árvore multidirecional implementada em memória principal é definido utilizando uma estrutura com três campos, conforme se vê abaixo.

```
typedef struct rotNoMulti {
          int               nFilhos; /* O número de filhos do nó */
          tChaveIndice      chaves[G - 1]; /* array contendo chaves  */
                                           /* e índices de registros */
          struct rotNoMulti *filhos[G]; /* array contendo as posições dos filhos */
                                        /* no arquivo que contém a árvore       */
       } tNoMulti, *tArvoreMulti;
```

Nessa definição de tipo, tem-se que:

- `G` é uma constante que representa o grau da árvore
- `nFilhos` é um campo inteiro que representa o número de filhos de um nó
- `filhos[]` é um array, cujo número de elementos é igual a `G`, contendo ponteiros para os filhos do nó
- `chaves[]` é um array com `G – 1` elementos contendo pares do tipo definido como:

```
typedef struct {
          tChave chave;
          int    indice;
       } tChaveIndice;
```

Nessa última definição de tipo, tem-se que:

- `chave` é a chave do registro ao qual a chave pertence
- `indice` é o índice (i.e., a posição) do registro no arquivo que armazena os registros

A **Figura 6–12** ilustra a interpretação de um item de informação do tipo `tChaveIndice` armazenado num nó de uma árvore multidirecional de busca.

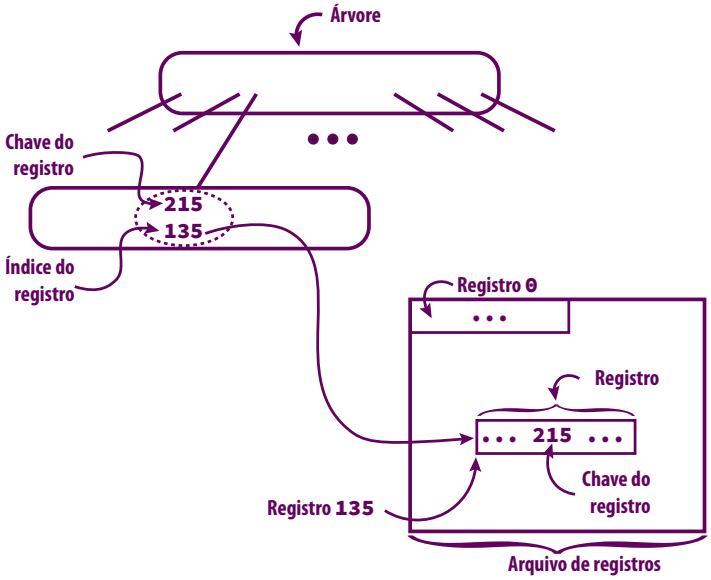


FIGURA 6–12: CONTEÚDO EFETIVO DE UM NÓ DE ÁRVORE MULTIDIRECIONAL DE BUSCA

Aqui, a chave será considerada do tipo **int**, de modo que o tipo `tChave` é definido como:

```
typedef int tChave;
```

6.1.6 Análise

Assim como ocorre com árvores de busca binárias, a ordem na qual as chaves são inseridas afeta o posicionamento das chaves na árvore e, consequentemente, o formato da árvore. Uma inserção pode transformar uma folha em semifolha e, assim, desbalancear a árvore, mas, na prática, essa técnica não produz árvores muito desbalanceadas.

Uma desvantagem do método de inserção em árvores multidirecionais descendentes é que são criadas folhas contendo apenas uma chave e algumas folhas podem ser criadas antes que outras folhas estejam completas. Por isso esse método pode causar grande desperdício de memória.

A vantagem de árvores de busca multidirecionais descendentes é que seus nós superiores são completos, de modo que um grande número de chaves é encontrado em caminhos curtos. No entanto, essas árvores podem tornar-se degeneradas (i.e., muito profundas) quando um grande número de chaves ordenadas são inseridas, assim como ocorre com árvores binárias ordinárias de busca (v. **Capítulo 4**).

6.2 Estruturas de Dados em Memória Secundária

Conforme foi visto nos últimos capítulos referentes a implementações de tabelas de busca em memória principal, o melhor que se pode obter em termos de custo temporal para operações básicas de busca, inserção e remoção é $\theta(\log n)$. Dentre as estruturas vistas até aqui, a única que garante esses custos é a árvore AVL. Mas, apesar de esses custos serem formidáveis em memória principal, eles não são tão bons para uma tabela de busca implementada em memória secundária.

Como uma CPU não lida com disco diretamente, para qualquer dado ser acessado, ele tem que ser primeiro lido do disco para a memória principal. Dados são armazenados em disco em unidades chamadas **blocos** ou **páginas** (v. **Seção 1.1.3**). Cada acesso a disco deve ler ou escrever um ou múltiplos blocos. Isto é, mesmo que se precise acessar um único byte armazenado num bloco de disco que contém milhares de bytes, precisa-se ler o bloco inteiro. Isso mostra por que estruturas de dados usadas em memória interna não podem ser diretamente implementadas como estruturas de indexação de memória secundária.

Árvores binárias, mesmo que completamente balanceadas, não constituem uma boa alternativa, visto que, no pior caso, cada nó acessado para uma busca ou atualização numa dessas estruturas estará num bloco diferente. Assim todas essas operações requerem custos de transferências $\theta(\log n)$ no pior caso para executar uma operação de busca ou atualização. Para entender melhor a gravidade da situação considere o mesmo exemplo apresentado na **Seção 4.4.5**. Naquele exemplo, argumentou-se que uma busca numa árvore AVL que armazena *1.000.000* de chaves requer, no máximo, cerca de *20* acessos a nós, o que é considerado excelente para uma operação em memória principal. Mas, já pensou quantas operações uma CPU moderna poderia executar enquanto são efetuados *20* acessos à memória secundária[1]?

Processar dados que se encontram em memória secundária (i.e., num meio de armazenamento externo) é radicalmente diferente de processar dados que se encontram em memória principal. A principal diferença reside no fato de uma CPU não processar diretamente dados que não se encontrem em memória principal ou acima dela na hierarquia de memória discutida no **Capítulo 1**. Quer dizer, dados que se encontram em memória secundária precisam ser transferidos para a memória principal antes de ser processados.

Dados que residem em memória principal são armazenados em variáveis estáticas e dinâmicas. Variáveis estáticas são *automaticamente*[2] alocadas à medida que o programa precisa delas, e liberadas quando esse não é o caso. Por outro lado, em linguagens algorítmicas convencionais, como C, variáveis dinâmicas são explicitamente alocadas e liberadas pelo programador por meio de chamadas de funções de alocação dinâmica de memória. O conteúdo de uma variável estática pode ser acessado usando-se simplesmente seu nome. Por sua vez, o conteúdo de uma variável dinâmica é acessado por meio de seu endereço e do operador de indireção. Tanto num caso quanto no outro, dados são alterados por meio de operadores com efeito colateral, como, por exemplo, os operadores de incremento e atribuição.

Dados armazenados em memória secundária, como um disco rígido, por exemplo, precisam ser carregados em memória principal antes que possam ser processados. Essa operação é efetuada por meio de uma chamada de função de leitura como, por exemplo, **fread()**. Dados são armazenados ou alterados em memória secundária por meio de funções de escrita, como **fwrite()**, por exemplo.

Dados residentes em memória secundária não podem ser acessados diretamente por nome ou por meio de ponteiros. Ou seja, o acesso se dá pela posição relativa ao início do arquivo e pela chamada de uma função de posicionamento, tal como **fseek()**, por exemplo. Arquivos enormes, como aqueles tipicamente indexados por árvores multidirecionais, precisam usar tipos com largura adequada para suportar valores muito grandes de índices de bytes em tais arquivos (v. **Seção 2.14**).

A **Tabela 6–1** apresenta algumas diferenças entre processamento de dados residentes em memória principal e em memória secundária.

[1] Se o leitor não souber fazer uma estimativa, mesmo que grosseira, para a resposta a essa questão sugere-se que ele estude o **Capítulo 1**.

[2] Aqui, *automaticamente* significa que o compilador gera código que permite alocar e liberar essas variáveis quando o programa é executado.

Memória Principal	Memória Secundária
Encadeamento é obtido por meio de ponteiros que armazenam endereços de nós	Encadeamento se dá armazenando-se posições de nós (i.e., índices de bytes)
Ponteiro nulo (NULL) indica o fim de um caminho	Posição inválida indica o fim de um caminho. Essa posição inválida deve ser representada por um valor inteiro negativo, visto que zero é uma posição válida em arquivo

Tabela 6–1: Estruturas Encadeadas em Memórias Principal e Secundária

Quando uma árvore é mantida em arquivo, um *ponteiro* para um nó é interpretado como a posição no arquivo desse nó. Determinar essa posição é deveras facilitado quando o arquivo que armazena a árvore contém apenas seus nós e eles são do mesmo tamanho. A **Tabela 6–2** mostra as principais diferenças entre efetuar buscas em duas árvores de busca idênticas, sendo que uma delas é armazenada em memória principal e a outra é armazenada em arquivo.

Memória Principal	Memória Secundária
Usa dois ponteiros, frequentemente denominados `p` e `q`, sendo que `q` segue `p` um nível acima	Usa duas variáveis inteiras, frequentemente denominadas `p` e `q`, para armazenar posições de nós no arquivo que armazena a árvore. A variável `q` armazena a posição de um nó que está um nível acima da posição do nó indicada por `p`.
Os nós já se encontram em memória principal	Nós precisam ser localizados e carregados (lidos) em memória principal. Uma função [como `LeNoMultiMS()`, apresentada na **Seção 6.3.3**] é bastante útil para efetuar essa operação.
Nós são tipicamente acessados por meio de ponteiros (i.e., usando-se o operador seta)	Nós são tipicamente acessados por meio de variáveis (estruturas) que os armazenam (i.e., usando-se o operador ponto)

Tabela 6–2: Descidas em Árvores em Memórias Principal e Secundária

Tipicamente, na análise de desempenho de algoritmos que atuam sobre estruturas de dados em memória secundária, utiliza-se o **modelo de memória externa padrão**, proposto por Aggarwal e Vitter em 1988 (v. **Bibliografia**). De acordo com esse modelo, o computador consiste em:

- Um único processador
- Uma memória interna capaz de armazenar M bytes
- Uma memória externa ilimitada

Além disso, esse modelo apresenta os seguintes pressupostos:

- Em cada operação de entrada ou saída o computador transfere um bloco contíguo de B bytes da memória interna para a memória secundária ou vice-versa, sendo que $1 \leq B \leq M$.
- O custo temporal de um algoritmo é definido como o número de operações de entrada ou saída efetuadas durante sua execução. O custo de operações efetuadas em memória interna é considerado nulo.
- O tamanho n dos dados de entrada processados pelo algoritmo é definido como o número de blocos que os constituem.

A notação a ser usada em análise de algoritmos para memória secundária considera letras maiúsculas como números de bytes e letras minúsculas como números de blocos, como mostra a **Tabela 6–3**. Bloco é considerado a unidade de leitura ou escrita em memória secundária.

Notação	Significado
B	Número de bytes por bloco
M	Número de bytes que podem ser mantidos em memória interna (i.e., o tamanho da memória principal disponível)
N	Número de bytes dos dados de entrada (i.e., o tamanho em bytes do arquivo de entrada)
S	Número de bytes dos dados de saída (i.e., o tamanho em bytes do arquivo de saída)
$m = M/B$	Número de blocos que podem ser mantidos em memória interna (i.e., a capacidade da memória principal em termos de número de blocos)
$n = N/B$	Número de blocos nos dados (arquivo) de entrada
$s = S/B$	Número de blocos nos dados (arquivo) de saída

Tabela 6–3: Notação para Análise de Algoritmos em Memória Secundária

6.3 Árvores Multidirecionais de Busca em Memória Secundária

Árvores de busca multidirecionais são bastante usadas na prática quando implementadas em dispositivos de armazenamento externo que permitem acesso direto (p. ex., HD). Como cada acesso a um nó da árvore requer uma operação de leitura no meio de armazenamento externo e essas operações são relativamente lentas, sistemas de armazenamento externo baseados em árvores de busca multidirecionais tentam maximizar o tamanho de cada nó (p. ex., árvores multidirecionais de ordem 200 ou mais são comuns). Quando muitos nós não são completos, árvores de busca multidirecionais podem desperdiçar muito espaço em disco.

Os registros podem ser armazenados juntamente com as chaves (i.e., cada registro é armazenado num nó da árvore) ou separadamente (i.e., cada chave é armazenada num nó juntamente com um ponteiro para o respectivo registro).

Como o tamanho de um nó é afetado pela quantidade de dados lidos de cada vez no meio externo, obtém-se uma árvore de ordem maior mantendo-se os registros fora dos nós (tabela de busca com chave externa). Essa opção compensa mesmo que implique numa leitura adicional para acessar o registro após a localização de sua chave.

Recomenda-se ao leitor que tenha dificuldade de entender a implementação em memória secundária, que será apresentada a seguir, que estude detidamente a implementação em memória principal, que se encontra no site do livro, antes de prosseguir.

6.3.1 Dimensionamento de Grau e Preenchimento de Estruturas

O tipo `tChaveIndice` definido na **Seção 6.1.5** é o tipo do par chave/valor armazenado como *chave* nos nós da árvore. O tipo `tNoMultiMS` definido a seguir representa os nós de uma árvore multidirecional de busca implementada em arquivo.

```
typedef struct {
           int          nFilhos;
           tChaveIndice chaves[G - 1];
           int          filhos[G];
       } tNoMultiMS, *tArvoreMultiMS;
```

Observe que o tipo dos elementos do array `filhos[]`, apesar de representar índices de nós no meio de armazenamento externo, não pode ser um tipo inteiro sem sinal porque a posição inválida (equivalente ao endereço

NULL em memória principal) é indicada por um valor inteiro negativo. Mais precisamente, esse valor é definido pela macro:

```
#define POSICAO_NULA -1
```

A largura do tipo dos elementos do array `filhos[]` também não precisa ser de 64 bits, pois apesar de o arquivo que conterá a árvore ser bem grande, ele não é tão grande ao ponto de justificar o uso dessa largura (v. **Seção 2.14**).

O valor da constante `G`, que determina o grau da árvore, é calculado em função do tamanho do bloco usado pelo sistema de arquivos utilizado e dos tamanhos dos tipos dos campos que constituem cada nó da árvore. Para efetuar esse cálculo, suponha que:

- *TB* é o tamanho de um bloco lido ou escrito num arquivo (v. **Seção 1.5.3**)
- *TCI* é o tamanho de um par chave/índice armazenado num nó; esse valor pode ser calculado usando-se o operador **sizeof** como `sizeof(tChaveIndice)`
- *TI* é a largura do tipo inteiro usado para representar a posição de um filho de um nó e do número de filhos do nó; esse valor pode ser calculado usando-se o operador **sizeof** como `sizeof(int)`
- *G* é o grau da árvore, que se deseja determinar

Assim pode-se calcular o tamanho (*TN*) de um nó da árvore da seguinte maneira:

$$TN = TI + (G - 1) \cdot TCI + G \cdot TI$$

Fatorando-se essa expressão tem-se que:

$$TN = G \cdot (TCI + TI) + TI - TCI$$

Agora, como, idealmente, deve-se ter $TN \leq TB$, obtém-se:

$$G \leq (TB + TCI - TI)/(TCI + TI)$$

Idealmente, deve-se escolher o grau da árvore de acordo com o tamanho de cada bloco lido/escrito no meio de armazenamento externo no qual a árvore se encontra armazenada. Portanto em princípio, bastaria considerar o grau da árvore como sendo:

$$G = (TB + TCI - TI)/(TCI + TI) - 1 \qquad \textbf{Equação 6–1}$$

e o problema estaria resolvido.

Acontece, porém, que o tamanho de uma estrutura em memória principal nem sempre coincide com a soma dos tamanhos dos seus campos, pois compiladores costumam adicionar preenchimentos para permitir alinhamento de campos. Discutir esse tema em profundidade está além do escopo desse livro[3], mas o leitor deve ter ciência desse problema potencial, pois ele poderá detonar todo o dimensionamento de grau exposto acima, fazendo com que um nó ocupe mais de um bloco, requerendo dois acessos ao disco para leitura de cada nó. Assim a detecção e a solução para esse problema serão brevemente discutidas aqui.

Em primeiro lugar, a ocorrência de preenchimento pode ser detectada simplesmente calculando-se a soma dos tamanhos dos campos da estrutura e o tamanho da estrutura inteira. Esses cálculos são efetuados utilizando-se o operador **sizeof** e os tipos dos campos da estrutura, no primeiro caso, e esse mesmo operador em conjunto com o tipo da própria estrutura. Se esses valores forem iguais, não haverá preenchimento; se o tamanho da estrutura for maior do que a soma dos tamanhos dos campos haverá preenchimento; caso contrário, você errou nos cálculos, pois isso é impossível de ocorrer.

Os cálculos descritos acima devem ser efetuados antes de o programa ser considerado *pronto*, pois, caso seja detectada a ocorrência de preenchimento, o programador precisa levá-la em consideração. No caso em questão, a maneira mais simples de lidar com preenchimento de estruturas é impedindo que esse preenchimento seja

[3] Consulte, por exemplo, o livro *Programando em C: Volume 2* (2009) do autor deste livro (v. **Bibliografia**).

escrito em arquivo, o que é obtido escrevendo sempre (literalmente) cada campo da estrutura individualmente. Nesse caso, é importante que cada leitura de estrutura também seja lida campo a campo. Essa abordagem parece ser entediante e sujeita a erros, mas esse não é o caso. Para implementá-la, basta que o programador escreva uma função contendo uma instrução para leitura de cada campo da estrutura e outra função contendo uma instrução de escrita para cada um desses campos. Então em vez de usar uma função da biblioteca padrão de C [p. ex., **fread**()] para leitura de uma estrutura, usa-se a função que lê os campos da estrutura individualmente. O programador deve proceder de modo semelhante com relação à escrita de estrutura em disco.

Outra abordagem para lidar com preenchimento de estruturas consiste em reduzir o tamanho do grau obtido como foi descrito antes, de tal modo que o tamanho do nó mais o tamanho do preenchimento seja o mais próximo possível do tamanho de um bloco no sistema utilizado. Esse resultado é obtido por tentativa e erro, já que, agora, há duas variáveis (o tamanho do bloco e o tamanho do preenchimento) e apenas uma equação. Não é difícil realizar essa segunda abordagem, mas, utilizando-a, o grau da árvore poderá ser menor do que aquele obtido com a primeira abordagem, o que indica uma solução subótima.

As definições de constantes a seguir refletem o que foi discutido acima sem levar em consideração um possível preenchimento de estrutura.

```
#define TB   4096 /* Tamanho do bloco lido/escrito */
#define TCI  sizeof(tChaveIndice) /* Tamanho de um par chave/índice */
   /* Tamanho de um filho e do inteiro que representa o grau do nó */
#define TI   sizeof(int)
#define G ((TB + TCI - TI)/(TCI + TI) - 1) /* Cálculo do grau da árvore */
```

Como exemplo de cálculo de grau de uma árvore multidirecional armazenada em arquivo, considere os seguintes valores típicos:

- Tamanho do bloco lido/escrito (*TB*): *4096* bytes
- Tamanho de cada par chave/índice (*TCI*): *8* bytes
- Tamanho de cada índice (*TI*): *4* bytes
- Tamanho do inteiro que representa o grau do nó (*TG*) = 4 bytes

Considerando-se esses valores e usando a **Equação 6–1** apresentada antes, o grau da árvore, pode ser obtido como:

$$G = (TB + TCI - TI)/(TCI + TI) - 1 = (4096 + 8 - 4)/(8 + 4) - 1 \cong 340$$

6.3.2 Tratamento de Exceções

Lembra a Lei de Murphy para processamento de arquivos descrita na **Seção 2.13**? Essa lei enfatiza o fato de operações de entrada e saída com arquivos requererem atenção especial com relação a condições de exceção.

Aqui, a macro básica utilizada em tratamento de condições de exceção continua sendo `ASSEGURA`, cuja definição é repetida abaixo para facilidade de referência:

```
#define ASSEGURA(condicao, msg) if (!(condicao)) {\
                                   fprintf(stderr, "\n%s\n", msg);\
                                   exit(1); \
                                }
```

As seguintes macros são usadas em conjunto com a macro `ASSEGURA` para exibir mensagens de erro indicando o tipo de erro e em que função do programa o erro ocorreu:

```
#define ERRO_OPEN(f) "ERRO: Impossivel abrir o arquivo em " #f "()"
#define ERRO_SEEK(f) "ERRO: Impossivel mover apontador de arquivo em " #f "()"
```

```
#define ERRO_TELL(f) "ERRO: Impossivel obter apontador de arquivo em " #f "()"
#define ERRO_FWRITE(f) "ERRO: Impossivel escrever em arquivo em " #f "()"
#define ERRO_FREAD(f) "ERRO: Impossivel ler arquivo em " #f "()"
#define ERRO_STREAM_NULL(s, f) "ERRO: Stream " #s "' e' NULL em " #f "()"
#define ERRO_POSICAO(f) "ERRO: Posicao invalida em " #f "()"
```

O parâmetro `f` nas macros acima deve ser substituído pelo nome da função na qual a macro é invocada e `#f` no corpo dessas macros significa que esse parâmetro será convertido em string.

Essas últimas macros são utilizadas em conjunto com a macro `ASSEGURA` e levam em consideração todos os erros básicos que podem ocorrer em processamento de arquivos. O uso dessas macros tem dois objetivos:

1. Uniformizar as mensagens de erro mais comuns.
2. Reduzir o tamanho da intervenção de instruções de tratamento de exceções. Por exemplo, em vez de se escrever:

   ```
   ASSEGURA(pos >= 0, "Erro em LeNoMultiMS(): posicao invalida");
   ```

   pode-se escrever o seguinte, usando a macro `ERRO_POSICAO`:

   ```
   ASSEGURA(pos >= 0, ERRO_POSICAO(LeNoMultiMS));
   ```

6.3.3 Leitura e Escrita de Nós

Uma implementação de árvore multidirecional em arquivo precisa com frequência ler e atualizar nós que se encontram em arquivo. Por consequência ter funções que realizem essas tarefas facilita tal implementação.

A função `LeNoMultiMS()`, apresentada a seguir, faz a leitura de um nó de uma árvore multidirecional armazenada em arquivo. Essa função recebe como parâmetro a posição de um nó dentro de um arquivo, lê o nó nesse arquivo e armazena o conteúdo lido no endereço em memória especificado por seu terceiro parâmetro. Mais precisamente, os parâmetros dessa função são:

- `stream` (entrada) — stream associado ao arquivo que armazena a árvore e no qual será feita a leitura do nó
- `pos` (entrada) — posição no arquivo onde será feita a leitura (i.e., o índice do nó no arquivo)
- `*no` (saída) — ponteiro para o nó que conterá o resultado da leitura

```
static void LeNoMultiMS(FILE *stream, int pos, tNoMultiMS *no)
{
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, LeNoMultiMS));
   ASSEGURA(pos >= 0, ERRO_POSICAO(LeNoMultiMS));

      /* Tenta mover o apontador de arquivo para o local */
      /* de leitura; se não conseguir, aborta o programa */
   MoveApontador(stream, sizeof(tNoMultiMS)*pos, SEEK_SET);

   fread(no, sizeof(tNoMultiMS), 1, stream); /* Tenta ler o nó */

   ASSEGURA(!ferror(stream), ERRO_FREAD(LeNoMultiMS)); /* Houve erro de leitura? */
}
```

A função `EscreveNoMultiMS()`, implementada abaixo, é usada para atualizar o conteúdo de um nó no arquivo que contém a árvore e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que contém a árvore
- `pos` (entrada) — posição do nó no arquivo que contém a árvore
- `*pNo` (entrada) — nó que será atualizado

```
static void EscreveNoMultiMS(FILE *stream, int pos, const tNoMultiMS *pNo)
{
      /* Tenta mover o apontador de posição para o local onde o nó está armazenado */
   MoveApontador(stream, sizeof(*pNo)*pos, SEEK_SET);

    fwrite(pNo, sizeof(*pNo), 1, stream); /* Escreve o nó no arquivo */

      /* Se ocorreu erro, aborta o programa */
   ASSEGURA(!ferror(stream), ERRO_FWRITE(EscreveNoMultiMS));
}
```

A função `MoveApontador()` chamada pelas funções `LeNoMultiMS()` e `EscreveNoMultiMS()` foi definida na **Seção 2.14**.

6.3.4 Busca

A função `BuscaMultiMS()`, apresentada a seguir, faz uma busca numa árvore multidirecional armazenada em arquivo e retorna o índice do registro correspondente à chave de busca, se ela for encontrada, ou a constante `POSICAO_NULA`, se a chave não for encontrada. Os parâmetros dessa função são:

- `stream` (entrada) — stream associado ao arquivo no qual a árvore reside
- `chave` (entrada) — a chave de busca

```
int BuscaMultiMS(FILE *stream, tChave chave)
{
   tNoMultiMS no;   /* Armazenará cada nó lido no arquivo */
   int        posNo = 0, /* Posição do nó no arquivo; a busca    */
                         /* começa na raiz que ocupa a posição 0 */
              i; /* Índice de uma chave num nó */

      /* Desce na árvore até encontrar a chave de busca ou uma posição nula */
   while (posNo != POSICAO_NULA) {
      LeNoMultiMS(stream, posNo, &no); /* Lê o nó na posição indicada */

         /* Tenta encontrar a chave no nó recém-recuperado */
      i = BuscaEmNoMultiMS(chave, &no);

         /* Verifica se a chave foi encontrada. A primeira comparação abaixo é     */
         /* necessária para evitar acesso a uma posição inválida no array chaves[] */
      if (i < no.nFilhos - 1 && chave == no.chaves[i].chave)
         return no.chaves[i].indice; /* A chave foi encontrada */

      posNo = no.filhos[i]; /* Desce mais um nível na árvore */
   }
   return POSICAO_NULA;  /* A chave não foi encontrada */
}
```

A função `BuscaMultiMS()` chama a função `BuscaEmNoMultiMS()` para tentar encontrar dentro de um nó uma chave que seja menor do que ou igual a uma dada chave de busca. Essa última função retorna o índice da chave dentro do nó se for encontrada uma chave maior do que a chave de busca; em caso contrário, ela retorna o número de chaves. Os parâmetros dessa função são:

- `chave` (entrada) — a chave
- `no` (entrada) — ponteiro para o nó no qual será feita a busca

```
static int BuscaEmNoMultiMS(tChave chave, const tNoMultiMS *no)
{
   int i,
       nChaves = no->nFilhos - 1; /* Número de chaves do nó */

      /* Procura no nó uma chave que seja maior do que ou igual à chave de busca */
```

```
   for (i = 0; i < nChaves; ++i)
         /* Verifica se tal chave foi encontrada */
      if (chave <= no->chaves[i].chave)
         return i;   /* A chave foi encontrada */

      /* A chave de busca é maior do que qualquer chave do nó */
   return nChaves; /* A chave NÃO foi encontrada */
}
```

6.3.5 Inserção

A função `InsereMultiMS()`, vista a seguir, é responsável por inserção em árvores multidirecionais de busca implementadas em arquivo. Essa função tem como parâmetros:

- `*posicaoRaiz` (entrada/saída) — posição no arquivo da raiz da árvore
- `chaveEIndice` (entrada) — ponteiro para um par chave/índice que será inserido
- `streamArvore` (entrada) — stream associado ao arquivo que contém a árvore

O retorno dessa função é `1`, se a inserção ocorrer, ou `0`, se não ocorrer inserção porque a chave já existe na árvore.

```
int InsereMultiMS(int *posicaoRaiz, tChaveIndice *chaveEIndice, FILE *streamArvore)
{
   tNoMultiMS no, novoNo;
   int        pNo, pNovoNo,
              encontrado,    /* Indicará se a chave foi encontrada */
              indiceDaChave; /* Índice da chave no nó onde ela */
                             /* encontrada ou inserida         */

      /* O stream que representa o arquivo que contém a árvore não pode ser NULL */
   ASSEGURA( streamArvore, ERRO_STREAM_NULL(streamArvore, InsereMultiMS) );

   if (*posicaoRaiz == POSICAO_NULA) {
         /* A árvore ainda não foi criada */
      IniciaNoMultiMS(chaveEIndice, &no);
      *posicaoRaiz = 0;

         /* Armazena a raiz da árvore na posição 0 do arquivo*/
      EscreveNoMultiMS(streamArvore, 0, &no);

      return 1;
   }
         /* Tenta encontrar o nó que contém a chave */
   pNo = EncontraNoMultiMS( streamArvore, chaveEIndice->chave, &indiceDaChave,
                            &encontrado );

      /* Se a chave foi encontrada, não há mais nada */
      /* a fazer, pois ela é considerada primária    */
   if (encontrado)
      return 0; /* Não houve inserção */

   /* Neste ponto, sabe-se que a chave não foi encontrada */

      /* Lê no arquivo o nó cuja posição foi       */
      /* retornada pela função EncontraNoMultiMS() */
   LeNoMultiMS(streamArvore, pNo, &no);

      /* Verifica se há espaço para inserção nesse nó */
   if (no.nFilhos < G) {
      /* Há espaço para inserção nesse nó */

         /* Usando esse esquema de inserção, se o nó tem espaço sobrando */
         /* ele é uma folha. Então tenta-se inserir a chave no nó-folha  */
```

```
         /* cuja posição é indicada por 'pNo'.                          */
      InsereEmFolhaMultiMS( streamArvore, pNo, indiceDaChave, *chaveEIndice );

      return 1;  /* Serviço completo */
   }

      /*** Neste ponto, sabe-se que não há nenhum nó ***/
      /*** na árvore com espaço para conter a chave  ***/

      /* É necessário criar um novo nó para colocar a chave */
   IniciaNoMultiMS(chaveEIndice, &novoNo);

      /* O novo nó será armazenado ao final do arquivo; portanto sua posição */
      /* no arquivo será igual ao número de registros (nós) do arquivo.      */
      /* NB: a indexação do arquivo começa em zero; por isso, não se soma    */
      /* 1 a esse valor).                                                    */
   pNovoNo = NumeroDeRegistros(streamArvore, sizeof(tNoMultiMS));

      /* O nó encontrado não tinha espaço para inserção da chave. Portanto */
      /* o índice da chave retornado pela função EncontraNoMultiMS() é o   */
      /* índice do novo nó que será filho do nó encontrado                 */
   no.filhos[indiceDaChave] = pNovoNo;

      /* O nó encontrado foi alterado; é preciso atualizá-lo no arquivo */
   EscreveNoMultiMS(streamArvore, pNo, &no);

   /****                                    ****/
   /**** Insere o novo nó ao final do arquivo ****/
   /****                                    ****/

      /* Primeiro, tenta-se mover o apontador do arquivo para o final */
      /* do arquivo. Se isso não for possível, aborta-se o programa   */
   MoveApontador(streamArvore, 0, SEEK_END);

      /* Agora tenta-se escrever o nó no arquivo. Se */
      /* isso não for possível, aborta-se o programa */
   fwrite(&novoNo, sizeof(tNoMultiMS), 1, streamArvore);
   ASSEGURA(!ferror(streamArvore), ERRO_FWRITE(InsereMultiMS));

   return 1;  /* Serviço completo */
}
```

A função `NumeroDeRegistros()` invocada por `InsereMultiMS()` calcula o número de registros de um arquivo e sua implementação é relativamente fácil quando esses registros são do mesmo tamanho (o que é o caso aqui).

A função `InsereMultiMS()` chama a função `IniciaNoMultiMS()` para iniciar um nó contendo apenas um par chave/índice e a implementação dessa última função é a seguinte:

```
static void IniciaNoMultiMS(const tChaveIndice *chaveIndice, tNoMultiMS *pNo)
{
   int i;

   if (chaveIndice)
      pNo->chaves[0] = *chaveIndice; /* A chave será a primeira do nó */

      /* O número de filhos num nó de uma árvore multidirecional é igual */
      /* ao número de chaves mais um, a não ser que não haja chave       */
   pNo->nFilhos = chaveIndice ? 2 : 0;

      /* Inicia todos os possíveis filhos do nó */
   for (i = 0; i < G; ++i)
      pNo->filhos[i] = POSICAO_NULA; /* Um novo nó ainda não possui filhos */
}
```

A função `IniciaNoMultiMS()`, definida acima, usa como parâmetros:

- `chaveIndice` (entrada) — um par chave/índice a ser armazenado no nó
- `pNo` (saída) — endereço do nó recém-criado

O primeiro parâmetro dessa função poderá ser **NULL**. Nesse caso, será criado um nó vazio (i.e., sem nenhuma chave).

A função `EncontraNoMultiMS()` chamada por `InsereMultiMS()` procura uma chave numa árvore multidirecional de busca armazenada em arquivo e retorna a posição no arquivo do nó que contém a chave, se ela for encontrada. Se a chave não for encontrada, o nó cuja posição é retornada é o pai do nó no qual ela deveria estar. Os parâmetros dessa função são:

- `stream` (entrada) — stream associado ao arquivo que contém a árvore
- `chave` (entrada) — a chave de busca
- `*posicaoDaChaveNoNo` (saída) — conterá o índice da chave no nó
- `*encontrado` (saída) — indicará se a chave foi encontrada

```
static int EncontraNoMultiMS( FILE *stream, tChave chave, int *posicaoDaChaveNoNo,
                              int *encontrado )
{
   int       p,  /* p conterá a posição desejada    */
             q,  /* q conterá a posição do pai de p */
             posChave;
   tNoMultiMS umNo;

      /* Verifica se o stream é NULL */
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, EncontraNoMultiMS));

      /* Tenta mover o apontador de arquivo para seu início */
   MoveApontador(stream, 0, SEEK_SET);

   p = 0; /* Começa busca na raiz da árvore */
   q = POSICAO_NULA; /* A raiz não tem pai */

   while (p != POSICAO_NULA) {
      LeNoMultiMS(stream, p, &umNo); /* Lê um nó da árvore na posição p do arquivo */

         /* Tenta encontrar a chave no nó corrente */
      posChave = BuscaEmNoMultiMS(chave, &umNo);

         /* Verifica se a chave foi encontrada */
      if( ( posChave < umNo.nFilhos - 1) && chave == umNo.chaves[posChave].chave ) {
         *encontrado = 1;  /* A chave foi encontrada */
         *posicaoDaChaveNoNo = posChave;

            /* Retorna a posição do nó no arquivo que contém a árvore */
         return p;
      }

         /* A chave ainda não foi encontrada; desce um nível na árvore.  */
         /* Antes que p passe a referir-se a um filho, guarda seu valor. */
      q = p;
      p = umNo.filhos[posChave];
   }

   *encontrado = 0; /* A chave não foi encontrada */

   *posicaoDaChaveNoNo = posChave; /* Posição onde a chave deve ser inserida */

   return q;  /* Retorna o pai do nó p */
}
```

A função `InsereEmFolhaMultiMS()` chamada pela função `InsereMultiMS()` insere uma chave numa folha incompleta de uma árvore multidirecional de busca armazenada em arquivo e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que contém a árvore
- `posicaoNo` (entrada) — posição no arquivo do nó no qual será feita a inserção
- `posicaoDaChave` (entrada) — posição da nova chave no array de chaves do nó
- `chaveEIndice` (entrada) — a chave e seu respectivo índice que serão inseridos

```
static void InsereEmFolhaMultiMS( FILE *stream, int posicaoNo, int posicaoDaChave,
                                  tChaveIndice chaveEIndice )
{
   int        i;
   tNoMultiMS umNo;

      /* Verifica se o stream é válido */
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, InsereEmFolhaMultiMS));

      /* Lê o nó-folha no arquivo que contém a árvore */
   LeNoMultiMS(stream, posicaoNo, &umNo);

      /* O número de filhos do nó deve ser estritamente menor do que */
      /* o grau da árvore. Se não o for, o nó não é uma folha, deve  */
      /* haver algo errado e o programa será abortado.               */
   ASSEGURA( umNo.nFilhos < G, "ERRO: Tentativa de inserir em folha completa" );

      /* Abre espaço para a nova chave */
   for (i = umNo.nFilhos - 1; i > posicaoDaChave; --i)
      umNo.chaves[i] = umNo.chaves[i - 1];

      /* Armazena a nova chave */
   umNo.chaves[posicaoDaChave] = chaveEIndice;

   ++umNo.nFilhos; /* O número de filhos do nó é acrescido de um */

   EscreveNoMultiMS(stream, posicaoNo, &umNo); /* Atualiza nó no arquivo */
}
```

6.3.6 Remoção

A função `RemoveChaveMultiMS()`, que será definida a seguir, o usa os seguintes parâmetros:

- `*streamArvore` (entrada/saída) — stream associado ao arquivo que contém a árvore de busca
- `chave` (entrada) — a chave que será removida

A função `RemoveChaveMultiMS()` retorna `1`, se a remoção for bem-sucedida, ou `0`, em caso contrário.

```
int RemoveChaveMultiMS(FILE **streamArvore, tChave chave)
{
   tChaveIndice sucessoraEIndice;
   tNoMultiMS   no, /* Armazenará cada nó lido no arquivo */
                noPai; /* Armazenará o pai de 'no' */
   int          i,
                posNo = 0, /* Posição no arquivo do nó que contém a chave, */
                           /* se ela for encontrada. A busca começa na     */
                           /* raiz que está na posição 0                   */
                posNoPai = POSICAO_NULA, /* Posição de 'noPai' */
                iPai; /* Índice do filho do penúltimo nó visitado que */
                      /* contém a posição no arquivo de 'no'          */

   ASSEGURA(*streamArvore, ERRO_STREAM_NULL(streamArvore, RemoveChaveMultiMS) );

      /* A busca inicia na raiz e prossegue até que se */
      /* encontre a chave a ser removida ou um nó nulo */
```

```
while (posNo != POSICAO_NULA) {
      /* Lê o nó na posição indicada por 'posNo' */
   LeNoMultiMS(*streamArvore, posNo, &no);

      /* Tenta encontrar a chave no nó recém-recuperado */
   i = BuscaEmNoMultiMS(chave, &no);

       /* Verifica se a chave foi encontrada */
   if (i < no.nFilhos - 1 && chave == no.chaves[i].chave)
      break; /* A chave foi encontrada */

      /* Atualiza os valores de 'posNoPai', 'noPai' e   */
      /* 'iPai' antes de descer mais um nível na árvore */
   posNoPai = posNo;
   noPai = no;
   iPai = i;

   posNo = no.filhos[i]; /* Desce mais um nível na árvore */
}

if (posNo == POSICAO_NULA)
   return 0; /* Chave não foi encontrada */

/* Neste ponto sabe-se que 'no' contém a chave que será removida */
/* e que a posição dessa chave nesse nó é i. O índice do filho   */
/* de'noPai' que contém a posição em arquivo de 'no' é 'iPai'.   */

/* O nó será logicamente removido, pois ele não poderá mais ser acessado, mas*/
/* continuará a ocupar espaço no arquivo até que a árvore seja reconstruída  */

   /* Verifica quantos filhos a chave a ser removida possui */
if ( no.filhos[i] == POSICAO_NULA || no.filhos[i + 1] == POSICAO_NULA ) {
   /* A chave tem, no máximo, um filho */

      /* Verifica se o nó só tem uma chave */
   if(no.nFilhos == 2) {
      /* A última chave do nó será removida. Logo esse nó também será removido */

         /* Se o nó a ser removido não tem pai, trata-se da raiz */
      if (posNoPai == POSICAO_NULA) {
         *streamArvore = NULL; /* A árvore acabou-se */
         return 1; /* Game over */
      }
         /* Verifica quantos filhos o nó a ser removido possui */
      if(no.filhos[i] != POSICAO_NULA) {
            /* O nó tem filho esquerdo. O pai desse filho */
            /* passará a ser o pai do nó a ser removido   */
         noPai.filhos[iPai] = no.filhos[i];
         EscreveNoMultiMS(*streamArvore, posNoPai, &noPai);
      } else if(no.filhos[i + 1] != POSICAO_NULA) {
            /* O nó tem filho direito. O pai desse filho */
            /* passará a ser o pai do nó a ser removido  */
         noPai.filhos[iPai] = no.filhos[i + 1];
         EscreveNoMultiMS(*streamArvore, posNoPai, &noPai);
      } else {
            /* O nó não tem nenhum filho. O pai do */
            /* nó a ser removido passa a ser nulo  */
         noPai.filhos[iPai] = POSICAO_NULA;
      }
   } else {  /* Havia mais de uma chave no nó */
      CompactaNoMultiMS(&no, i); /* Essa função completa a remoção */
```

```
            /* O nó precisa ser atualizado em arquivo */
         EscreveNoMultiMS(*streamArvore, posNo, &no);
      }
      return 1;
   }

   /* Neste ponto, sabe-se que a chave a ser removida possui dois filhos */
   /* não vazios. Nesse caso, substitui-se a chave a ser removida por    */
   /* sua sucessora imediata e remove-se essa sucessora                  */

      /* A chave sucessora imediata da chave a ser removida é a menor chave da */
      /* subárvore cuja raiz é o filho direito da chave a ser removida         */
   sucessoraEIndice = MenorChaveMultiMS(*streamArvore, no.filhos[i + 1]);

      /* Agora chama-se esta função recursivamente para remover a chave */
      /* sucessora. Haverá apenas uma chamada recursiva porque a chave  */
      /* sucessora possui, no máximo, um filho.                         */
   if (!RemoveChaveMultiMS(streamArvore, sucessoraEIndice.chave))
      return 0;

      /* Substitui a chave a ser removida por sua sucessora */
   no.chaves[i] = sucessoraEIndice;
   EscreveNoMultiMS(*streamArvore, posNo, &no);

   return 1;
}
```

É importante observar que, na prática, a remoção da chave da árvore deveria vir acompanhada da respectiva remoção do registro correspondente no arquivo de registro, o que não é feito aqui.

A função `CompactaNoMultiMS()`, chamada por `RemoveChaveMultiMS()` e apresentada a seguir, é usada para compactar um nó após uma operação de remoção e tem como parâmetros:

- `pNo` (entrada) — ponteiro para o nó que será compactado
- `posicao` (entrada) — posição da chave removida

```
static void CompactaNoMultiMS(tNoMultiMS *pNo, int posicao)
{
   int i;

   ASSEGURA(pNo->filhos[posicao] == POSICAO_NULA ||
            pNo->filhos[posicao + 1] == POSICAO_NULA,
            "Nenhum dos filhos da chave removida e' nulo");
      /* Primeiro, move as chaves */
   for (i = posicao; i < pNo->nFilhos - 2; ++i)
      pNo->chaves[i] = pNo->chaves[i + 1];

      /* A movimentação de filhos depende de qual dos filhos  */
      /* da chave removida é vazio. Se o filho esquerdo não   */
      /* for vazio, a movimentação começa uma posição adiante */
   if (pNo->filhos[posicao])
      posicao++;

      /* Move os filhos do nó */
   for (i = posicao; i < pNo->nFilhos - 1; ++i)
      pNo->filhos[i] = pNo->filhos[i + 1];

   --pNo->nFilhos; /* O número de filhos (e de chaves) diminuiu */
}
```

A função `MenorChaveMultiMS()`, chamada por `RemoveChaveMultiMS()` e vista abaixo, encontra a menor chave de uma árvore multidirecional de busca armazenada em arquivo e é usada aqui para achar a chave sucessora

imediata de uma chave a ser removida, quando essa última chave possui dois filhos. Os parâmetros da função `MenorChaveMultiMS()` são:

- `streamArvore` (entrada) — stream associado ao arquivo que armazena a árvore
- `raiz` (entrada) — posição da raiz da árvore no arquivo que contém a árvore

```
tChaveIndice MenorChaveMultiMS(FILE *streamArvore, int raiz)
{
   tNoMultiMS umNo; /* Armazena um nó */
   int       pos;  /* Armazena a posição de um nó */

      /* Verifica se o stream que representa a árvore é válido */
   ASSEGURA(streamArvore, ERRO_STREAM_NULL(streamArvore, MenorChaveMultiMS));

   LeNoMultiMS(streamArvore, raiz, &umNo); /* Lê o nó que representa a raiz da árvore */

   pos = umNo.filhos[0]; /* Armazena em 'pos' o filho mais à esquerda da raiz */

      /* Encontra o nó mais à esquerda na árvore */
   while (pos != POSICAO_NULA) {
         /* Lê o nó cuja posição no arquivo é indicada por 'pos' */
      LeNoMultiMS(streamArvore, pos, &umNo);

      pos = umNo.filhos[0]; /* Desce-se até o filho mais à esquerda deste nó */
   }
   return umNo.chaves[0]; /* Retorna o par chave/índice que tem a menor chave */
}
```

6.4 Árvores B

6.4.1 Conceitos

Árvore B de ordem (ou grau) G é uma árvore multidirecional de busca de ordem G balanceada na qual cada nó, exceto a raiz, possui no mínimo $\lceil G/2 \rceil$ filhos. Por exemplo, numa árvore B de ordem *9* ou *10*, cada nó possui, no mínimo, *5* filhos (ou *4* chaves).

Árvores B apresentam as seguintes vantagens principais com relação às árvores multidirecionais de busca descendentes: (1) o número máximo de nós acessados numa busca é pequeno e (2) todos os nós, exceto a raiz, são completos ou semicompletos, de modo que pouco espaço é desperdiçado. Árvores B e suas variantes são muito utilizadas na prática em sistemas de gerenciamento de arquivos e de bancos de dados, por exemplo.

Muitas vezes, em vez de grau, árvores B são especificadas usando-se o número mínimo de filhos que um nó pode ter, que é o teto da divisão do grau por dois. Obviamente, essa é uma formulação equivalente, mas que introduz uma complicação a mais, visto que todas as árvores têm sido especificadas por seus graus até aqui.

Alguns autores usam o termo *árvore B* para representar uma família inteira de árvores balanceadas armazenadas em memória secundária, enquanto outros autores usam o mesmo termo para referir-se ao tipo específico de estrutura de dados descrito aqui, que é a árvore originalmente proposta por Bayer e McCreight em 1972 (v. **Bibliografia**).

6.4.2 Busca

Como árvores B e árvores multidirecionais descendentes diferem apenas no modo como inserções e remoções são efetuadas, uma operação de busca numa árvore B segue exatamente o mesmo algoritmo descrito na **Seção 6.1.2** para busca em árvores multidirecionais descendentes.

6.4.3 Inserção

Os dois primeiros passos do algoritmo de inserção em árvores B são idênticos aos respectivos passos do algoritmo de inserção em árvores multidirecionais descendentes apresentados na **Seção 6.1.3**:

[1] Encontra-se o nó-folha dentro do qual a chave será inserida, usando uma função como a função `EncontraNoMultiMS()` apresentada na **Seção 6.3.5**.

[2] Se a folha não estiver completa, insere-se a chave usando uma função como `InsereEmFolhaMultiMS()` apresentada na referida seção

Árvores B e árvores multidirecionais descendentes diferem no terceiro passo do algoritmo de inserção; i.e., quando o nó-folha encontrado no segundo passo é completo. Nesse caso, em vez de criar-se um novo nó contendo apenas uma chave, **divide-se** a folha completa em duas folhas.

Suponha que o grau da árvore *G* seja ímpar. Então as *G* chaves (*G – 1* chaves na folha completa mais a chave a ser inserida) são divididas em três grupos: as ⌊*G/2*⌋ chaves menores são colocadas na folha da esquerda, as ⌊*G/2*⌋ chaves maiores são colocadas na folha da direita e a chave do meio é inserida no nó-pai seguindo o mesmo procedimento. Os filhos à esquerda e à direita da chave colocada no nó-pai serão, respectivamente, as folhas esquerda e direita resultantes da divisão do nó-folha original. A **Figura 6–13** apresenta um exemplo de inserção numa árvore B de grau *5*. Note como um nó é dividido e como uma chave sobe para um nó no nível imediatamente superior.

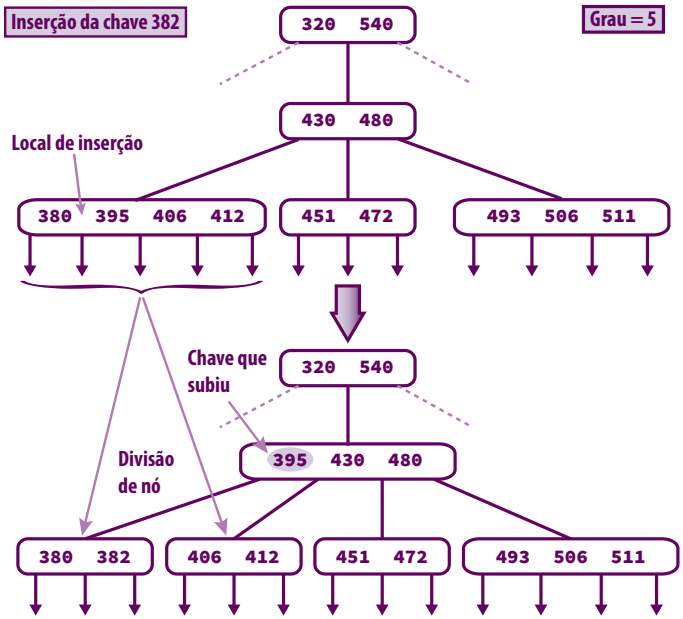


FIGURA 6–13: INSERÇÃO EM ÁRVORE B

Se a ordem *G* for par, as primeiras *G/2* chaves são colocadas na folha da esquerda e as últimas *(G/2) – 1* chaves são colocadas na folha da direita ou, alternativamente, pode-se colocar *(G/2) – 1* chaves na folha da esquerda e *G/2* chaves na folha da direita. Essas duas abordagens são respectivamente denominadas **tendência esquerda** e **tendência direita**. Novamente, a chave do meio passa para o nó-pai.

A **Figura 6–14** apresenta um exemplo de inserção numa árvore B de ordem 4 com tendência esquerda, enquanto a **Figura 6–15** apresenta um exemplo de inserção numa árvore B de ordem 4 com tendência direita.

Quando o nó-pai para o qual a chave do meio é deslocada está completo, repete-se o procedimento, como se ela fosse uma nova chave sendo inserida. Esse processo pode continuar até que a raiz seja atingida e, se ela também estiver completa, ela é dividida e cria-se uma nova raiz, como mostra a **Figura 6–16**. Nessa figura, o valor da chave a ser inserida é *27* e o grau da árvore é *4*. Note que, como mostra a **Figura 6–16 (a)**, o nó no qual a inserção deverá ser efetuada, encontra-se completo, de modo que ocorrerá uma divisão de nós. A **Figura 6–16 (b)** mostra essa divisão de nós, com a subsequente subida de uma chave para o nível superior. Observe nessa última

figura, que o novo nó (i.e., aquele contendo a chave *35*) resultante da referida divisão fica temporariamente sem pai, pois, ao final do processo de inserção, ele será filho direito da chave que sobe, que, como mostra a última figura, é aquela cujo valor é *30*. A **Figura 6–16 (c)** mostra o resultado final do processo de inserção. Note que, quando a chave que subiu foi inserida na raiz da árvore, ocorreu outra divisão de nós, de modo que a nova chave que subiu (nesse caso, aquela cujo valor é *40*) constituiu uma nova raiz contendo apenas essa última chave.

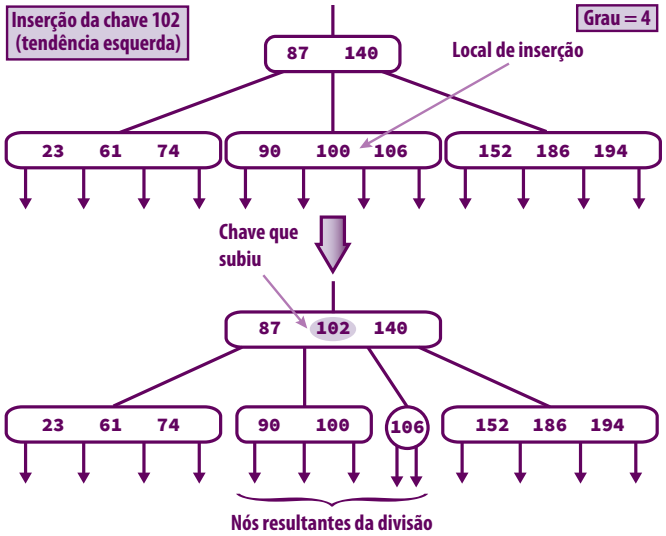


Figura 6–14: Inserção em Árvore B com Tendência Esquerda

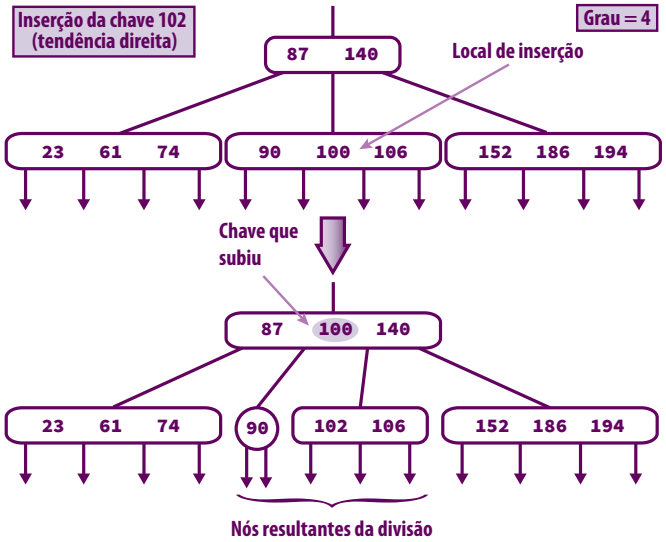


Figura 6–15: Inserção em Árvore B com Tendência Direita

É importante observar que qualquer inserção mantém uma árvore B balanceada e que a altura de uma árvore B aumenta quando a raiz é dividida e uma nova raiz é criada, ao passo que a largura de uma árvore B aumenta quando outros nós são divididos. Em virtude dessa abordagem de inserção, a altura de uma árvore aumenta apenas quando uma nova raiz é criada. Desse modo, árvores B crescem de baixo para cima e, por isso, são denominadas **árvores multidirecionais ascendentes**.

Para que uma inserção ocorra numa árvore B, é necessária uma busca para localizar a folha na qual a chave deve ser inserida. Essa busca percorre um caminho que começa na raiz da árvore e termina na referida folha. Quando essa folha é completa, a inserção retrocede nesse mesmo caminho dividindo todos os nós completos

encontrados no sentido inverso (i.e., da referida folha até a raiz). Esse percurso inverso encerra quando é encontrado um nó incompleto onde se pode fazer uma inserção sem necessidade de divisão ou quando a raiz da árvore é encontrada. Então se estiver completa, ela é dividida e uma nova raiz é criada.

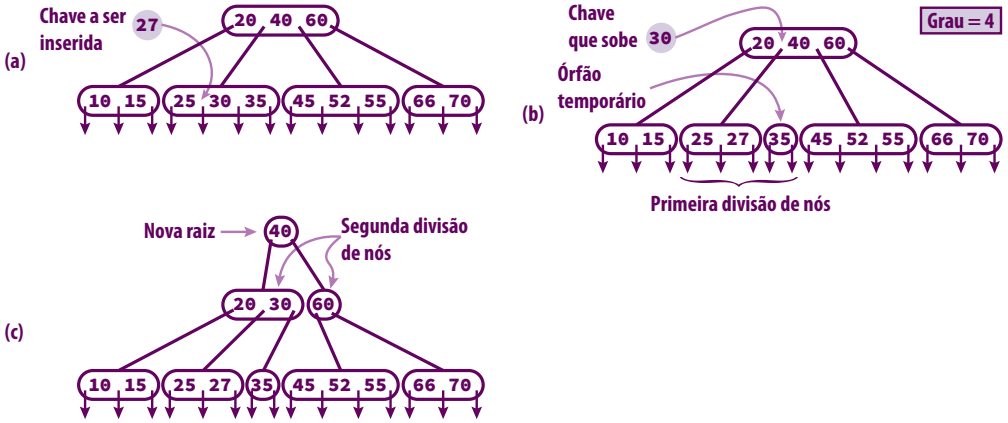


Figura 6–16: Divisão de Nó em Árvore B com Criação de Nova Raiz

De acordo com essa descrição de algoritmo, o passo inicial da inserção consiste em encontrar o caminho até um nó-folha, em vez de simplesmente encontrar esse nó (como ocorre com inserção em árvores multidirecionais descendentes). Esse caminho é armazenado numa pilha na qual cada elemento contém o nó, seu endereço e seu índice entre seus irmãos. Em seguida, a inserção é feita explorando-se o caminho inverso (i.e., desempilhando-se) conforme foi descrito antes.

O algoritmo de inserção em árvores B segue os passos descritos na **Figura 6–17**.

Algoritmo InsereEmÁrvoreB

Entrada: A chave e a raiz da árvore B

Saída: A árvore modificada, se a operação for bem-sucedida, e um valor informando o sucesso ou fracasso da operação

1. Se a árvore estiver vazia, crie um novo nó, insira a chave nele e encerre informando o sucesso da operação
2. Faça uma busca usando a chave a ser inserida e empilhe os nós que se encontram no caminho desde a raiz até o nó no qual a chave será inserida
3. Se a chave for encontrada, esvazie a pilha e encerre informando o fracasso da operação (pois a chave é primária)
4. Atribua à variável *chave* a chave recebida como parâmetro
5. Atribua o endereço nulo à variável *filhoDireito*
6. Enquanto a pilha não estiver vazia, faça o seguinte
   - 6.1 Desempilhe o nó no qual ocorrerá uma inserção
   - 6.2 Se o nó no qual a chave será inserida não estiver completo, insira a chave e seu filho direito nesse nó e encerre informando o sucesso da operação
   - 6.3 Caso contrário, divida o nó usando o algoritmo **DivideNóEmÁrvoreB**
   - 6.4 Atualize as variáveis *chave* e *filhoDireito* com os valores retornados pela função **DivideNóEmÁrvoreB**
7. Crie uma nova raiz tendo como filho esquerdo a antiga raiz e como filho direito o nó resultante da última divisão de nós

Figura 6–17: Algoritmo de Inserção em Árvore B

O algoritmo de divisão de nós que complementa o algoritmo de inserção em árvores B é apresentado na **Figura 6–18**.

ALGORITMO DIVIDENÓEMÁRVOREB

ENTRADA: A chave a ser inserida, a posição de inserção, o filho direito da chave e o nó completo da árvore B no qual ocorrerá a inserção

SAÍDA: O novo nó resultante da divisão e a chave que subirá para um nó no próximo nível superior

1. Se a posição de inserção estiver na parte superior do array de chaves, faça o seguinte:
   - **1.1** Copie a metade das chaves do nó dividido menos um para um novo nó direito
   - **1.2** Insira a nova chave no nó da direita
   - **1.3** A chave que subirá para o próximo nível superior da árvore será a última chave do nó que foi dividido
2. Se a posição de inserção estiver na parte inferior do array de chaves, faça o seguinte:
   - **2.1** Copie a metade das chaves do nó dividido para o novo nó
   - **2.2** Insira a nova chave no nó que foi dividido
   - **2.3** A chave que subirá para o próximo nível superior da árvore será a última chave do nó que foi dividido após a inserção da nova chave
3. Se a posição de inserção coincidir com o índice central do array de chaves, faça o seguinte:
   - **3.1** Copie a metade das chaves do nó dividido menos um para o novo nó
   - **3.2** A chave que subirá para o nível superior da árvore será a nova chave

FIGURA 6–18: ALGORITMO DE DIVISÃO DE NÓ EM ÁRVORE B

6.4.4 Remoção

A remoção de uma chave pode requerer a combinação de nós, de modo que a árvore continue satisfazendo os critérios estabelecidos para árvores B, conforme será visto a seguir.

Algoritmos

O algoritmo de remoção em árvores B segue os passos descritos na **Figura 6–19**.

ALGORITMO REMOVEEMÁRVOREB

ENTRADA: A chave e a árvore B

SAÍDA: A árvore modificada, se a operação for bem-sucedida, e um valor indicando o sucesso ou fracasso da operação

1. Faça uma busca pela chave a ser removida, empilhando os nós que se encontram no caminho da raiz até o nó que contém essa chave
2. Se a chave não for encontrada, encerre a operação informando que ela fracassou
3. Se a chave for encontrada numa folha
   - **3.1** Remova a chave usando **REMOVEEMFOLHADEÁRVOREB**
   - **3.2** Encerre indicando que a operação foi bem-sucedida
4. Se a chave for encontrada num nó que não é folha
   - **4.1** Substitua a chave por sua sucessora (que se encontra numa folha)
   - **4.2** Remova a chave sucessora usando **REMOVEEMFOLHADEÁRVOREB**
   - **4.3** Encerre indicando que a operação foi bem-sucedida

FIGURA 6–19: ALGORITMO DE REMOÇÃO EM ÁRVORE B

O algoritmo **RemoveEmFolhaDeÁrvoreB** de remoção de uma chave de uma folha de árvore B segue os passos descritos na **Figura 6–20**.

Algoritmo RemoveEmFolhaDeÁrvoreB

Entrada: Uma pilha contendo os nós encontrados desde a raiz até a folha

Saída: A pilha esvaziada e a árvore B alterada

1. Desloque para uma posição anterior cada chave que segue a chave a ser removida
2. Decremente o número de filhos do nó
3. Ajuste o nó usando **JuntaNósDeÁrvoreB**
4. Esvazie a pilha

Figura 6–20: Algoritmo de Remoção em Folha de Árvore B

O algoritmo **JuntaNósDeÁrvoreB**, que combina um nó com outros de uma árvore B quando ele fica com grau abaixo do mínimo estabelecido, segue os passos descritos na **Figura 6–21**.

Algoritmo JuntaNósDeÁrvoreB

Entrada: Uma folha e uma pilha contendo os nós encontrados desde a raiz até ela

Saída: A pilha e a árvore B alteradas

1. Se o número de chaves da folha (*N*) não for menor do que o número mínimo de chaves permitido, encerre
2. Se o irmão vizinho à esquerda (*E*) do nó *N* tiver um número de chaves maior do que o mínimo (v. **Figura 6–22**)
   - **2.1** Insira a sucessora da última chave do nó *E* (encontrada no nó-pai) no início das chaves do nó *N*
   - **2.2** Faça o filho esquerdo da (nova) primeira chave de *N* apontar para o filho direito da última chave de *E*
   - **2.3** Substitua a sucessora da última chave do nó *E* por essa chave
   - **2.4** Decremente o número de filhos de *E*
   - **2.5** Incremente o número de filhos de *N*
   - **2.6** Encerre
3. Se o irmão vizinho à direita (*D*) do nó *N* tiver um número de chaves maior do que o mínimo (v. **Figura 6–23**)
   - **3.1** Acrescente a sucessora da última chave do nó *N* ao final das chaves desse nó
   - **3.2** Faça o filho direito da (nova) última chave de *N* apontar para o filho esquerdo da primeira chave de *D*
   - **3.3** Substitua a sucessora da última chave do nó *N* pela primeira chave do nó *D*
   - **3.4** Desloque cada chave e filho de *D* uma posição para trás
   - **3.5** Decremente o número de filhos de *D*
   - **3.6** Incremente o número de filhos de *N*
   - **3.7** Encerre
4. Se nenhum irmão vizinho do nó *N* tiver um número de nós maior do que o mínimo
   - **4.1** Se *N* tem um irmão vizinho direito *D* (v. **Figura 6–24**)
     - **4.1.1** Acrescente a chave sucessora (*S*) da última chave do nó *N* ao final das chaves de *N*

Continua

Figura 6–21: Algoritmo de Junção de Nós em Árvore B

Algoritmo JuntaNósDeÁrvoreB (Continuação)

4.1.2 Faça o filho direito da (nova) última chave de *N* apontar para o filho esquerdo da primeira chave de *D*

4.1.3 Desloque cada chave do nó-pai de *N* e seu filho direito, a partir da chave sucessora *S*, para uma posição anterior

4.1.4 Decremente o número de filhos do pai de *N*

4.1.5 Acrescente todas as chaves de *D* e seus filhos direitos ao final das chaves de *N*

4.1.6 Atualize o número de filhos de *N*

4.1.7 Libere o nó *D*

4.1.8 Ajuste o pai de *N* usando **JuntaNósDeÁrvoreB**

4.2 Caso contrário, se *N* tem um irmão vizinho esquerdo *E* (v. **Figura 6–25**)

4.2.1 Acrescente ao final das chaves de *E* a chave sucessora (*S*) da última chave de *E*

4.2.2 Faça o filho direito da (nova) última chave de *E* apontar para o filho esquerdo da primeira chave de *N*

4.2.3 Desloque cada chave e filho direito correspondente do nó-pai de *N*, a partir da chave sucessora *S*, para uma posição anterior

4.2.4 Decremente o número de filhos do pai de *N*

4.2.5 Acrescente cada chave de *N* e seu filho direito ao final das chaves de *E*

4.2.6 Atualize o número de filhos de *E*

4.2.7 Libere o nó *N*

4.2.8 Ajuste o pai de *E* usando **JuntaNósDeÁrvoreB**

Figura 6–21 (Cont): Algoritmo de Junção de Nós em Árvore B

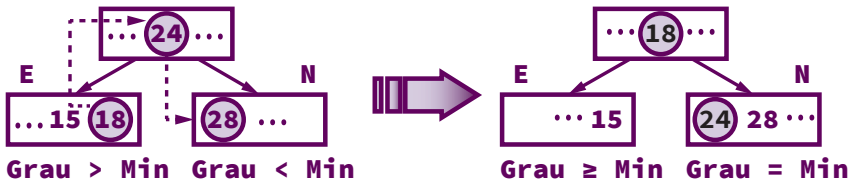


Figura 6–22: Ilustração do Passo 2 do Algoritmo JuntaNósDeÁrvoreB

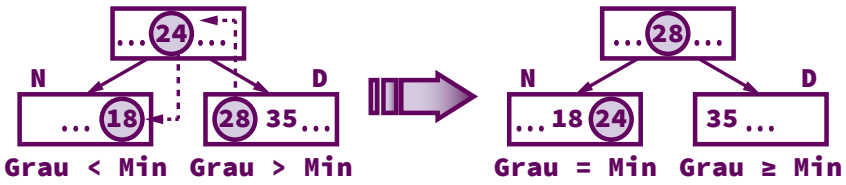


Figura 6–23: Ilustração do Passo 3 do Algoritmo JuntaNósDeÁrvoreB

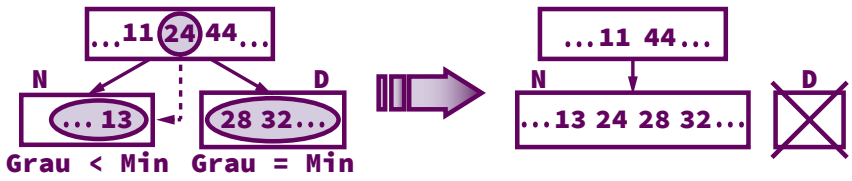


Figura 6–24: Ilustração do Passo 4.1 do Algoritmo JuntaNósDeÁrvoreB

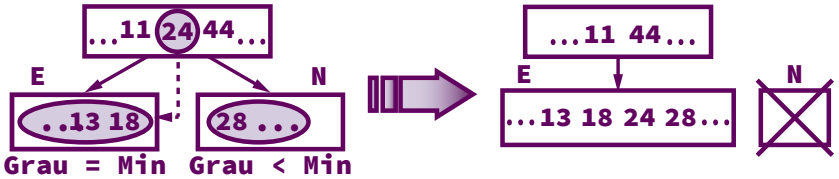


FIGURA 6–25: ILUSTRAÇÃO DO PASSO 4.2 DO ALGORITMO JUNTANÓSDEÁRVOREB

Exemplos de Remoção

Os exemplos a seguir ilustram alguns casos de remoção numa árvore B com ordem igual a *5*. Na **Figura 6–26** a chave que se deseja remover é aquela cujo valor é *8*. Esse é o tipo de remoção mais simples, pois ele não envolve a alteração de outros nós além daquele que contém a chave removida.

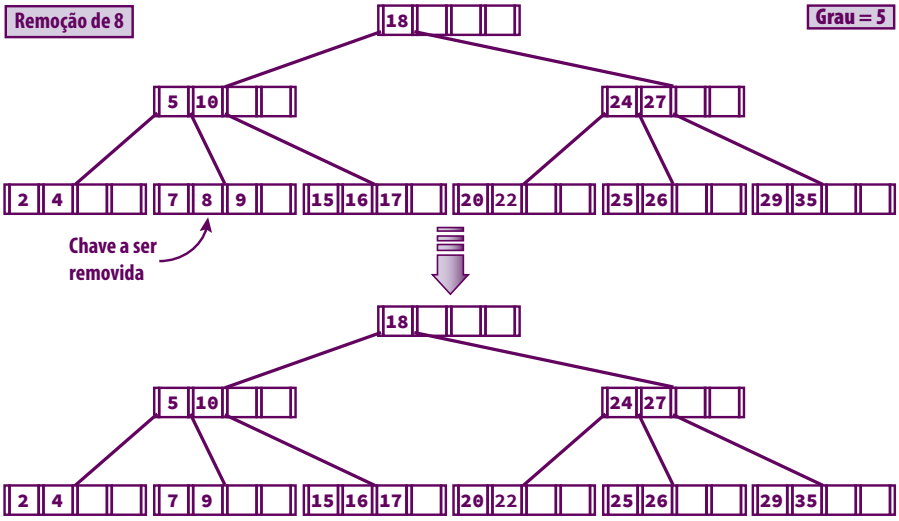


FIGURA 6–26: EXEMPLO DE REMOÇÃO EM ÁRVORE B SEM JUNÇÃO DE NÓS

Na **Figura 6–27**, a chave *7* é removida. Esse exemplo de remoção explora o **Passo 3** do algoritmo JUNTANÓSDEÁRVOREB.

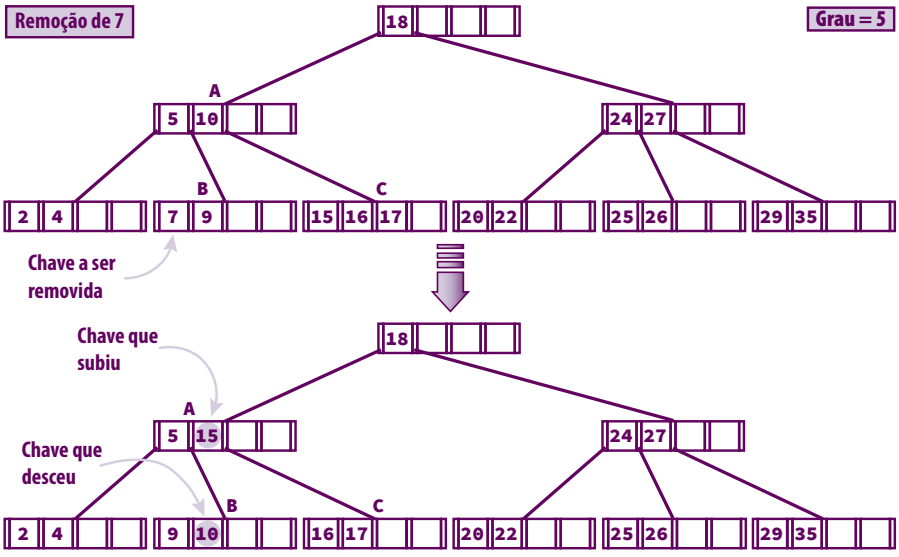


FIGURA 6–27: EXEMPLO DE REMOÇÃO SIMPLES EM ÁRVORE B

A remoção ilustrada na **Figura 6–27** é um pouco mais complicada do que aquela do exemplo anterior, pois ela envolve a alteração de três nós:

- O nó rotulado com *B* é aquele que contém a chave removida. Após a remoção, esse nó ficou com seu número de chaves abaixo do limite permitido, de modo que, para restabelecer o número mínimo de chaves desse nó, ele recebe a doação da chave *10* vinda do nó *A*, que contém seu pai.
- Agora, após essa doação, o nó *A* torna-se deficiente em número de chaves, de forma que o nó *C* lhe cede a chave *15* para suprir essa deficiência.
- Finalmente, os três nós envolvidos nessa operação ficam com números de nós permitidos para árvores B e o processo é encerrado.

A remoção da chave *10* ilustrada na **Figura 6–28** é ainda mais complicada do que aquelas dos exemplos anteriores, pois, desta vez, ocorrem duas junções de nós.

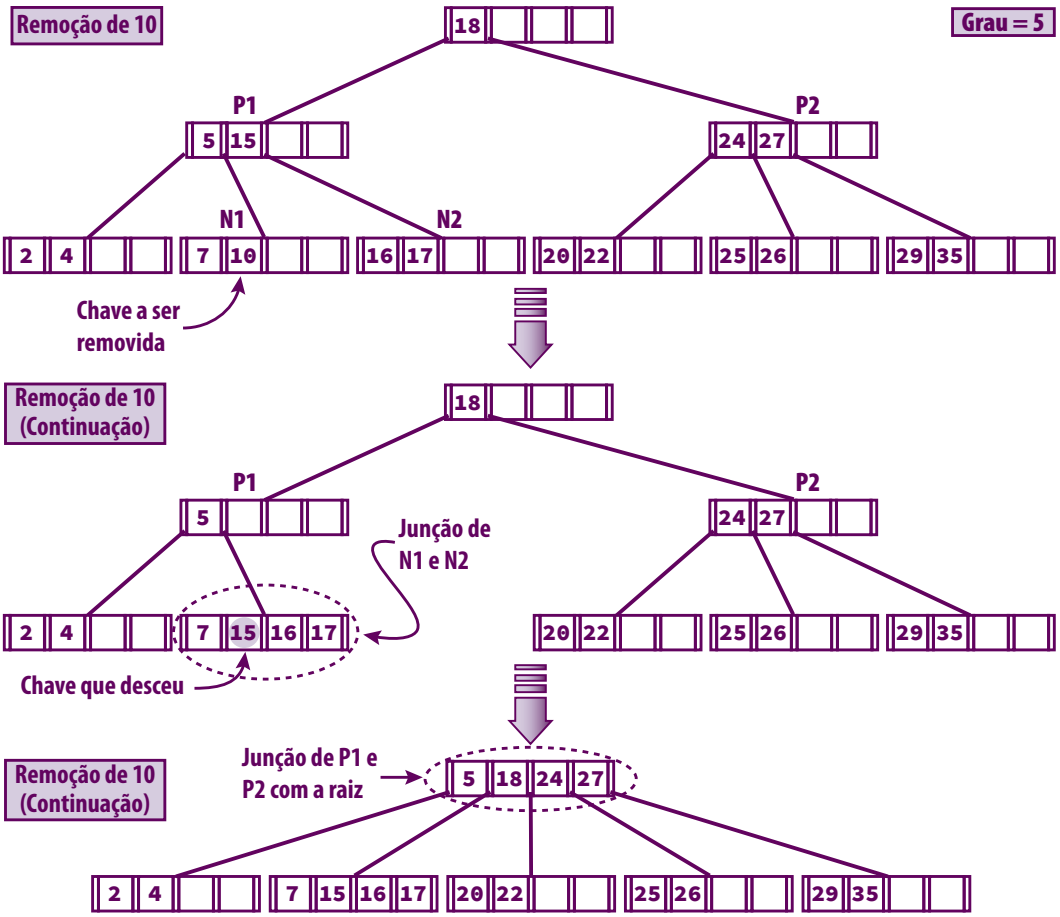


Figura 6–28: Exemplo de Remoção em Árvore B com Junção de Nós

No exemplo ilustrado na **Figura 6–29**, a chave a ser removida é *18*. O que diferencia esse exemplo dos exemplos anteriores é o fato de a chave a ser removida não se encontrar numa folha. Nesse caso, a referida chave é substituída por sua antecessora imediata (i.e., a chave *17*). Como foi visto no **Capítulo 3**, a chave *18* poderia ter sido substituída por sua sucessora imediata, que é a chave *20*. Aqui, a escolha da antecessora é pragmática e didática. Quer dizer, se a substituta da chave a ser removida tivesse sido a sucessora imediata, o nó contendo essa chave substituta ficaria deficiente em número de chaves e precisaria ser recomposto. Enfim o exemplo se tornaria mais longo e mais difícil de entender.

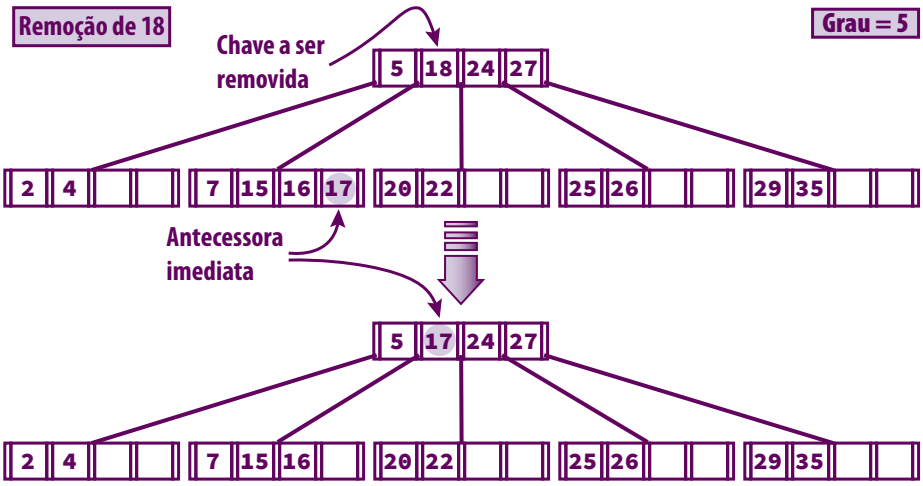


Figura 6–29: Exemplo de Remoção de Nó Interno em Árvore B

Como exemplo mais complexo de remoção numa árvore B, suponha que a árvore da **Figura 6–30** tenha grau igual a *5* e se deseje remover a chave *3*.

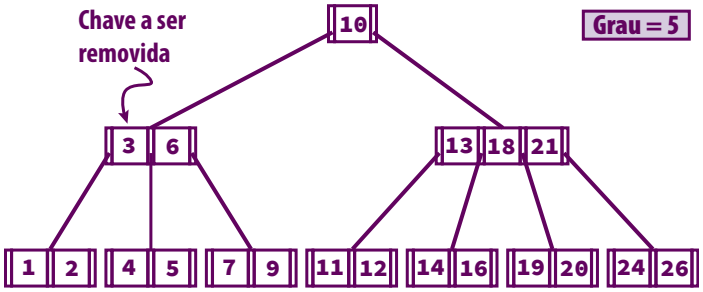


Figura 6–30: Exemplo de Remoção em Árvore B 1

O primeiro passo na remoção da chave *3* é encontrar sua chave sucessora, que é a chave *4*. Essa chave sucessora deve substituir *3*, mas essa substituição deixa o nó contendo *5* com um número de chaves menor do que o mínimo permitido (que é *2*), como mostra a **Figura 6–31**.

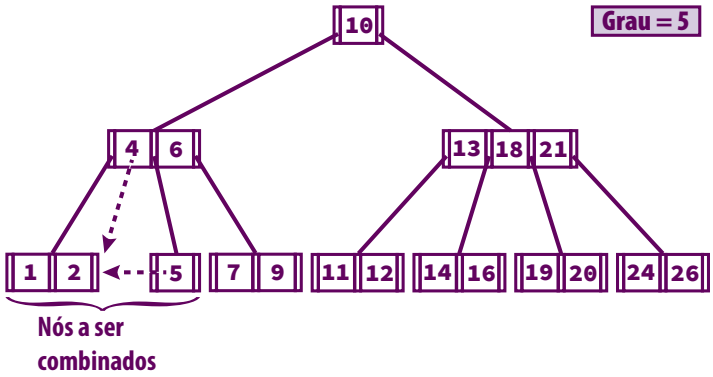


Figura 6–31: Exemplo de Remoção em Árvore B 2

Como nenhum dos irmãos vizinhos do nó contendo *5* tem chaves sobressalentes, deve-se combinar esse nó com um dos seus irmãos. Aqui, combinam-se o nó contendo *1* e *2* com aquele contendo *5* (v. **Figura 6–31**). Além disso, a chave antecessora (i.e., *4*) da chave *5* é acrescentada a essa combinação de nós. O resultado dessa combinação é mostrado na **Figura 6–32**.

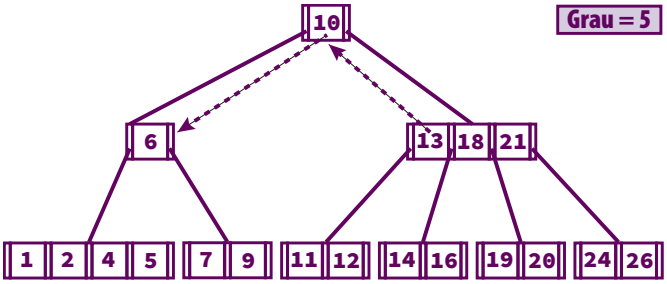


Figura 6–32: Exemplo de Remoção em Árvore B 3

Na **Figura 6–32**, a situação se complica porque o nó contendo *6* fica com um número insuficiente de chaves. Entretanto o irmão desse nó possui uma chave sobressalente. Assim pega-se a chave *13* do irmão do nó deficiente, move-se para o nó-pai e traz-se a chave *10* do nó-pai para juntar-se à chave *6* no nó deficiente (v. **Figura 6–32**). Note que o nó contendo as chaves *11* e *12* passa a ser o filho direito da chave *10*, como mostra a **Figura 6–33**.

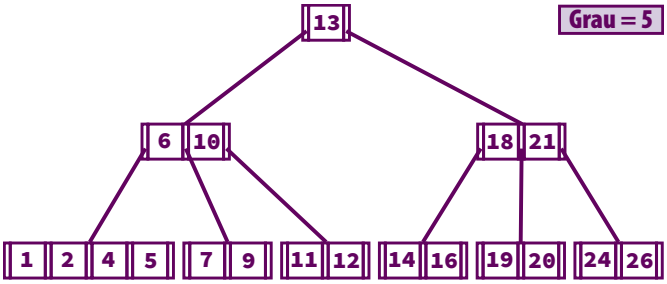


Figura 6–33: Exemplo de Remoção em Árvore B 4

6.4.5 Implementação

Tipos e Constantes

As mesmas definições de constantes e tipos utilizadas na implementação de árvore multidirecional descendente em memória secundária, de modo que os mesmos tipos `tNoMultiMS` e `tArvoreMultiMS` (v. **Seção 6.3.1**) serão usadas na implementação de árvore B. Além desses tipos, o tipo `tNoCaminhoB`, apresentado a seguir, será usado para definir elementos de uma pilha que armazena caminhos de nós visitados durante uma operação de inserção ou remoção.

```
typedef struct {
          tNoMultiMS no;       /* O nó              */
          int        endereco; /* Seu endereço      */
          int        pos;      /* Sua posição entre seus irmãos */
       } tNoCaminhoB;
```

Note que cada nó num caminho de inserção ou remoção é armazenado na pilha. Isso não é estritamente necessário, visto que a posição de cada nó no arquivo também é armazenada. Mas armazenando-se esses nós, evitam-se possíveis acessos adicionais para leitura de nós no meio de armazenamento secundário.

Funções Auxiliares

Antes da apresentação da função que implementa o algoritmo de inserção para árvores B descrito na **Seção 6.4.3**, é necessário discutir algumas funções auxiliares que facilitam a implementação desse algoritmo.

As seguintes funções utilizadas na implementação de árvore B são idênticas às funções de mesmas denominações usadas para árvores multidirecionais descendentes implementadas em arquivo (v. **Seção 6.3**):

- `BuscaEmNoMultiMS()` — utilizada pelas funções que fazem busca e inserção em árvores B (v. **Seção 6.3.4**).
- `LeNoMultiMS()` — faz a leitura de um nó de uma árvore B armazenada em arquivo (v. **Seção 6.3.3**)
- `EscreveNoMultiMS()` — atualiza o conteúdo de um nó no arquivo (v. **Seção 6.3.3**)
- `IniciaNoMultiMS()` — cria um nó de uma árvore multidirecional armazenada em arquivo (v. **Seção 6.3.5**)

A função `EncontraCaminhoB()`, apresentada a seguir, armazena (i.e., empilha) o caminho de nós visitados desde a raiz até o nó no qual uma chave é encontrada ou no qual ela deve ser inserida. Essa função faz uso de uma pilha implementada como lista encadeada (v. exemplo completo no site deste livro). A função `EncontraCaminhoB()` é semelhante à função `EncontraNoMultiMS()` apresentada na **Seção 6.3.5**, mas agora o caminho da raiz até o nó encontrado é empilhado. Os parâmetros da função `EncontraCaminhoB()` são:

- `stream` (entrada) — stream associado ao arquivo que contém a árvore
- `chave` (entrada) — a chave de busca
- `raiz` (entrada) — posição no arquivo da raiz da árvore
- `posicaoDaChaveNoNo` (saída) — ponteiro para uma variável que conterá o índice da chave no nó
- `encontrado` (saída) — ponteiro para uma variável que indicará se a chave foi encontrada
- `*pilha` (saída) — pilha que armazenará o caminho de nós visitados

```
static void EncontraCaminhoB(FILE *stream, tChave chave, int raiz,
                             int *posicaoDaChaveNoNo, int *encontrado, tPilha *pilha)
{
   int         p,  /* Conterá a posição desejada */
               indiceNo; /* Índice de um nó entre seus irmãos */
   tNoMultiMS  umNo;
   tNoCaminhoB caminho;

      /* Verifica se o stream é NULL */
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, EncontraCaminhoB));

      /* Tenta mover o apontador de arquivo para seu início */
  MoveApontador(stream, 0, SEEK_SET);

   p = raiz; /* Começa busca na raiz da árvore */
   indiceNo = -1;  /* A raiz não tem irmãos */
   *encontrado = 0;  /* A chave ainda não foi encontrada */
   CriaPilha(pilha); /* Inicia a pilha */

      /* Desce na árvore até encontrar a chave de busca ou */
      /* atingir uma folha na qual a chave não se encontra */
   while (p != POSICAO_NULA) {
         /* Lê um nó da árvore na posição p do arquivo */
      LeNoMultiMS(stream, p, &umNo);

      /***                                ***/
      /*** Cria o item que será empilhado ***/
      /***                                ***/

      caminho.no = umNo; /* Armazena o nó no item da pilha */
      caminho.endereco = p; /* Armazena a posição do nó */
      caminho.pos = indiceNo; /* Armazena seu índice entre irmãos */

      Empilha(caminho, pilha); /* Faz o empilhamento */

         /* Tenta encontrar a chave no nó corrente */
      indiceNo = BuscaEmNoMultiMS(chave, &umNo);
```

```
         /* Verifica se a chave foi encontrada */
      if(( indiceNo < umNo.nFilhos - 1) && chave == umNo.chaves[indiceNo].chave) {
         *encontrado = 1;  /* A chave foi encontrada */
         *posicaoDaChaveNoNo = indiceNo;

         return;
      }

         /* A chave ainda não foi encontrada. Desce um nível na árvore */
      p = umNo.filhos[indiceNo];
   }

      /* A chave não foi encontrada. 'posicaoDaChaveNoNo' conterá o índice no */
      /* último nó visitado (folha) no qual a chave deverá ser inserida       */
   *posicaoDaChaveNoNo = indiceNo;
}
```

A função `InsereEmNoB()`, apresentada a seguir, insere uma chave e seu respectivo filho direito num nó incompleto de uma árvore B. Essa função é parecida com a função `InsereEmFolhaMultiMS()` utilizada em inserção em árvores multidirecionais descendentes. No caso de árvores B, contudo, inserções não ocorrem apenas em folhas. Os parâmetros dessa função são:

- `*pNo` (entrada/saída) — nó no qual será feita a inserção
- `pos` (entrada) — posição no array de chaves do nó na qual a chave será inserida
- `chaveEIndice` (entrada) — par chave/índice que será inserido
- `pFilho` (entrada) — ponteiro para o filho a ser inserido à direita do novo par

```
static void InsereEmNoB(tNoMultiMS *pNo, int pos, tChaveIndice chaveEIndice, int pFilho)
{
   int i;

      /* Verifica se o nó é válido */
   ASSEGURA( pNo, "Tentativa de insercao em no' nulo" );

      /* Verifica se a posição de inserção é válida */
   ASSEGURA(pos <= pNo->nFilhos, ERRO_POSICAO(InsereEmNoB));

      /* Abre espaço para a nova chave */
   for (i = pNo->nFilhos - 1; i > pos; --i){
      pNo->filhos[i + 1] = pNo->filhos[i];
      pNo->chaves[i] = pNo->chaves[i - 1];
   }

      /* Insere o novo par chave/índice e seu filho direito */
   pNo->chaves[pos] = chaveEIndice; /* Insere o par */
   pNo->filhos[pos + 1] = pFilho; /* Insere seu filho direito */

      /* O número de filhos (e de chaves) foi acrescido de um */
   pNo->nFilhos++;
}
```

A função `CopiaChavesB()`, apresentada a seguir, é utilizada para copiar chaves e filhos de um nó para outro. Essa função é útil no processo de divisão de nós, que é fundamental no processo de inserção em árvores B. Os parâmetros dessa função são:

- `pNo1` (entrada) — ponteiro para o nó cujas chaves serão copiadas
- `*pNo2` (saída) — nó que receberá as chaves copiadas
- `indiceI` (entrada) — índice da primeira chave a ser copiada
- `indiceF` (entrada) — índice da última chave a ser copiada

```
static void CopiaChavesB( const tNoMultiMS *pNo1, tNoMultiMS *pNo2,
                          int indiceI, int indiceF )
{
   int i,
       nChaves; /* Número de chaves a copiar. Essa variável */
                /* não é essencial, mas facilita a operação */

   nChaves = indiceF - indiceI + 1;

      /* Verifica se o número de chaves é válido */
   ASSEGURA( nChaves < G, "Chaves demais para copiar" );

      /* Idem */
   ASSEGURA( nChaves > 0,  "ERRO: Nao ha' chaves para copiar" );

      /* Copia chaves e filhos do nó apontado por */
      /* pNo1 para o nó apontado por pNo2         */
   for (i = 0; i < nChaves; ++i) {
      pNo2->chaves[i] = pNo1->chaves[indiceI + i];
      pNo2->filhos[i] = pNo1->filhos[indiceI + i];
   }

      /* Faltou copiar o último filho */
   pNo2->filhos[nChaves] = pNo1->filhos[indiceI + nChaves];

      /* Atualiza o número de chaves do nó que recebeu a doação */
   pNo2->nFilhos = nChaves + 1;
}
```

A função `DivideNoB()`, apresentada a seguir, implementa o processo de divisão de nós e é fundamental no processo de inserção em árvores B. Os parâmetros dessa função são:

- `*pNoDiv` (entrada/saída) — nó que será dividido
- `pos` (entrada) — posição onde a nova chave será inserida se ela for menor do que a chave intermediária
- `chaveEIndice` (entrada) — a chave (e seu respectivo índice) que será inserida num dos nós resultantes da divisão ou num nó que fica num nível acima
- `pFilhoDireita` (entrada) — posição no arquivo do nó que será o filho à direita da chave inserida
- `*pNoDireita` (saída) — posição no arquivo do novo nó no qual serão colocadas as chaves superiores do nó dividido
- `*chaveMeio` (saída) — a chave do meio (i.e., aquela que subirá para o nó-pai do nó dividido)
- `stream` (entrada) — stream associado ao arquivo que contém a árvore

```
static void DivideNoB( tNoMultiMS *pNoDiv, int pos,
                       tChaveIndice chave, int pFilhoDireita,
                       int *pNoDireita, tChaveIndice *chaveMeio, FILE *stream )
{
   static const int meio = G/2; /* Evita recalcular esse valor */
                                /* a cada chamada desta função */
   tNoMultiMS       noDireita; /* Novo nó que será acrescentado à árvore */

      /* Inicia o novo nó que receberá as chaves superiores */
   IniciaNoMultiMS(NULL, &noDireita);

      /* Efetua a inserção com a devida divisão de nós */
   if (pos > meio) { /* Caso 1: A nova chave pertence ao novo nó da direita */
         /* Copia metade menos uma chave do nó dividido para o novo nó */
      CopiaChavesB(pNoDiv, &noDireita, meio + 1, G - 2);

         /* Insere a nova chave no novo nó da direita */
      InsereEmNoB( &noDireita, pos - meio - 1, chave, pFilhoDireita );
```

```
      pNoDiv->nFilhos = meio + 1; /* Ajusta o número de nós do nó dividido */

         /* A chave que subirá é a última chave do */
         /* nó que foi dividido depois da divisão  */
      *chaveMeio = pNoDiv->chaves[meio];
   } else if (pos == meio) { /* Caso 2: A nova chave é aquela que subirá um nível */
         /* Copia metade das chaves do nó dividido para o novo nó */
      CopiaChavesB(pNoDiv, &noDireita, meio, G - 2);

      pNoDiv->nFilhos = meio + 1; /* Ajusta o número de nós do nó dividido */

         /* O filho direito da nova chave passa */
         /* a ser o primeiro filho do novo nó   */
      noDireita.filhos[0] = pFilhoDireita;

      *chaveMeio = chave; /* A chave que subirá é a nova chave */
   } else { /* Caso 3: A nova chave pertence ao nó que será dividido */
         /* Copia metade das chaves do nó dividido para o novo nó */
      CopiaChavesB(pNoDiv, &noDireita, meio, G - 2);

         /* Ajusta o número de nós do nó dividido */
      pNoDiv->nFilhos = meio + 1;

         /* Insere a nova chave no nó que foi dividido */
      InsereEmNoB(pNoDiv, pos, chave, pFilhoDireita);

         /* A chave que subirá é a última chave do */
         /* nó que foi dividido depois da divisão  */
      *chaveMeio = pNoDiv->chaves[meio];

         /* Corrige o número de filhos do nó dividido, pois    */
         /* sua última chave subirá para o próximo nível acima */
      --pNoDiv->nFilhos;
   }

      /* O novo nó será armazenado ao final do arquivo; */
      /* portanto sua posição no arquivo será igual ao  */
      /* número de registros (nós) do arquivo (NB: a    */
      /* indexação do arquivo começa em zero; por isso, */
      /* não se acrescenta 1 a esse valor).             */
   *pNoDireita = NumeroDeNosB(stream);

   EscreveNoMultiMS(stream, *pNoDireita, &noDireita); /* Atualiza nó no arquivo */
}
```

A função `DivideNoB()` insere uma chave num nó completo, criando um novo nó para a metade superior das chaves e mantendo a metade inferior das chaves no nó dividido. A chave intermediária é guardada para posterior inserção num nó que fica num nível acima. Observe que, se o grau da árvore for par, a árvore terá tendência à esquerda; i.e., o nó dividido terá uma chave a mais do que o novo nó. É importante notar ainda que o nó apontado por `pNoDiv` deve ser atualizado no arquivo que contém a árvore pela função que chama a presente função.

As funções auxiliares apresentadas até aqui são aquelas utilizadas em busca e inserção em árvores B. As funções auxiliares usadas em remoção serão apresentadas oportunamente mais adiante.

Busca

A função `BuscaB()`, que implementa o processo de busca em árvores B armazenadas em arquivo é essencialmente igual àquela que faz o mesmo para árvores multidirecionais descendentes. Por essa razão, a função não será apresentada aqui.

Inserção

A função `InsereB()`, apresentada a seguir, é a função principal do processo de inserção em árvores B. Essa função insere um par chave/índice numa árvore B armazenada em arquivo e seus parâmetros são:

- `*posicaoRaiz` (entrada/saída) — posição da raiz da árvore no arquivo
- `chaveEIndice` (entrada/saída) — ponteiro para um par chave/índice que será encontrado ou inserido
- `streamArvore` (entrada) — stream associado ao arquivo que contém a árvore

A função `InsereB()` retorna `1`, se houver inserção, ou `0`, em caso contrário.

```
int InsereB( int *posicaoRaiz, tChaveIndice *chaveEIndice, FILE *streamArvore )
{
   tNoMultiMS    no; /* Novo nó que poderá ser inserido */
   int           encontrado, /* Indicará se a chave foi encontrada */
                 indiceDaChave, /* Índice da chave no nó onde     */
                                /* ela for encontrada ou inserida */
                 pFilhoDireito,
                 pNoDireito;
   tPilha        pilha; /* Pilha que armazenará o caminho de nós visitados */
   tNoCaminhoB   caminho; /* Armazenará cada item desempilhado */
   tChaveIndice  chaveAInserir, chaveQueSobe;

      /* O stream que representa o arquivo     */
      /* que contém a árvore não pode ser NULL */
   ASSEGURA( streamArvore, ERRO_STREAM_NULL(streamArvore, InsereB) );

      /* Verifica se a árvore já existe */
   if (*posicaoRaiz == POSICAO_NULA) {
      /* A árvore ainda não foi criada */

         /* Cria a raiz  */
      IniciaNoMultiMS(chaveEIndice, &no);
      indiceDaChave = 0;
      *posicaoRaiz = 0;

         /* Armazena a raiz da árvore na posição 0 do arquivo */
      EscreveNoMultiMS(streamArvore, 0, &no);

      return 1;
   }

         /* Tenta encontrar o nó que contém a chave e o caminho que leva até ele */
   EncontraCaminhoB( streamArvore, chaveEIndice->chave, *posicaoRaiz,
                     &indiceDaChave, &encontrado, &pilha );

      /* Verifica se a chave foi encontrada */
   if (encontrado) {
         /* A chave foi encontrada. Obtém o valor do par   */
         /* chave/índice que se encontra no topo da pilha. */
      caminho = Desempilha(&pilha);

         /* Atualiza-se o índice do par 'chaveEIndice', */
         /* cujo endereço foi recebido  como parâmetro. */
      chaveEIndice->indice = caminho.no.chaves[indiceDaChave].indice;

      EsvaziaPilha(&pilha); /* Antes de retornar, esvazia-se a pilha */

      return 0; /* Não houve inserção */
   }

      /* A diferença significativa entre esta função e a função correspondente */
      /* para árvores multidirecionais descendentes começa aqui                */
```

```
      /* A inserção começa numa folha. Assim o filho */
      /* à direita da chave a ser inserida é nulo    */
   pFilhoDireito = POSICAO_NULA;

      /* A primeira chave a ser inserida é aquela recebida como parâmetro */
   chaveAInserir = *chaveEIndice;

      /* Percorre o caminho inverso até encontrar */
      /* a raiz ou um nó incompleto para inserção */
   while (!PilhaVazia(pilha)) {
      caminho = Desempilha(&pilha); /* Obtém informações sobre o nó corrente */

         /* Verifica se é possível inserir neste nó */
      if (caminho.no.nFilhos < G) {
            /* Há espaço para inserção neste nó */
         InsereEmNoB(&caminho.no, indiceDaChave, chaveAInserir, pFilhoDireito);

            /* Atualiza o nó na árvore */
         EscreveNoMultiMS(streamArvore, caminho.endereco, &caminho.no);

            /* O conteúdo da pilha não é mais necessário */
         EsvaziaPilha(&pilha);

         return 1; /* Serviço concluído */
      }

         /* O nó corrente é completo e precisa ser dividido */
      DivideNoB( &caminho.no, indiceDaChave, chaveAInserir,
                 pFilhoDireito, &pNoDireito, &chaveQueSobe, streamArvore );

         /* Atualiza o nó na árvore */
      EscreveNoMultiMS(streamArvore, caminho.endereco, &caminho.no);

         /* O filho à direita da chave que subirá para o nível */
         /* superior será o novo nó criado no nível atual      */
      pFilhoDireito = pNoDireito;
      indiceDaChave = caminho.pos;
      chaveAInserir = chaveQueSobe;
   }

   /* Neste ponto, sabe-se que o último caminho desempilhado continha uma  */
   /* raiz completa, que já foi dividida no laço while. Resta criar uma    */
   /* nova raiz contendo a chave que deve subir um nível tendo como filho  */
   /* direito o nó dividido e à esquerda a antiga raiz após a divisão      */

   IniciaNoMultiMS(&chaveQueSobe, &no); /* Cria a nova raiz */

      /* O filho da esquerda da nova raiz é o nó  */
      /* que restou da raiz antiga após a divisão */
   no.filhos[0] = caminho.endereco;

      /* O filho da direita da nova raiz é o novo nó resultante da divisão */
   no.filhos[1] = pNoDireito;

      /* A nova raiz será acrescentada ao final do arquivo que armazena a */
      /* árvore. Portanto sua posição é igual ao número corrente de nós.  */
   *posicaoRaiz = NumeroDeNosB(streamArvore);

   EscreveNoMultiMS(streamArvore, *posicaoRaiz, &no); /* Atualiza raiz no arquivo */

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

A função `IniciaNoMultiMS()` chamada por `InsereB()` foi definida na **Seção 6.3.5**.

Remoção

A função `RemoveChaveB()` remove a chave especificada de uma árvore B armazenada em arquivo. Os parâmetros dessa função são:

- `*streamArvore` (entrada) — stream associado ao arquivo que armazena a árvore
- `*raiz` (entrada/saída) — posição no arquivo da raiz da árvore
- `chave` (entrada) — a chave que será removida

A função `RemoveChaveB()` retorna `1`, se a remoção for bem-sucedida, ou `0`, em caso contrário. O algoritmo seguido por essa função é aquele apresentado na **Figura 6–19**.

```
int RemoveChaveB(FILE *streamArvore, int *raiz, tChave chave)
{
   int          encontrado, /* A chave foi encontrada? */
                indiceDaChave, /* Índice da chave no nó no qual */
                               /* ela está ou deveria estar     */
                iEntreIrmaos, /* O índice de um nó entre seus irmãos */
                p, /* Posição de um nó no arquivo que armazena a árvore */
                pNoDaRemocao; /* Posição em arquivo do nó que */
                              /* contém a chave a ser removida */
   tPilha       pilha, /* Pilha que armazenará o caminho de nós visitados */
                pilhaAux; /* Uma pilha auxiliar */
   tNoCaminhoB  caminho; /* Um elemento da pilha que representa o caminho de nós */
   tNoMultiMS   no; /* Um nó da árvore */

      /* Se a árvore for vazia, não há chave para remover */
   if (*raiz == POSICAO_NULA)
      return 0; /* Não há chave para remover */

      /* Encontra o caminho da raiz até o nó */
      /* que contém a chave a ser removida   */
   EncontraCaminhoB(streamArvore, chave, *raiz, &indiceDaChave, &encontrado, &pilha);

      /* Se a chave não for encontrada, não há mais nada a */
      /* fazer. Mas antes de retornar, esvazia-se a pilha. */
   if (!encontrado) {
      EsvaziaPilha(&pilha);

      return 0; /* Não há chave para remover */
   }

   /*** A chave foi encontrada e informações sobre ***/
   /*** o nó que a contém estão no topo da pilha   ***/

      /* Obtém o elemento no topo da pilha sem desempilhá-lo */
   caminho = ElementoTopo(pilha);

      /* Verifica se o nó no qual será feita a remoção é uma folha */
   if (caminho.no.filhos[0] == POSICAO_NULA) {
         /* O nó é uma folha. A função RemoveEmFolhaB() completa o serviço */
      RemoveEmFolhaB(&pilha, streamArvore);

      return 1; /* Serviço concluído */
   }

   /* A chave a ser removida não se encontra numa folha. É necessário substituí    */
   /* -la por sua sucessora e, depois, remover esta sucessora. Em seguida, será    */
   /* feito um caminhamento, empilhando os nós no caminho que leva até a sucessora */

      /* Guarda a posição em arquivo do nó que contém a chave */
```

```
   pNoDaRemocao = caminho.endereco;

      /*** Será feita uma descida na árvore usando ***/
      /*** p para indicar um nó no nível corrente  ***/

      /* Atribui a p a posição do filho direito da chave a ser removida */
   p = caminho.no.filhos[indiceDaChave + 1];
   iEntreIrmaos = indiceDaChave + 1;

   ASSEGURA(p != POSICAO_NULA, "p nao deveria ser POSICAO_NULA em RemoveChaveB()");

      /* Segue descendo até encontrar uma subárvore esquerda vazia */
   while (p != POSICAO_NULA) {
      LeNoMultiMS(streamArvore, p, &no); /* Lê o nó na posição indicada por p */

         /* Armazena num item da pilha o nó mais recentemente visitado, */
         /* sua posição no arquivo da árvore e seu índice entre irmãos  */
      caminho.no = no;
      caminho.endereco = p;
      caminho.pos = iEntreIrmaos;

      Empilha(caminho, &pilha); /* Faz o empilhamento */

      p = no.filhos[0]; /* Desce mais um nível à esquerda da primeira chave */

      iEntreIrmaos = 0; /* A primeira chave tem índice zero */
   }

   /* A chave sucessora da chave a ser removida é a primeira chave do    */
   /* último nó lido. Essa chave sucessora deve substituir a chave a     */
   /* ser removida. Portanto é preciso substituir a chave a ser removida */
   /* no nó que se encontra na pilha.                                    */

      /* Cria uma pilha auxiliar para guardar os nó */
      /* que se encontram sobre o nó a ser alterado */
   CriaPilha(&pilhaAux);

   do { /* Desempilha até encontrar o nó que será alterado */
      caminho = Desempilha(&pilha);
      Empilha(caminho, &pilhaAux);
   } while (caminho.endereco != pNoDaRemocao);

      /* Substitui a chave a ser removida por sua sucessora */
   caminho.no.chaves[indiceDaChave] = no.chaves[0];

      /* Atualiza nó no arquivo */
   EscreveNoMultiMS(streamArvore, caminho.endereco, &caminho.no);

      /* Reconstrói a pilha */
   while (!PilhaVazia(pilhaAux)) {
      caminho = Desempilha(&pilhaAux);
      Empilha(caminho, &pilha);
   }

      /* Resta remover a chave sucessora na folha que se encontra no topo da pilha */
   RemoveEmFolhaB(&pilha, streamArvore);

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

A função `RemoveChaveB()` chama as seguintes funções auxiliares:

- `CompactaNoMultiMS()`, que compacta um nó de uma árvore B após uma remoção. Essa função é idêntica àquela com a mesma denominação apresentada na **Seção 6.3.6**.

- `RemoveEmFolhaB()`, que remove uma chave de uma folha de uma árvore B (v. adiante).

As demais funções chamadas por `RemoveChaveB()` representam operações básicas sobre pilhas: `EsvaziaPilha()`, `ElementoTopo()`, `Empilha()`, `CriaPilha()`, `Desempilha()`, `PilhaVazia()`. Espera-se que leitor não tenha dúvidas com relação a cada uma dessas operações (caso contrário, aconselha-se a consultar o **Volume 1** desta obra).

A função `RemoveEmFolhaB()` remove uma chave de uma folha de uma árvore B e seus parâmetros são:

- `*pilha` (entrada/saída) — pilha que armazena os nós encontrados no caminho da raiz até a folha (incluindo-a)
- `stream` (entrada) — stream associado ao arquivo que contém a árvore

```
static void RemoveEmFolhaB(tPilha *pilha, FILE *stream)
{
   tNoCaminhoB caminho; /* Um elemento da pilha que representa o caminho de nós */
   int         i;

   ASSEGURA( !PilhaVazia(*pilha), "ERRO: Pilha vazia em RemoveEmFolhaB()" );

      /* No topo da pilha, encontra-se a folha na qual a remoção será efetuada */
   caminho = Desempilha(pilha);

         /* Move as chaves */
   for (i = caminho.pos; i < caminho.no.nFilhos - 2; ++i)
      caminho.no.chaves[i] = caminho.no.chaves[i + 1];

   caminho.no.nFilhos--; /* O número de filhos da folha foi reduzido */

      /* Chama JuntaNosB() para completar o serviço */
   JuntaNosB(&caminho.no, pilha, stream);

      /* Atualiza o nó na árvore */
   EscreveNoMultiMS(stream, caminho.endereco, &caminho.no);

      /* O conteúdo da pilha não é mais necessário */
   EsvaziaPilha(pilha);
}
```

A função `JuntaNosB()` chamada por `RemoveEmFolhaB()` combina um nó com outros nós de uma árvore B quando ele fica com um número de chaves abaixo do mínimo permitido. Essa função é longa demais para ser apresentada neste livro. O leitor poderá encontrá-la no site dedicado ao livro na internet.

6.4.6 Persistência de Dados

Esta seção discute funções que garantem a **persistência dos dados** armazenados numa árvore B entre seções de execução de um programa-cliente. Quer dizer, as funções que serão discutidas aqui permitem que uma árvore B criada numa seção de execução de um programa-cliente possa continuar sendo usada em seções de execução subsequentes do mesmo programa sem que a árvore precise ser reconstruída a cada seção. Essa persistência de dados garantida por essas funções é importante porque, tipicamente, a construção de uma árvore B utilizada como tabela de busca para recuperação de dados num arquivo contendo milhões de registros leva um tempo considerável.

A função `LeRaizB()` lê num arquivo binário a posição da raiz de uma árvore B armazenada em arquivo. Seu único parâmetro é o nome do arquivo que armazena a raiz da árvore e ela retorna o valor lido no arquivo.

```
int LeRaizB(const char *nomeArqRaiz)
{
   FILE *stream;
   int   raiz;

      /* Tenta abrir o arquivo que contém a raiz da árvore para leitura apenas */
```

```
   stream = AbreArquivo(nomeArqRaiz, "rb");
   fread(&raiz, sizeof(raiz), 1, stream);
      /* Certifica-se que não ocorreu erro de leitura */
   ASSEGURA(!ferror(stream), ERRO_FREAD(LeRaizB));
   FechaArquivo(stream, nomeArqRaiz); /* O arquivo não precisa mais estar aberto */
   return raiz;
}
```

A função `EscreveRaizB()` escreve num arquivo binário a posição da raiz de uma árvore B armazenada em arquivo e seus parâmetros são:

- `nomeArqRaiz` — string que representa o nome do arquivo que armazena a raiz da árvore
- `raiz` — índice do nó que representa a raiz no arquivo que representa a árvore

```
void EscreveRaizB(const char *nomeArqRaiz, int raiz)
{
   FILE *stream;
      /* Tenta abrir o arquivo que contém a raiz da árvore para escrita apenas */
   stream = AbreArquivo(nomeArqRaiz, "wb");
   fwrite(&raiz, sizeof(raiz), 1, stream);
      /* Certifica-se que não ocorreu erro de escrita */
   ASSEGURA(!ferror(stream), ERRO_FWRITE(EscreveRaizB));
   FechaArquivo(stream, nomeArqRaiz); /* O arquivo não precisa mais estar aberto */
}
```

6.4.7 Programa-cliente

Um programa-cliente que use a implementação de árvore B apresentada acima deve incluir funções que verifiquem se a árvore B com a qual ele espera contar existe no local (i.e., diretório) esperado. Se esse for o caso, ele pode instar o usuário a decidir se ele pretende usar a árvore já existente ou criar uma nova árvore. Se o programa não encontra a árvore no local esperado ou se o usuário deseja construir uma nova árvore o programa lê o arquivo de registro e constrói a devida árvore. Um programa que assim procede e oferece as operações básicas de uma tabela de busca pode ser encontrado no site dedicado ao livro na internet.

6.4.8 Análise

Como foi visto acima, o grau de uma árvore B depende do tamanho da chave e do bloco. Nesta seção, será mostrado que a altura de uma árvore B será bem pequena quando o grau da árvore for suficientemente grande.

O **grau mínimo** d de uma árvore multidirecional é o menor grau permitido para cada nó dessa árvore. No caso de árvores B, o grau mínimo é a metade do grau da árvore (i.e., $d = \lceil G/2 \rceil$).

Teorema 6.1: Sejam a a altura, d o grau mínimo e n o número de chaves de uma árvore B. Então:

$$a \leq \left\lfloor \log_d \left(\frac{n+1}{2} \right) \right\rfloor + 1$$

Prova: A árvore B de maior altura é obtida quando cada nó possui o menor grau permitido. Ou seja, quando a raiz da árvore possui dois filhos e os demais nós possuem d filhos, como mostra a **Figura 6–34**. Na situação exposta nessa figura, o número total de nós da árvore B é dado por:

Número de chaves no primeiro nível: 1 +
Número de chaves no segundo nível: $2(d-1)$ +

Número de chaves no terceiro nível: $2d(d-1)$ +
Número de chaves no quarto nível: $2d^2(d-1)$ +
⋮ ⋮ ⋮

Usando esse raciocínio, obtém-se que o número total de chaves no pior caso de uma árvore B é dado por:

$$n_{pior\ caso} = 1 + \left(\sum_{i=0}^{a-2} 2d^i\right)(d-1) = 1 + 2 \cdot \frac{1-d^{a-1}}{1-d} \cdot (d-1) = 2d^{a-1} - 1$$

O último resultado foi obtido aplicando-se a fórmula de soma de elementos de uma progressão geométrica e fatorando-se a expressão resultante da aplicação dessa fórmula. Diante desse resultado, a relação entre o número de chaves em qualquer árvore B e sua altura pode ser expressa como:

$$n \geq -1 + 2d^{a-1}$$

Resolvendo-se essa inequação para a altura a, obtém-se:

$$a \leq \log_d \left(\frac{n+1}{2}\right) + 1$$

■

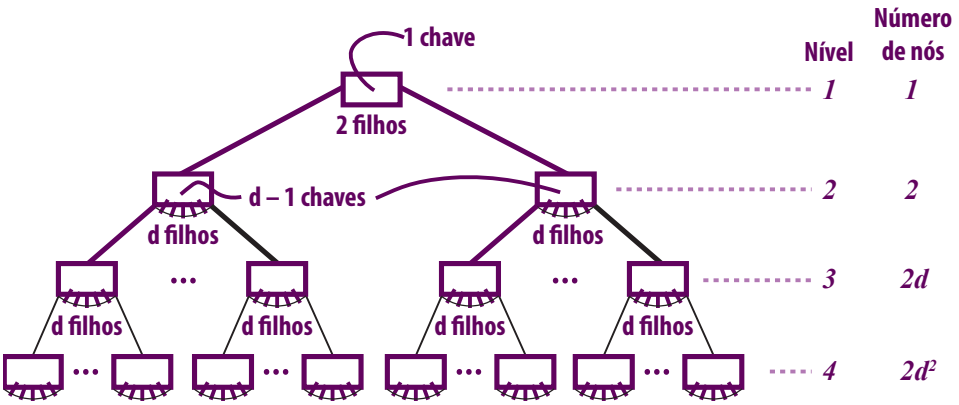


Figura 6–34: Altura Máxima de uma Árvore B

Corolário 6.1: No nível a de uma árvore B, há, pelo menos, $2d^{a-1} - 1$ chaves.

Prova: V. prova do **Teorema 6.1**. ■

O pior caso de busca numa árvore B ocorre quando ela possui o menor número de filhos permitidos em cada nó. Isto é, quando ela possui dois filhos na raiz e $d = \lceil G/2 \rceil$ filhos em cada um dos demais nós, sendo G o grau da árvore. Suponha que, como foi visto no exemplo da **Seção 6.3.1**, o grau de uma árvore B seja *340*, de modo que seu grau mínimo seja *170*. Então de acordo com o **Corolário 6.1**, ela terá no nível *3*, pelo menos:

$$2 \cdot d^{a-1} - 1 \text{ chaves} = 2 \cdot 170^2 - 1 \cong 57.800 \text{ chaves}$$

Por sua vez, no nível *4*, essa mesma árvore poderia conter, no mínimo, *9.826.000* de chaves.

Um fator de otimização para tabelas de busca implementadas como árvores B consiste em deixar a raiz sempre armazenada em memória principal, pois como qualquer operação com árvore B começa sempre na raiz da árvore, economiza-se um acesso ao disco sempre que for necessário efetuar uma delas.

Considerando o exemplo anterior, podem-se realizar operações típicas sobre essa árvore B com capacidade de armazenar quase 10 milhões de chaves com, no máximo, apenas três operações de leitura ou escrita. Evidentemente, na primeira dessas operações, a raiz da árvore deve ser lida. Também, se a raiz for alterada, ela deve ser reescrita no meio de armazenamento.

De acordo com o **Teorema 6.1**, para um valor de G (o grau da árvore) suficientemente grande, a altura de uma árvore B é pequena mesmo quando se armazena nela um grande número de chaves. Por exemplo, se G for igual a *200* (e, portanto, d for igual a *100*) e o número de chaves for igual a *100.000.000* (cem milhões), tem-se que $a \leq 4$, de modo que, na pior situação, encontrar uma chave nessa árvore requer no máximo quatro acessos ao disco. Melhor ainda, se a raiz da árvore for mantida sempre em memória principal, esse número de acessos é reduzido para três.

Teorema 6.2: Qualquer operação de busca, inserção ou remoção em árvores B com grau mínimo d e n chaves requer menos do que $log_d\, n$ acessos à memória secundária.

Prova: No caso de busca, a prova é trivial (v. abaixo). Inserção numa árvore B requer a execução dos seguintes passos:

1. Uma operação de busca pelo local de inserção que, de acordo com o **Teorema 6.1**, requer no máximo

$$\log_d\left(\frac{n+1}{2}\right)+1$$

operações de leitura em disco.

2. Inserção da chave no devido nó eventualmente seguida por divisões de nós que, no pior caso, podem se propagar até a raiz da árvore.

No melhor caso de inserção, que é aquele no qual não ocorre nenhuma divisão, o número de acessos ao disco é acrescido de apenas uma operação de escrita em arquivo do nó modificado. Contudo, no pior caso, que é aquele em que ocorrem divisões até a raiz da árvore, o número de acessos adicionais de escrita em disco é igual ao número de nós visitados em busca pelo local de inserção.

A prova para remoção é feita de modo semelhante ao caso para inserção e é deixada como exercício. ■

Pode-se mostrar[4] que a probabilidade média de ocorrência de uma divisão de nó numa árvore B é dada por:

$$\frac{1}{\lceil G/4 \rceil - 1}$$

Esse resultado indica que, quanto maior for o grau da árvore, menor é a probabilidade de ocorrência de divisão de nós. Por exemplo, se uma árvore B tiver grau G igual a *200*, tem-se que a possibilidade de ocorrência de divisão é igual a *2%*. Ou seja, nesse caso, a cada *100* inserções espera-se que ocorram duas divisões de nós. Por consequência a propagação da divisão de nós até a raiz durante uma inserção é um evento muito raro.

A exigência de que cada nó interno tenha pelo menos $\lceil G/2 \rceil$ filhos implica que cada bloco de disco usado para suportar uma árvore B é pelo menos preenchido pela metade. Raramente, os nós de uma árvore B são completos, mas estudos analíticos e experimentais indicam que a ocupação de nós é próxima de *67%*, o que é considerado muito bom.

6.5 Árvores B+

6.5.1 Conceitos

Numa **árvore B+** todas as chaves são mantidas em folhas e outros nós contêm repetições das chaves (nem todas as chaves são repetidas). Essas folhas são conectadas e constituem uma lista encadeada. Além disso, os registros estão associados apenas às chaves contidas nas folhas, de modo que uma busca termina sempre numa folha (i.e., a busca não para se a chave for encontrada num nó que não seja folha). A principal vantagem oferecida por árvores B+ é a facilidade de se percorrerem os nós sequencialmente, que é uma das maiores deficiências de

[4] Consulte, por exemplo, *Drozdek, A., Data Structures and Algoritmos in C++* (v. **Bibliografia**).

árvores B. A lista encadeada de folhas de uma árvore B+ é denominada **conjunto sequencial**. As ligações do conjunto sequencial permitem fácil processamento sequencial. Nós internos formam um **conjunto de índices** e as folhas são encadeadas da esquerda para a direita formando um conjunto sequencial.

Os níveis superiores de uma árvore B+, que são organizados como uma árvore B, consistem apenas de um roteiro para permitir rápida localização de chaves. A **Figura 6–36** mostra a separação lógica do conjunto de índices e do conjunto sequencial.

6.5.2 Busca

Uma operação de busca numa árvore B+ começa na raiz da árvore e prossegue seguindo os filhos encontrados no conjunto de índices, como ocorre numa busca em árvore B, até encontrar uma folha. Como todas as chaves que efetivamente fazem parte de uma tabela representada por uma árvore B+ residem em folhas, operações de busca em árvores B+ diferem de buscas em árvores B no sentido de que uma busca não é encerrada quando uma chave no conjunto de índices é igual à chave de busca. Em vez disso, o filho direito dessa chave é seguido e a busca prossegue até atingir uma folha.

O algoritmo completo de busca em árvore B+ é exibido na **Figura 6–35**.

ALGORITMO BUSCAEMÁRVOREB+

ENTRADA: Uma chave de busca, a raiz de uma árvore B+ e o arquivo que contém a árvore

SAÍDA: O valor associado à chave que casa com a chave de busca ou um valor informando que a chave não foi encontrada

1. Faça com que a raiz da árvore seja o nó corrente
2. Enquanto o nó corrente for um nó interno, faça:
   - 2.1 Encontre a posição em que a chave está ou deveria estar no nó corrente
   - 2.2 Se a chave foi encontrada, faça com que o nó corrente passe a ser seu filho direito
   - 2.3 Caso contrário, se a chave for menor do que alguma chave do nó corrente, faça com que o nó corrente seja o filho esquerdo da chave que ocupa a (virtual) posição da chave de busca
   - 2.4 Caso contrário, faça com que o próximo nó corrente seja o filho diteiro da chave mais à direita do corrente nó atual
3. Se a chave for encontrada na folha atingida, retorne o valor associado a ela
4. Caso contrário, retorne um valor indicando o fracasso da operação

FIGURA 6–35: ALGORITMO DE BUSCA EM ÁRVORE B+

Quando se atinge uma folha, a chave deve estar nessa folha se ela existir na árvore. Isto é, se a chave não estiver nessa folha, pode-se concluir que ela não faz parte da tabela. Considere como exemplo a localização da chave *53* na árvore B+ da **Figura 6–36**. Nesse caso, percorre-se o caminho composto pelos nós *A*, *B* e *E*. Se for desejado percorrer sequencialmente todas as chaves dessa árvore a partir da chave `53`, basta seguir os ponteiros que ligam os nós folhas.

6.5.3 Inserção

Inserção em árvores B+ é, em muitos aspectos, semelhante à inserção em árvores B, mas o leitor precisa estar atento, pois, apesar das semelhanças, também há várias diferenças. Quando há espaço para inserção numa folha, a chave é inserida em ordem e o conjunto de índices não é alterado. Por outro lado, se a folha estiver completa, ela é dividida e uma nova folha é inserida no conjunto sequencial. Então as chaves são divididas entre a nova folha e a folha antiga e a primeira chave da nova folha é *copiada* (e não *movida*) para o nó-pai. O resto da história é semelhante ao que ocorre com inserção em árvores B.

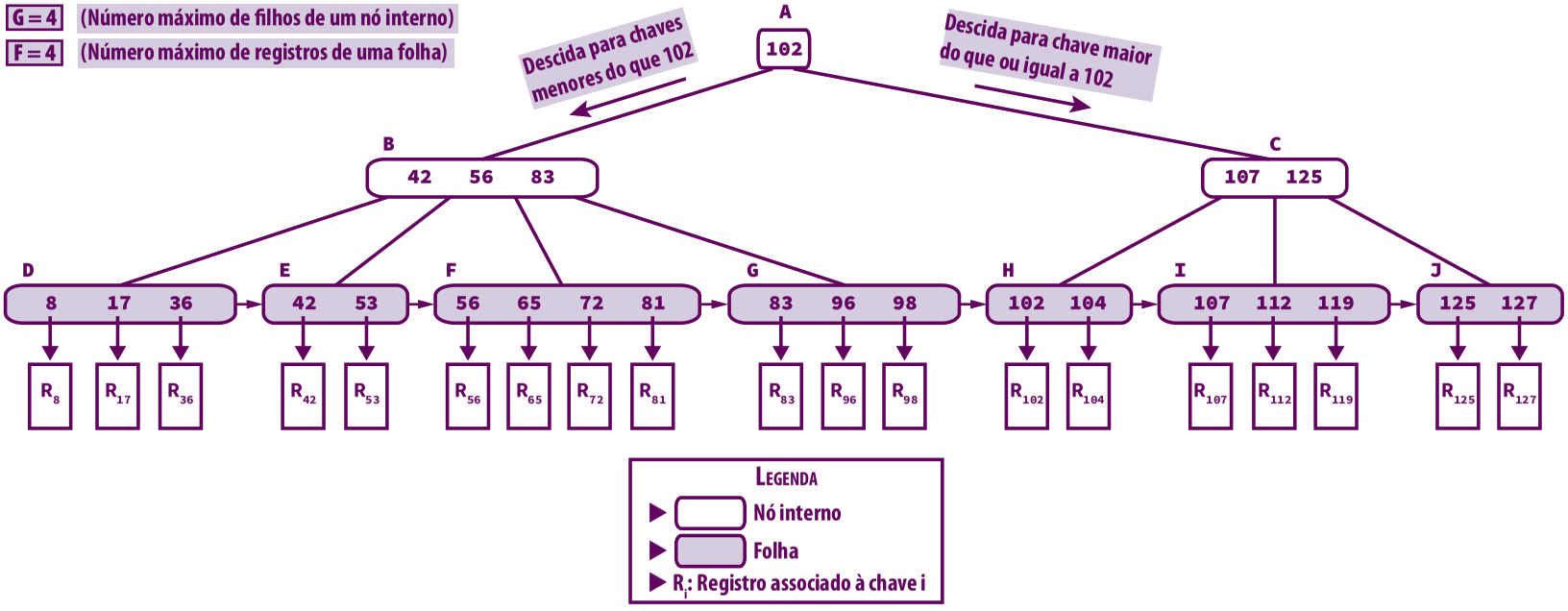


FIGURA 6–36: EXEMPLO DE ÁRVORE B+

O algoritmo de inserção em árvores B+ segue os passos descritos na **Figura 6–37**.

ALGORITMO INSEREEMÁRVOREB+

ENTRADA: A chave a ser inserida e seu valor associado
ENTRADA/SAÍDA: A árvore B+ na qual ocorrerá a inserção
SAÍDA: Um valor informando o sucesso ou o fracasso da operação

1. Se a árvore estiver vazia:
  1.1 Crie uma folha nova
  1.2 Insira a chave nessa folha
  1.3 Faça a folha apontar para uma posição que indique final de encadeamento
  1.4 Faça com que a raiz da árvore seja representada por essa folha
  1.5 Encerre informando o sucesso da operação
2. Faça uma busca usando a chave a ser inserida empilhando os nós encontrados no caminho até a folha na qual a chave será inserida
3. Se a chave for encontrada, encerre a operação informando o fracasso da operação (pois se supõe que a chave seja primária)
4. Se a folha na qual a chave será inserida não estiver completa, insira a chave nesse nó e encerre informando o sucesso da operação
5. A folha (doravante denominada F_E) na qual a chave será inserida está completa
6. Crie uma nova folha (doravante denominada F_D)
7. Copie metade das chaves maiores da folha F_E para a folha F_D
8. Faça a folha F_D apontar para a folha para a qual F_E aponta
9. Faça a folha F_E apontar para a folha F_D
10. Atribua à variável *chave* a primeira chave da folha F_D
11. Atribua à variável *posição* a posição da folha F_D em arquivo
12. Enquanto a pilha não estiver vazia, faça o seguinte:
  12.1 Desempilhe um nó interno *I*
  12.2 Se o nó *I* não estiver cheio:
    12.2.1 Insira *chave* tendo *posição* como seu filho direito em seu devido lugar no nó *I*
    12.2.2 Encerre informando o sucesso da operação
  12.3 Caso contrário, se o nó *I* estiver cheio
    12.3.1 Divida o nó *I* usando o algoritmo **DIVIDENÓINTERNOB+**
    12.3.2 Atualize os valores das variáveis *chave* e *posição*
13. Crie uma nova raiz contendo o valor da variável *chave* como chave, e tendo como filho esquerdo o último nó dividido e como filho direito o novo nó resultante da divisão
14. Retorne informando o sucesso da operação

FIGURA 6–37: ALGORITMO DE INSERÇÃO EM ÁRVORE B+

O algoritmo **DIVIDENÓINTERNOB+** que complementa o algoritmo de inserção em árvores B+ é semelhante ao algoritmo de divisão de nós para árvores B apresentado na **Figura 6–18** (v. **Seção 6.4.3**).

Nos exemplos a seguir, o número máximo de filhos de um nó interno *G* é *4* e o número máximo de chaves *F* numa folha também é *4*. Portanto o número mínimo de chaves num nó interno é *1* e o número mínimo de chaves numa folha é *2*. Para simplificar, o encadeamento das folhas é omitido das figuras.

A **Figura 6–38** mostra a criação do primeiro nó interno de uma árvore B+ após a inserção de uma chave que causa a divisão da única folha da árvore.

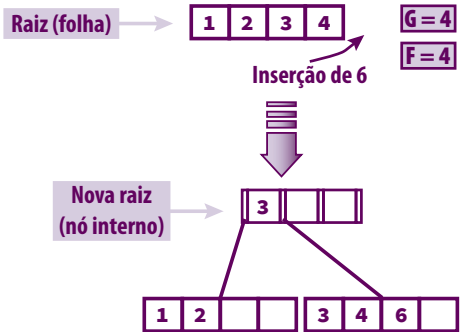


Figura 6–38: Exemplo de Inserção em Árvore B+ 1

A **Figura 6–39** mostra um caso simples de inserção que não requer divisão de nós. Por sua vez, a **Figura 6–40** apresenta um exemplo de inserção que requer uma divisão de folha, mas não requer divisão de nó interno. Ao final desse passo, a folha F_E estará apontando para a folha F_D que, por sua vez, estará apontando para a folha F_I. Esse encadeamento de folhas não é ilustrado na referida figura.

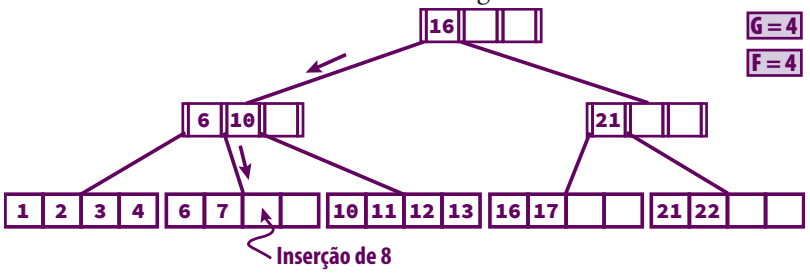


Figura 6–39: Exemplo de Inserção em Árvore B+ 2

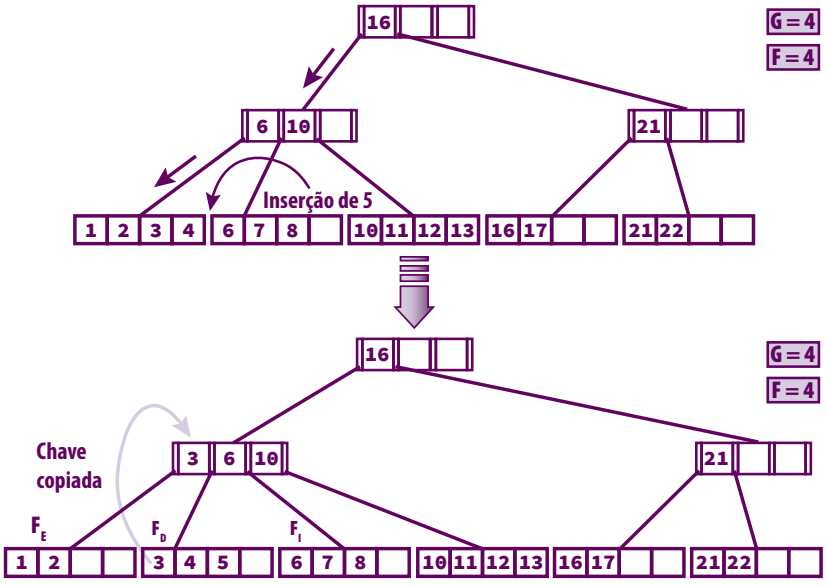


Figura 6–40: Exemplo de Inserção em Árvore B+ 3

A **Figura 6–41** mostra a inserção da chave *14* na árvore B+ resultante da **Figura 6–40**. Como exercício, o leitor é convidado a explicar em detalhes como a árvore resultante dessa inserção é obtida à luz dos algoritmos apresentados na **Figura 6–37** e na **Figura 6–18**.

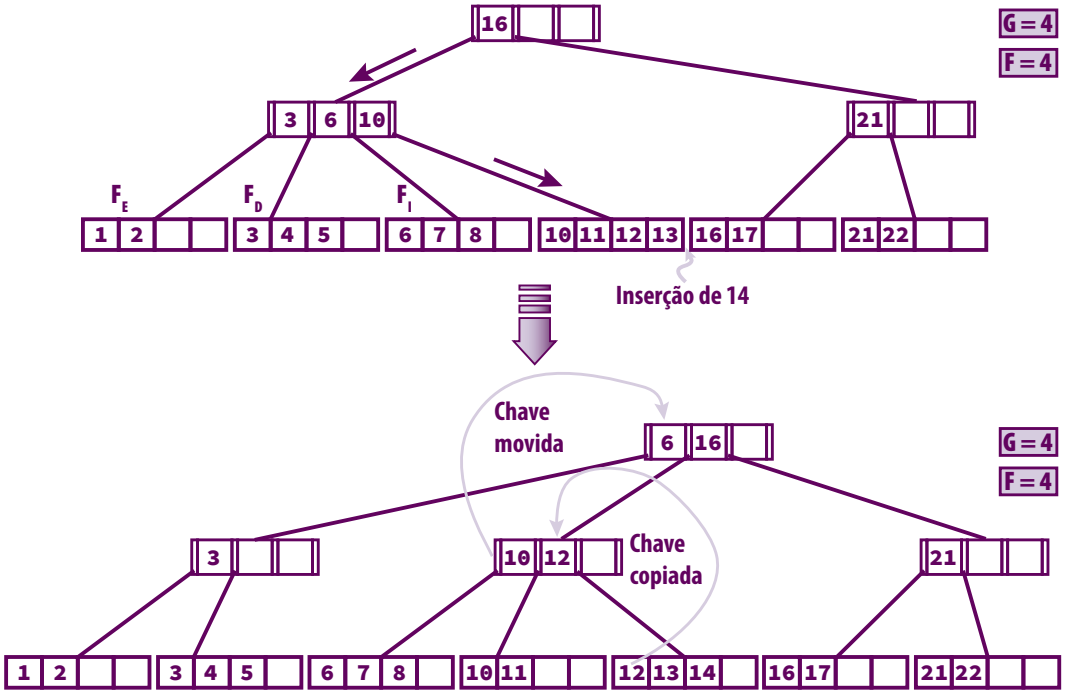


Figura 6–41: Exemplo de Inserção em Árvore B+ 4

6.5.4 Remoção

Obviamente, a chave a ser removida deve sempre residir numa folha. Se o número de chaves restantes na folha na qual ocorre a remoção não ficar abaixo da metade, o complemento da remoção requer apenas reorganização das chaves na folha para mantê-la ordenada.

Quando uma remoção faz com que uma folha fique com um número de chaves abaixo do esperado, podem ocorrer duas situações:

[1] As chaves da folha e as chaves de uma folha-irmã são redistribuídas entre si, de tal modo que ambas permaneçam com um número de chaves dentro do limite permitido. Nesse caso, a chave do nó-pai cujo filho direito é a folha direita envolvida na distribuição de chaves deve ser substituída pela menor chave dessa folha.

[2] A folha é removida e suas chaves são incluídas numa irmã. Então as chaves dessa folha e sua irmã são combinadas para constituir uma única folha. Nesse caso, o filho do nó-pai que aponta para a folha que foi removida e a chave à sua direita (ou esquerda) devem ser removidos. Outros ajustes em nós ancestrais podem ser necessários, assim como ocorre com árvores B (v. **Seção 6.4.4**).

É interessante notar que, com exceção da primeira folha de uma árvore B+ (i.e., aquela mais à esquerda da árvore), cada primeira chave de uma folha aparece num nó interno. Assim o número de chaves em nós internos é igual ao número de folhas menos um. Excetuando-se a chave mais à esquerda de uma árvore B+, os seguintes fatos são verdadeiros:

- A primeira chave da primeira folha filha de um dado nó interno aparece num nó interno que está num nível superior ao nível mais baixo de nós internos.
- As primeiras chaves das demais folhas filhas de um dado nó interno aparecem nesse nó interno.

A **Figura 6–42** ilustra essas afirmações. Note que a folha mais à esquerda nessa figura não é a folha mais à esquerda da árvore.

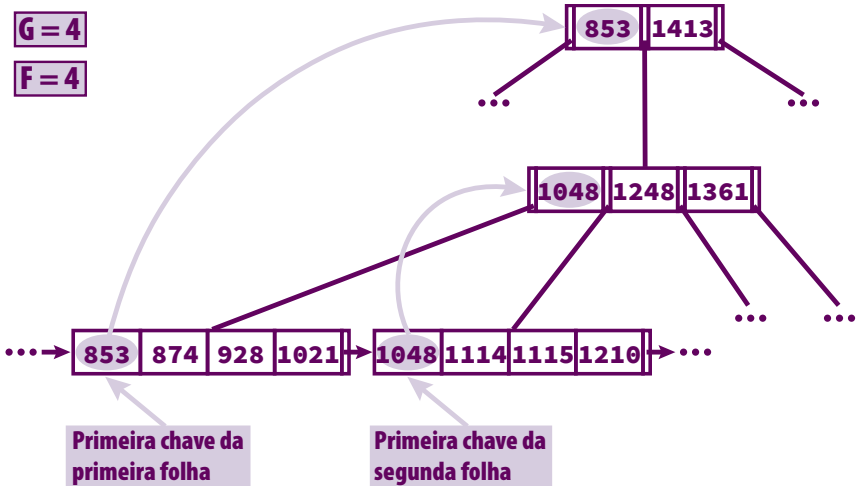


Figura 6–42: Primeira Chave de uma Folha de Árvore B+

Uma consequência desses fatos na remoção de uma chave é que, quando a primeira chave de uma folha (salvo a exceção apontada acima) é alterada, essa alteração deve se refletir no nó interno que contém essa chave. Isso ocorre em duas ocasiões:

[1] A referida primeira chave é removida, como ilustra a **Figura 6–43**.

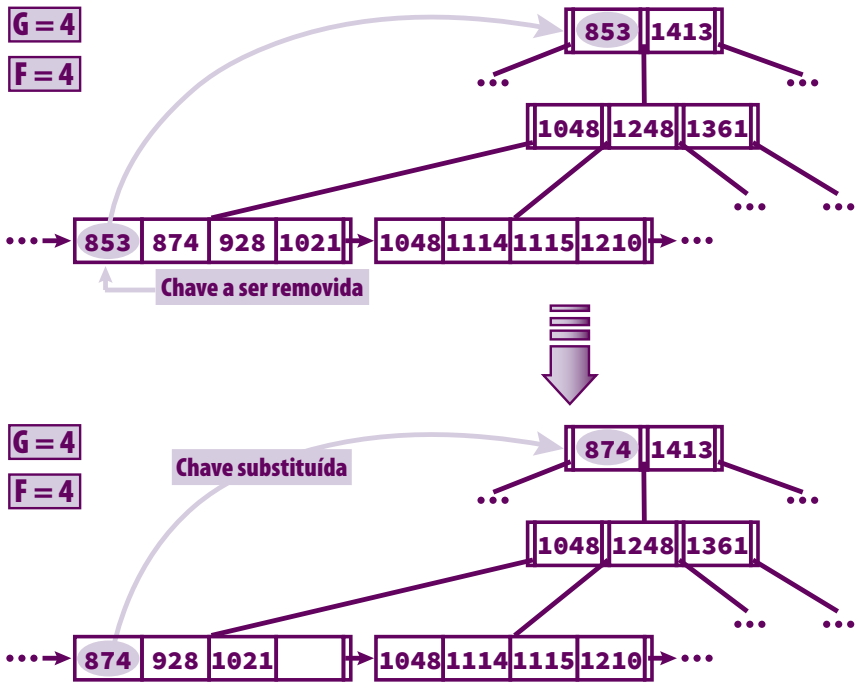


Figura 6–43: Remoção da Primeira Chave de uma Folha de Árvore B+

[2] Após uma remoção, a primeira chave de uma folha é alterada devido a uma transferência de chaves de uma folha para outra. Essa situação é mostrada na **Figura 6–44**. Na situação ilustrada nessa figura, a folha da esquerda *F* tem uma chave removida o que faz com ela fique com um número de folhas abaixo do mínimo permitido. Assim ela recebe a menor chave da folha direita *D*, de modo que o nó interno que contém essa chave precisa ser atualizado. É importante notar que, se a remoção ocorresse na folha da direita *D*, de tal maneira que ela se tornasse deficiente e a folha da esquerda *F* cedesse

uma chave para suprir essa deficiência, novamente, o nó interno contendo a antiga primeira chave de *D* precisaria ser atualizado.

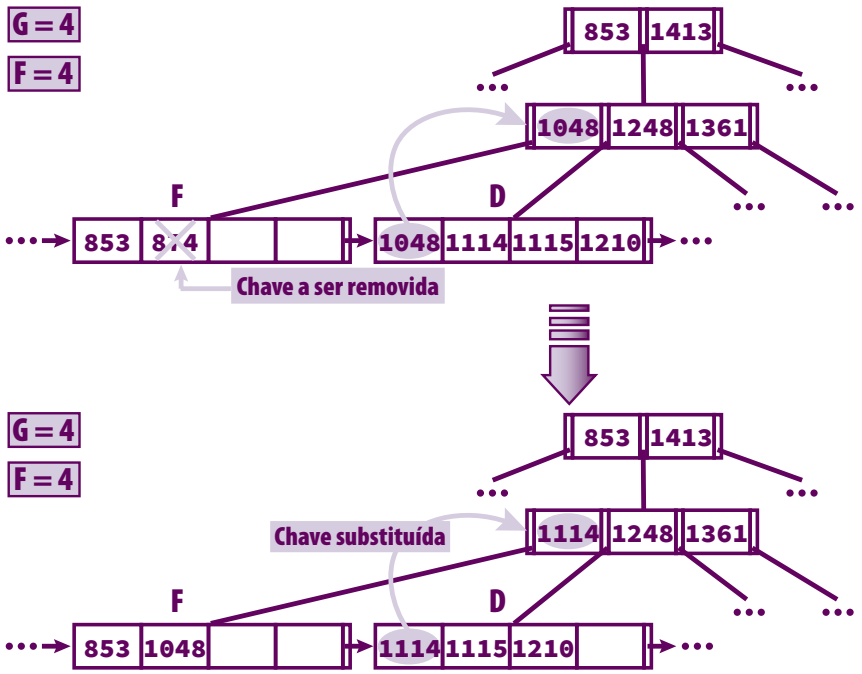


Figura 6–44: Remoção com Redistribuição de Chaves em Árvore B+

Quando, após uma remoção, ocorre uma fusão de nós, de sorte que um nó deixa de existir, deve haver uma remoção correspondente num nó interno. Essa situação é ilustrada na **Figura 6–45**. Nessa figura, a folha *F* (ou *D*) é extinta e, consequentemente, a chave que dividia as folhas *F* e *D* é removida do nó interno que a contém.

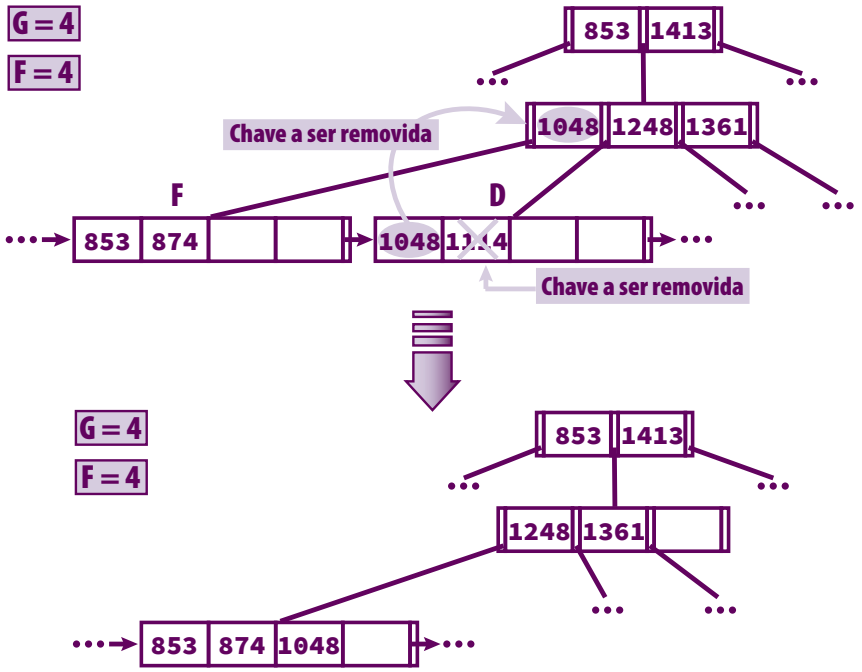


Figura 6–45: Remoção com Fusão de Folhas em Árvore B+

O algoritmo de remoção de árvores B+ segue os passos descritos na **Figura 6–46**.

ALGORITMO REMOVEEMÁRVOREB+

ENTRADA: A chave e a árvore B+

SAÍDA: A árvore modificada, se a operação for bem-sucedida, e um valor indicando o sucesso ou fracasso da operação

1. Faça uma busca pela chave a ser removida, empilhando os nós que se encontram no caminho da raiz até a folha que deverá conter essa chave

2. Se a chave não for encontrada numa folha, encerre a operação informando que ela fracassou

3. Se a chave for encontrada numa folha F, remova-a, promovendo os deslocamentos de chaves que se fizerem necessários

4. Se a folha F for a raiz da árvore

4.1 Se F ficar vazia, torne a árvore vazia

4.2 Encerre informando o sucesso da operação

5. Se a chave removida era a menor chave da folha F, guarde a nova menor chave (ela poderá ser necessária para substituir a chave removida em algum nó interno)

6. Se a folha F não ficar com um número de chaves menor do que o mínimo estabelecido, atualize o índice usando o algoritmo **ATUALIZAINDICE**

7. Caso contrário, faça o seguinte:

7.1 Se a irmã esquerda F_E da folha F tiver um número de chaves maior do que o mínimo requerido:

7.1.1 Transfira a maior chave de F_E para F

7.1.2 Atualize os nós internos usando o algoritmo **ATUALIZAINDICE**

7.2 Caso contrário, se a irmã direita F_D da folha F tiver um número de chaves maior do que o mínimo requerido

7.2.1 Transfira a menor chave de F_D para F

7.2.2 Atualize os nós internos usando o algoritmo **ATUALIZAINDICE**

7.3 Caso contrário, se F possui irmã esquerda F_E

7.3.1 Transfira todas as chaves de F para F_E

7.3.2 Remova do último nó interno visitado o apontador para a folha F e a chave que separa as folhas F para F_E

7.3.3 Atualize os nós internos usando o algoritmo **ATUALIZAINDICE**

7.4 Caso contrário (F possui irmã direita F_D):

7.4.4 Transfira todas as chaves de F para F_D

7.4.5 Remova do último nó interno visitado o apontador para a folha F e a chave que separa as folhas F para F_D

7.4.6 Atualize os nós internos usando o algoritmo **ATUALIZAINDICE**

8. Retorne informando o sucesso da operação

FIGURA 6–46: ALGORITMO DE REMOÇÃO EM ÁRVORE B+

O algoritmo **ATUALIZAINDICEEMÁRVOREB+**, que combina um nó interno com outros de uma árvore B+ quando ele fica com grau abaixo do mínimo estabelecido, segue os passos descritos na **Figura 6–47**.

ALGORITMO ATUALIZAINDICEEMÁRVOREB+

ENTRADA: Uma pilha contendo os nós encontrados desde a raiz até o último nó interno visitado numa operação de remoção, a chave removida, a (eventual) chave substituta, a (eventual) chave a ser removida do índice

SAÍDA: A árvore B+ alterada

1. Enquanto a pilha não estiver vazia, faça o seguinte:
- **1.1** Desempilhe um nó (*N*)
- **1.2** Se a chave removida se encontrar em *N* e precisar ser substituída, troque-a pela chave substituta
- **1.3** Se não houver remoção em *N*, encerre
- **1.4** Caso contrário, remova a chave juntamente com seu filho direito
- **1.5** Se o número de chaves de *N* não for menor do que o mínimo, encerre
- **1.6** Caso contrário, ajuste os nós internos usando o algoritmo **JUNTANOSB+**

FIGURA 6–47: ALGORITMO DE ATUALIZAÇÃO DE ÍNDICE EM ÁRVORE B+

O algoritmo **JUNTANOSB+** invocado pelo algoritmo de remoção em árvores B+ é semelhante ao algoritmo de junção de nós para árvores B apresentado na **Figura 6–21** (v. **Seção 6.4.4**).

Novamente, nos exemplos a seguir, o número máximo de filhos de um nó interno *G* é *4* e o número máximo de chaves *F* numa folha também é *4*. Portanto o número mínimo de chaves num nó interno é *1* e o número mínimo de chaves numa folha é *2*. Para simplificar, o encadeamento das folhas é omitido das figuras que ilustram esse exemplos.

A **Figura 6–48** apresenta o caso mais simples de remoção quando a chave a ser removida é encontrada. Nesse caso, quando a chave é removida de uma folha, essa folha permanece com o número de chaves dentro do limite requerido. Além disso, a chave removida não aparece no índice, de modo que nenhuma outra ação se faz necessária.

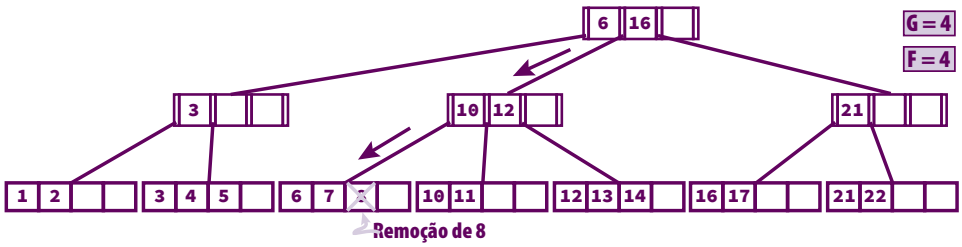


FIGURA 6–48: EXEMPLO DE REMOÇÃO EM ÁRVORE B+ 1

A **Figura 6–49** mostra um caso mais complicado de remoção, que requer atualização de um nó interno. Os números escritos em círculos nessa figura mostram a sequência de passos seguida para a obtenção da árvore resultante da remoção. Esses passos são:

❶ A chave *2* é removida da folha *F*, o que faz com que essa folha fique com um número de chaves abaixo do limite mínimo requerido (que é *2*)

❷ A chave *3* é transferida da folha-irmã direita *D* para a folha *F*

❸ As chaves de *D* são movidas uma posição para a esquerda

❹ A chave *3* do nó *I* é substituída por *4*, que é a menor chave do nó *D*

A **Figura 6–50** exibe outro caso complicado de remoção, que requer a remoção de uma chave de um nó interno. Os passos envolvidos nessa operação são:

❶ A chave *7* é removida da folha *F*, o que faz com que essa folha fique com um número de chaves abaixo do limite mínimo requerido
❷ A chave *6* é transferida da folha *F* para a folha *D*, o que requer o deslocamento para a direita das chaves que já estavam presentes em *D*
❸ A folha *F* é removida, visto que ela ficou vazia
❹ A chave que separava *F* e *D* e o ponteiro para a folha *F* são removidos do nó *I*, pois a folha *F* deixou de existir
❺ A chave *12* do nó *I* e seus filhos são movidos uma posição para a esquerda

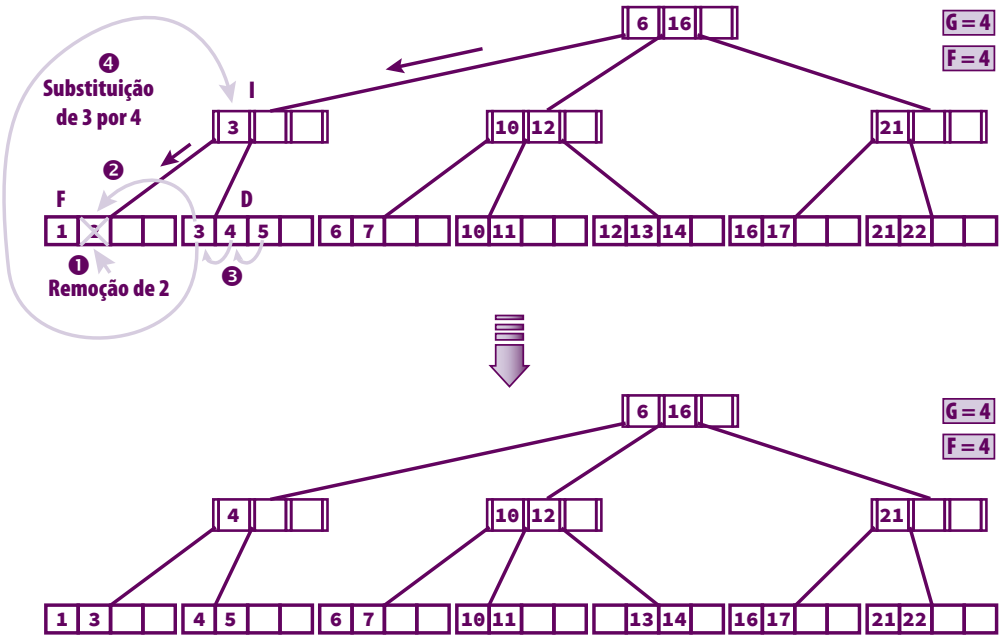


Figura 6–49: Exemplo de Remoção em Árvore B+ 2

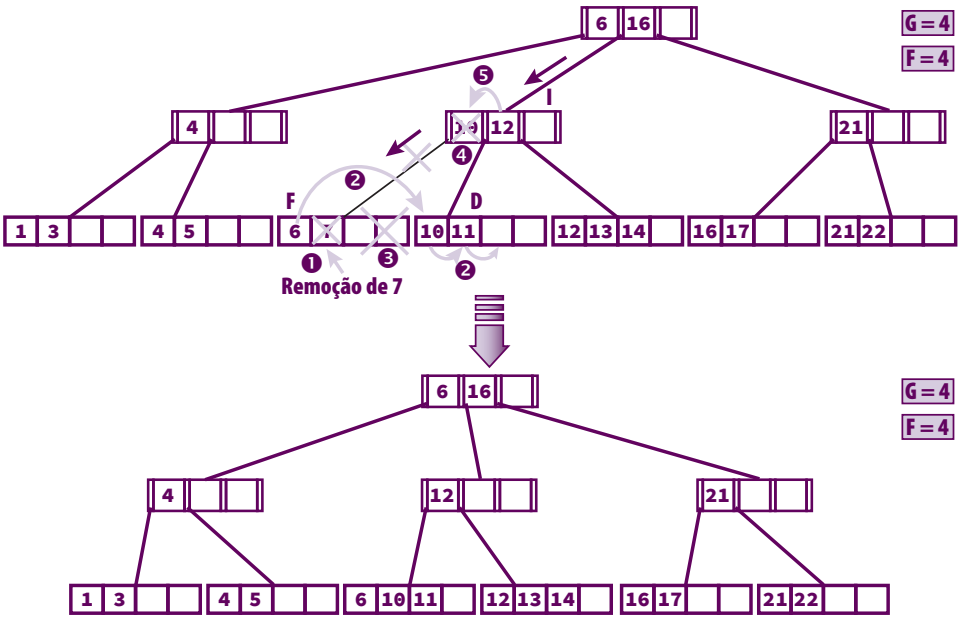


Figura 6–50: Exemplo de Remoção em Árvore B+ 3

A **Figura 6–51** apresenta outro caso complicado de remoção de uma chave numa árvore B+, que requer a atualização do índice da árvore. Na situação ilustrada na **Figura 6–51**, os passos envolvidos na atualização da árvore são os seguintes:

1. A chave *3* é removida da folha *F*, fazendo com que essa folha fique com o número de chaves abaixo do limite mínimo requerido
2. A chave *1* é transferida de *F* para sua irmã direita *D*, o que requer o deslocamento para a direita das chaves que já estavam presentes em *D*
3. A folha *F* se torna vazia e é excluída da árvore
4. A chave *4*, que é a única chave do nó *I*, é removida juntamente com seus filhos, deixando esse nó vazio (o nó *D* torna-se temporariamente órfão)
5. O nó *I*, que ficou vazio, é removido
6. A chave *6* da raiz é movida para o nó *J*, requerendo o deslocamento para a direita da chave *12* de *J* juntamente com os filhos dessa última chave
7. O filho esquerdo da chave *6* passa a apontar para o nó *D*
8. A chave *16* da raiz é movida uma posição para esquerda juntamente com seus filhos

A **Figura 6–52** mostra outro caso complicado de remoção em árvore B+, que requer atualização de nós internos. Com base nos algoritmos apresentados na **Figura 6–46** e na **Figura 6–47**, o leitor é desafiado a explicar em detalhes como a árvore resultante dessa inserção é obtida.

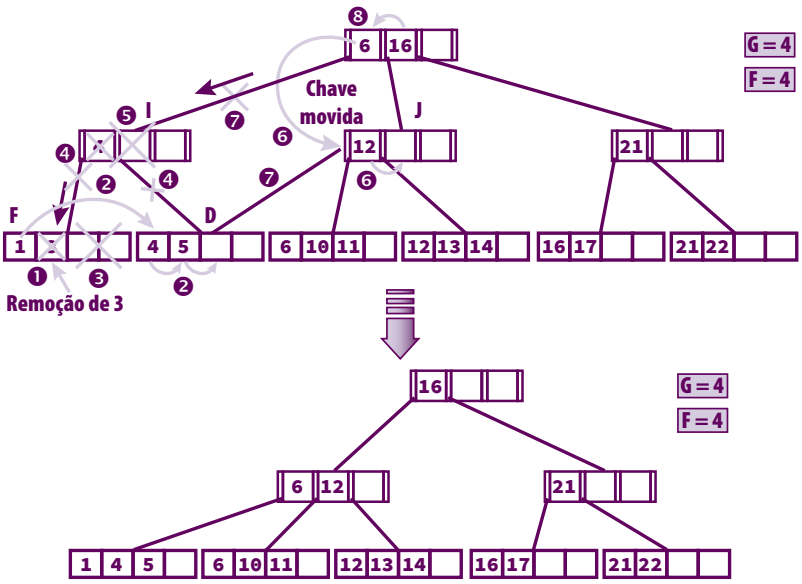


Figura 6–51: Exemplo de Remoção em Árvore B+ 4

6.5.5 Busca de Intervalo

As folhas de uma árvore B+ são tipicamente unidas umas às outras formando uma lista encadeada. Isso torna consultas de intervalo ou qualquer outro tipo de acesso ordenado às chaves mais eficiente. Esse encadeamento não aumenta substancialmente o consumo de espaço ou manutenção da árvore e ilustra uma das vantagens mais significativas de uma árvore B+ sobre uma árvore B. Numa árvore B, como nem todas as chaves estão presentes nas folhas, tal lista encadeada ordenada não pode ser construída.

Além de busca comum, uma árvore B+ também suporta a consulta de intervalo. Isto é, encontrar todos os objetos cujas chaves pertencem a um intervalo *R*. Para fazer isso, todas as folhas de uma árvore B+ são encadeadas.

Se for desejado procurar todos os objetos cujas chaves estão no intervalo $R = [inf .. sup]$, efetua-se uma busca comum pela chave *inferior*, que termina numa folha *f*. Então recuperam-se todos os registros em *f* a partir de *inf* ou da primeira chave maior do que *inf*, se a chave *inf* não existir. Depois, segue-se o ponteiro para a folha seguinte e assim por diante, até que seja encontrada uma chave maior do que *sup*.

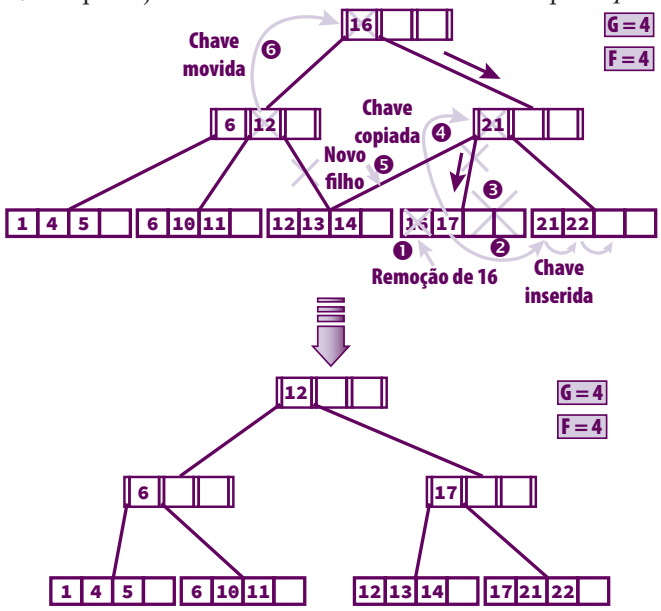


FIGURA 6–52: EXEMPLO DE REMOÇÃO EM ÁRVORE B+ 5

6.5.6 Implementação

Além dos tipos `tChave` e `tChaveIndice` definidos na **Seção 6.1.5**, os seguintes tipos e constantes simbólicas são utilizados na implementação de árvores B+.

```
   /* Constantes usadas para informar se   */
   /* um nó é interno, folha ou está vazio */
typedef enum {FOLHA, INTERNO, NO_VAZIO} tTipoDoNo;

      /***                                  ***/
      /*** Dimensionamento de nós da árvore ***/

#define TB   4096 /* Tamanho do bloco lido/escrito  */
#define TC   sizeof(tChave) /* Tamanho de uma chave */
#define TCI  sizeof(tChaveIndice)  /* Tamanho de um par chave/índice */
#define TI   sizeof(int) /* Tamanho de um filho e do inteiro */
                         /* que representa o grau do nó      */
#define TT   sizeof(tTipoDoNo) /* Tamanho de uma constante de enumeração */

#define CORRECAO 5 /* Correção devido a preenchimento (tentativa e erro) */

   /* Cálculo do grau da árvore */
#define G   ((TB - CORRECAO - TT - TI + TC)/(TC + TI))

#define TG   G%2 ? G/2 + 1 : G/2  /* Metade do grau da árvore */

   /* Número máximo de pares chave/índice em cada folha */
#define F   ((TB - CORRECAO - TT - 2*TI)/TCI)

   /* Metade do máximo de pares chave/índice numa folha */
#define TF   F%2 ? F/2 + 1 : F/2

      /*** Fim do dimensionamento de nós da árvore ***/
```

```
   /* Tipo usado para armazenamento de nós internos */
typedef struct {
           int    nFilhos;       /* Número de filhos do nó */
           tChave chaves[G - 1]; /* Array de chaves        */
           int    filhos[G];     /* Array de filhos do nó  */
        } tNoInterno;

   /* Tipo usado para armazenamento de folhas */
typedef struct {
           int          nChaves;   /* Número de chaves do nó */
           tChaveIndice chaves[F]; /* Array de pares chave/índice*/
           int          proximaFolha; /* Índice da próxima folha */
        } tNoFolha;

   /* Tipo usado para armazenamento de nós (internos/folhas) da árvore B+ */
typedef struct {
           tTipoDoNo     tipoDoNo; /* Tipo do nó */
           union {
              tNoInterno noInterno; /* Usado se o nó é interno */
              tNoFolha   noFolha;   /* Usado se o nó é folha   */
           } conteudo;
        } tNoBM, *tArvoreBM;
```

O dimensionamento de nós de uma árvore B+ é semelhante àquele discutido para árvores multidirecionais descendentes e árvores B (v. **Seção 6.3.1**). Note, entretanto, que aqui há duas complicações adicionais: (1) existem dois tipos de nós e (2) há necessidade de correção devido ao preenchimento das estruturas que representam nós da árvore. O restante da implementação de árvores B+ é longo demais para ser exibido num livro didático, mas ela pode ser encontrada no site dedicado ao livro na internet.

6.5.7 Análise

A principal vantagem de árvores B+ é permitir o acesso a chaves em ordem crescente sem precisar de caminhamento em ordem infixa, que é bastante dispendioso. Assim árvores B+ são bem convenientes para aplicações que requerem acessos direto e sequencial. Uma vantagem adicional de árvores B+ é que, como os nós internos não contêm índices de registros, pode-se usar o espaço economizado para aumentar o número de chaves em cada nó interno, o que ajuda a diminuir a altura do conjunto de índices.

Como, tipicamente, os registros residem em memória secundária, cada operação de recuperação de registro requer, pelo menos, dois acessos à memória secundária. Mas isso é melhor do que manter registros inteiros nas folhas da árvore, porque, nesse caso, essas folhas poderiam conter um número bem menor de chaves, de modo que a árvore se tornaria mais profunda e, consequentemente, as buscas requereriam um número maior de acessos ao disco.

Árvores B+ são tão eficientes quanto árvores B em termos de busca e inserção. Quer dizer, numa operação de busca ou atualização numa árvore B+ com grau mínimo d, é necessário acessar apenas $\theta(\log_d n)$ blocos em memória secundária. Como um valor típico de d é maior do que 100, mesmo se houver bilhões de chaves, a altura de uma árvore B+ será no máximo 4 ou 5. Esse resultado é formalizado no teorema a seguir.

Teorema 6.3: Numa árvore B+ com grau mínimo d e n chaves, (a) o custo de transferência em operações de busca, inserção e remoção é $\theta(\log_d n)$. (b) O custo de acesso de uma busca de intervalo é $\theta(\log_d n + m - d)$, sendo m o número de chaves obtidas numa consulta de intervalo.

Prova: (a) Para uma árvore B+, todos os algoritmos comuns de busca, inserção e remoção visitam os nós num caminho desde a raiz até uma folha. Logo, do ponto de vista de análise assintótica, o custo de transferência desses algoritmos é igual à altura da árvore. Quando há n chaves e o grau mínimo da árvore é d,

a altura da árvore é $\theta(\log_d n)$. Assim os custos de transferência desses algoritmos também é $\theta(\log_d n)$. (b) De acordo com o item (a) o custo de acesso à primeira chave do intervalo é $\theta(\log_d n)$. Depois, é necessário transferir $m - d$ blocos para acessar as demais chaves do intervalo. ■

6.6 Outras Variantes de Árvores B

Árvore B+ é a variante mais comum de árvore B, mas existem várias outras. Uma **árvore B*** é uma variante de árvore B na qual cada nó é pelo menos 2/3 preenchido (em vez do mínimo de preenchimento pela metade de árvores B). Inserção em árvores B* emprega um esquema local de redistribuição para retardar divisão até que dois nós-irmãos estejam cheios. Então os dois nós são divididos em três, cada um dos quais com 2/3 do número total de chaves envolvidas nessa operação. Esse esquema garante que ocupação dos nós seja pelo menos 66%, enquanto requer apenas pequenos ajustes dos algoritmos de inserção e remoção usados para árvores B originais. Deve-se notar que esse aumento em índice de utilização de nós tem como efeito colateral acelerar as buscas, visto que a altura da árvore resultante pode ser menor.

Numa árvore B*, todos os nós, exceto a raiz, contêm pelo menos 2/3 do número máximo de chaves permitido em cada nó (em vez da metade como em árvores B). Uma consequência desse fato é que a frequência com que ocorrem divisões de nós é reduzida. Além disso, as divisões em árvores B* envolvem três nós (e não dois como em árvores B). Quer dizer, em árvores B*, uma divisão de nós envolve dois nós originais que resultam em três após essa divisão.

A divisão de nós é adiada por meio da redistribuição de chaves entre irmãos quando o nó no qual deveria ocorrer uma inserção já se encontra completo. Por exemplo, suponha que a árvore B* da **Figura 6–53** tenha ordem igual a *9* e se deseje inserir a chave *11*. Observe que caso se tratasse de uma árvore B, ocorreria uma divisão de nós devido ao excesso de chaves no nó no qual ocorre a inserção. No caso de árvore B*, no entanto, essa divisão é adiada por meio de uma redistribuição de chaves entre nós irmãos, como mostra a **Figura 6–53**. Note que as chaves dos nós-irmãos são distribuídas igualmente entre eles e a chave mediana ocupa o lugar da chave que tem os dois irmãos em questão como filhos. Note ainda que essa redistribuição de chaves abre espaço nos nós-irmãos envolvidos na divisão.

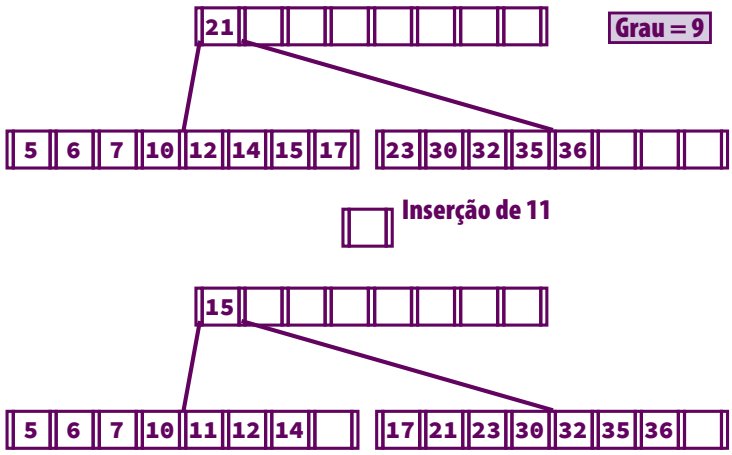


FIGURA 6–53: EXEMPLO DE INSERÇÃO EM ÁRVORE B* 1

Se um nó e seu irmão estiverem ambos completos, um nó é criado e as chaves desses nós mais a chave que os divide no nó-pai são distribuídas entre os três nós. Além disso, o nó-pai passa a ter uma chave a mais e os três nós participantes da divisão terão dois terços do número máximo permitido de filhos. Por exemplo, suponha que a árvore B* na porção superior da **Figura 6–54** tenha ordem igual a *9* e se deseje inserir a chave *9*. O resultado aparece na porção inferior dessa mesma figura.

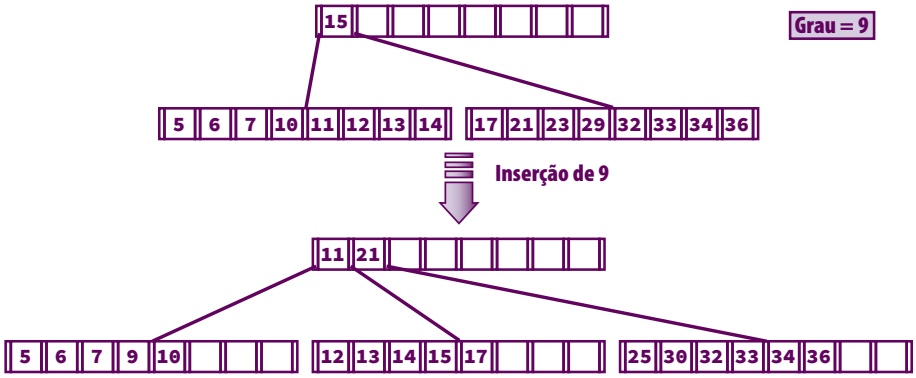


Figura 6–54: Exemplo de Inserção em Árvore B* 2

Outra variante de árvores B relativamente conhecida é a **árvore B#**. Se você tornou-se fã de árvores B e suas variantes, aconselha-se que consulte a internet sobre o vasto material que lá se encontra sobre o assunto.

6.7 Comparando Árvores Multidirecionais de Busca

Árvores B e B+ requerem codificações longas, mas que não são tão complicadas de entender quanto aquelas requeridas para árvores binárias balanceadas (p. ex., árvores AVL). Além disso, essa longa codificação tem recompensa, pois essas estruturas têm poucas competidoras para implementação de tabelas de busca em memória secundária. Uma dessas competidoras é a tabela de dispersão extensível (v. **Capítulo 8**), que é muito mais complicada de entender (e mais complicada ainda de implementar).

Árvores B e suas variantes (notadamente as árvores B+) são utilizadas em inúmeras aplicações práticas bastante conhecidas. Por exemplo, árvores B+ fazem parte de quase todos os sistemas de gerenciamento de bancos de dados, como Oracle e SQL Server. Além disso, sistemas operacionais (p. ex., Windows NT e Linux) também utilizam intensamente essas estruturas.

A **Tabela 6–4** apresenta comparações entre as árvores multidirecionais discutidas neste capítulo. Com exceção de árvores multidirecionais descendentes de busca, todas as demais árvores de busca discutidas neste capítulo têm custo de transferência $\theta(\log_d n)$, em que d é o grau mínimo da árvore.

	Árvore Multidirecional			
	Descendente	**B**	**B+**	**B***
Chaves armazenadas em...	Qualquer nó	Qualquer nó	Folhas	Qualquer nó
Grau mínimo de um nó	Não há	Metade do grau máximo	Metade do grau máximo	Dois terços do grau máximo
Divisão de nós	Não há	Um nó é dividido em dois	Um nó é dividido em dois	Dois nós são divididos em três
Combinação de nós	Não há	Dois nós são combinados em um	Dois nós são combinados em um	Três nós são combinados em dois
Permite inserção massiva?	Não	Sim	Sim	Sim
Facilita pesquisa de intervalo?	Não	Não	Sim	Não
Utilização	Didática	Prática	Prática	Prática

Tabela 6–4: Comparações entre Árvores Multidirecionais

6.8 Exemplos de Programação

6.8.1 Caminhamento em Árvore Multidirecional de Busca

Preâmbulo: Caminhamento em árvores multidirecionais tem o mesmo significado que caminhamento em árvores binárias: visitar todos os nós de uma árvore numa dada sequência. Aqui, *visitar um nó* implica em visitar todas as chaves do nó na ordem em que se encontram. Assim como ocorre no caso de árvores binárias de busca, um caminhamento em ordem infixa numa árvore multidirecional de busca acarreta na visitação em ordem crescente de todas as chaves armazenadas na árvore.

Problema: Escreva uma função que efetua um caminhamento em ordem infixa numa árvore B usando os tipos e constantes definidos na **Seção 6.4.5**.

Solução: A função `CaminhamentoInfixoB()`, apresentada a seguir, executa um caminhamento em ordem infixa numa árvore B e escreve num arquivo de texto todas as chaves da árvore em ordem crescente. Os parâmetros dessa função são:

- `streamArvore` (entrada) — stream associado ao arquivo que armazena a árvore
- `posNo` (entrada) — posição no arquivo que contém a árvore do nó inicial do caminhamento
- `streamChaves` (entrada) — stream associado ao arquivo onde serão escritas as chaves

```
void CaminhamentoInfixoB( FILE *streamArvore, int posNo, FILE *streamChaves )
{
   int         i;
   tNoMultiMS  umNo;

      /* Nenhum dos streams recebidos pode ser NULL */
   ASSEGURA(streamArvore, ERRO_STREAM_NULL(streamArvore, CaminhamentoInfixoB));
   ASSEGURA(streamChaves, ERRO_STREAM_NULL(streamChaves, CaminhamentoInfixoB));

      /* Visita cada nó da árvore em ordem infixa até   */
      /* que a posição 'posNo' assuma um valor inválido */
   if (posNo != POSICAO_NULA) {
         /* Lê o nó na posição 'posNo' do arquivo que contém a árvore */
      LeNoMultiMS(streamArvore, posNo, &umNo);

         /* Para cada nó, caminha-se recursivamente na subárvore esquerda de */
         /* cada chave e depois escreve-se essa chave no arquivo de texto    */
      for (i = 0; i < umNo.nFilhos-1; ++i) {
         CaminhamentoInfixoB( streamArvore, umNo.filhos[i], streamChaves );
            /* Visita a chave de índice i */
         fprintf(streamChaves, "%d\n", umNo.chaves[i].chave);
      }

         /* Agora caminha-se recursivamente na subárvore direita da última chave  */
      CaminhamentoInfixoB(streamArvore, umNo.filhos[umNo.nFilhos-1], streamChaves);
   }
}
```

6.8.2 Menor e Maior Chaves de uma Árvore Multidirecional de Busca

Problema: (a) Usando os tipos e constantes definidos na **Seção 6.4.5**, escreva uma função que encontra a menor chave numa árvore B. (b) Usando esses mesmos tipos e constantes, escreva uma função que encontra a maior chave numa árvore B.

Solução de (a): A função `MenorChaveMultiMS()` foi usada na **Seção 6.3.6** para encontrar a chave que é sucessora imediata de uma dada chave, mas, em geral, ela encontra a menor chave de uma árvore multidirecional de busca armazenada em arquivo.

Solução de (b): A função `MaiorChaveB()` encontra, numa árvore B, a maior chave com a respectiva posição (índice) do registro correspondente. Essa função retorna o par chave/índice que contém a maior chave da árvore e seus parâmetros são:

- `streamArvore` (entrada) — stream associado ao arquivo que contém a árvore
- `raiz` (entrada) — posição da raiz da árvore no arquivo que contém a árvore

```
tChaveIndice MaiorChaveB(FILE *streamArvore, int raiz)
{
   tNoMultiMS umNo; /* Armazena um nó */
   int        pos = raiz; /* Armazena a posição de um nó, começando pela raiz */

      /* Verifica se o stream que representa a árvore é válido */
   ASSEGURA( streamArvore, ERRO_STREAM_NULL(streamArvore, MaiorChaveB) );

      /* Verifica se a raiz da árvore é válida */
   ASSEGURA(raiz >= 0, ERRO_POSICAO(MaiorChaveB));

   do { /* Encontra o nó mais à direita na árvore */
         /* Lê o nó cuja posição no arquivo é indicada por 'pos' */
      LeNoMultiMS(streamArvore, pos, &umNo);

         /* Passa para o filho mais à direita deste nó */
      pos = umNo.filhos[umNo.nFilhos - 1];
   } while (pos != POSICAO_NULA);

      /* Retorna o par chave/índice que tem a maior chave */
   return umNo.chaves[umNo.nFilhos - 2];
}
```

6.8.3 Número de Nós de uma Árvore B

Problema: Escreva uma função que encontra o número de nós de uma árvore B que usa os mesmos tipos e constantes definidos na **Seção 6.4.5**.

Solução: A função `NumeroDeNosB()` calcula o número de nós de uma árvore B. Essa função retorna o número de nós da referida árvore e seu único parâmetro é `stream`, que representa o stream associado ao arquivo contendo a árvore.

```
int NumeroDeNosB(FILE *stream)
{
   int        nNos = 0;
   tNoMultiMS umNo;

      /* O stream que contém a árvore não pode ser NULL */
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, NumeroDeNosB) );

      /* Tenta mover o apontador de posição do arquivo para o seu início */
   MoveApontador(stream, 0, SEEK_SET);

      /* Acessa sequencialmente cada nó da árvore */
      /* contando o número de chaves em cada nó   */
   while (1) {
      fread(&umNo, sizeof(umNo), 1, stream); /* Lê um nó da árvore */

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

         /* Verifica se o nó não foi removido */
      if (umNo.nFilhos > 0)
         ++nNos;
   }
```

```
      /* Verifica se houve erro de leitura */
   ASSEGURA(!ferror(stream),ERRO_FREAD(NumeroDeNosB));

   return nNos; /* Serviço completo */
}
```

6.8.4 Número de Chaves de uma Árvore B

Problema: Escreva uma função que encontra o número de chaves de uma árvore B que usa os mesmos tipos e constantes definidos na **Seção 6.4.5**.

Solução: A função `NumeroDeChavesB()` B calcula o número de chaves de uma árvore B. Essa função retorna o referido número de chaves e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que armazena a árvore cujo número de chaves será determinado
- `posNo` (entrada) — a posição da raiz da árvore no arquivo que a contém

```
int NumeroDeChavesB(FILE *stream)
{
   int        nChaves = 0;
   tNoMultiMS umNo;

      /* O stream que contém a árvore não pode ser NULL */
   ASSEGURA(stream, ERRO_STREAM_NULL(stream, NumeroDeChavesB) );

      /* Tenta mover o apontador de posição do arquivo para o seu início */
   MoveApontador(stream, 0, SEEK_SET);

      /* Acessa sequencialmente cada nó da árvore */
      /* contando o número de chaves em cada nó   */
   while (1) {
      fread(&umNo, sizeof(umNo), 1, stream); /* Lê um nó da árvore */

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

         /* Conta o número de chaves do nó corrente */
      if (umNo.nFilhos > 0)
         nChaves += umNo.nFilhos - 1;
   }

      /* Verifica se houve erro de leitura */
   ASSEGURA(!ferror(stream),ERRO_FREAD(NumeroDeChavesB));

   return nChaves; /* Serviço completo */
}
```

6.8.5 Altura de uma Árvore B

Problema: Escreva uma função que calcula a altura de uma árvore B que usa os mesmos tipos e constantes definidos na **Seção 6.4.5**.

Solução: A função `AlturaB()` calcula e retorna a altura de uma árvore B e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que armazena a árvore cuja altura será determinada
- `raiz` (entrada) — a posição da raiz da árvore no arquivo que a contém

```
int AlturaB(FILE *stream, int raiz)
{
   int       prof = 0;
   tNoMultiMS no;
```

```
      /* Se a árvore estiver vazia, sua altura é zero */
   if (raiz == POSICAO_NULA)
      return 0;

      /* O stream que contém a árvore não pode ser NULL */
   ASSEGURA( stream, ERRO_STREAM_NULL(stream, AlturaB) );

      /* A altura de uma árvore B corresponde ao nível de qualquer folha, já que  */
      /* todas elas estão no mesmo  nível. Como a folha mais fácil de encontrar é */
      /* aquela mais à esquerda na árvore, é isso que faz o laço a seguir.        */
   do {
      LeNoMultiMS(stream, raiz, &no);

      prof++; /* Visitou-se mais um nó */

         /* Desce-se sempre pelo filho esquerdo do nó corrente */
      raiz = no.filhos[0];
   } while (raiz != POSICAO_NULA);

   return prof;
}
```

6.8.6 Busca de Intervalo em Árvore B+

Problema: Usando os mesmos tipos e constantes definidos na **Seção 6.5.6**, escreva uma função que retorna todas as chaves armazenadas numa árvore B+ que se encontram dentro de um intervalo de valores.

Solução: A função `BuscaIntervaloBM()` coleta num arquivo todas as chaves que estão entre duas chaves especificadas de uma árvore B+. Seus parâmetros são:

- `chave1` (entrada) — primeira chave que define o intervalo
- `chave2` (entrada) — segunda chave que define o intervalo
- `arvore` (entrada) — ponteiro para a raiz da árvore na qual será feita a busca
- `*streamArv` (entrada) — stream associado ao arquivo que contém a árvore
- `streamChaves` (entrada) — stream associado ao arquivo que conterá as chaves resultantes da busca

A função `BuscaIntervaloBM()` retorna `1`, se a coleta de chaves for bem-sucedida, ou `0`, em caso contrário.

```
int BuscaIntervaloBM( tChave chave1, tChave chave2, const tNoBM* arvore,
                      FILE *streamArv, FILE *streamChaves )
{
   tChave menorChave, /* Menor chave */
          maiorChave; /* Maior chave */
   tNoBM  no; /* Armazenará cada nó visitado */
   int    i,
          encontrado = 0, /* Usada para chamar BuscaEmNoBM() */
          posNo = 0; /* Posição do nó no arquivo */

      /* Verifica se streamArv é válido */
   ASSEGURA( streamArv, ERRO_STREAM_NULL(streamArv, BuscaIntervaloBM) );

      /* Verifica se streamChaves é válido */
   ASSEGURA( streamChaves, ERRO_STREAM_NULL(streamChaves, BuscaIntervaloBM) );

      /* Verifica se o ponteiro para a raiz é válido */
   ASSEGURA( arvore, "Erro: A arvore NULL em BuscaIntervaloBM");

      /* Verifica se a árvore é vazia */
   if (arvore->tipoDoNo == NO_VAZIO)
      return 0; /* A árvore é vazia */
```

```
if (chave1 == chave2) /* Se as chaves forem iguais, não há intervalo */
   return 0; /* Não há intervalo */

/* O trecho a seguir permite que as chaves que definem */
/* o intervalo sejam introduzidas em qualquer ordem    */

if (chave1 < chave2) { /* chave1 < chave2 */
   menorChave = chave1;
   maiorChave = chave2;
} else { /* chave1 >= chave2 */
   menorChave = chave2;
   maiorChave = chave1;
}

no = *arvore; /* A busca começa na raiz da árvore */

   /* Desce na árvore até encontrar uma folha */
while (no.tipoDoNo == INTERNO) {
   i = BuscaEmNoBM(menorChave, &no, &encontrado);

      /* Desce até o próximo nó */
   posNo = encontrado ? no.conteudo.noInterno.filhos[i + 1]
                      : no.conteudo.noInterno.filhos[i];

   LeNoBM(streamArv, posNo, &no);
}

   /* Chegou-se a uma folha. Encontra, nessa folha, a posição da */
   /* primeira chave que é maior do que ou igual à menor chave   */
i = BuscaEmNoBM(menorChave, &no, &encontrado);

   /* Armazena em arquivo as chaves encontradas a partir dessa posição */
   /* na folha encontrada até encontrar a última chave da última folha */
   /* ou uma chave maior do que 'maiorChave'                           */

   /* A coleta de chaves na primeira folha encontrada é diferente daquela   */
   /* nas demais folhas da sequência porque começa com a chave na posição i */
for ( ; i < no.conteudo.noFolha.nChaves &&
      no.conteudo.noFolha.chaves[i].chave <= maiorChave; ++i ){
      /* Escreve a chave corrente */
   fprintf( streamChaves, "%ld\n", no.conteudo.noFolha.chaves[i].chave );
}

   /* Se a última chave escrita foi a última chave da */
   /* folha, é preciso ajustar o índice dessa chave   */
if (i == no.conteudo.noFolha.nChaves)
   --i;

if (no.conteudo.noFolha.chaves[i].chave > maiorChave)
      /* Foi encontrada uma chave maior do que a maior */
      /* chave do intervalo e nada mais resta a fazer  */
   return 1;

   /* Escreve cada chave de cada folha seguinte que    */
   /* não seja maior do que a maior chave do intervalo */
do {
   posNo = no.conteudo.noFolha.proximaFolha; /* Passa para a próxima folha */

   LeNoBM(streamArv, posNo, &no); /* Lê a folha  */

      /* Escreve cada chave que não é maior do que a maior chave do intervalo */
   for (i = 0; i < no.conteudo.noFolha.nChaves &&
               no.conteudo.noFolha.chaves[i].chave <= maiorChave; ++i)
```

```
            /* Escreve a chave corrente */
         fprintf( streamChaves, "%ld\n", no.conteudo.noFolha.chaves[i].chave );

         /* Checa se é preciso ajustar o índice da última chave */
      if (i == no.conteudo.noFolha.nChaves)
         --i;
   } while (posNo != POSICAO_NULA && no.conteudo.noFolha.chaves[i].chave <= maiorChave);

   return 1;
}
```

Na função `BuscaIntervaloBM()`, não faz diferença qual das duas chaves é menor do que a outra, mas as chaves devem ser diferentes para que haja um intervalo.

6.9 Exercícios de Revisão

Árvores Multidirecionais Descendentes de Busca (Seção 6.1)

1. (a) O que é uma árvore multidirecional de busca de ordem *n*? (b) Em que situação prática, árvores de busca multidirecionais são tipicamente utilizadas?
2. (a) O que é uma árvore multidirecional de busca descendente? (b) Qual é a grande vantagem do uso desse tipo de árvores com relação a árvores binárias de busca?
3. O que é um nó completo de uma árvore multidirecional de busca descendente?
4. Por que, no contexto de árvores multidirecionais de busca fala-se em *filho direito* (ou *esquerdo*) *de uma chave*, em vez de *filho direito* (ou *esquerdo*) *de um nó*?
5. (a) O que é uma semifolha? (b) Mostre que, numa árvore multidirecional de busca descendente, qualquer semifolha é um nó completo ou é uma folha.
6. Descreva o algoritmo de busca para árvores multidirecionais descendentes.
7. Por que, na prática, os registros devem ser armazenados separadamente, ao invés de serem armazenados diretamente numa árvore multidirecional de busca?
8. Descreva o algoritmo de inserção para árvores multidirecionais apresentado na **Seção 6.1.3**.
9. Explique por que o método de inserção apresentado na **Seção 6.1.3** resulta numa árvore multidirecional de busca descendente.
10. Qual é a desvantagem do método de inserção apresentado na **Seção 6.1.3**?
11. (a) Descreva o algoritmo de remoção para árvores multidirecionais descendentes apresentado na **Seção 6.1.4**. (b) Por que utilizando-se esse algoritmo uma árvore multidirecional descendente de busca pode deixar de ser assim classificada?
12. (a) Por que árvores multidirecionais descendentes de busca não são úteis na prática? (b) Afinal, se esse é o caso, para que servem árvores multidirecionais descendentes de busca?
13. Que vantagens e desvantagens apresenta o algoritmo de inserção para árvores multidirecionais descendentes de busca?

Estruturas de Dados em Memória Secundária (Seção 6.2)

14. (a) O que é transferência de disco? (b) O que é custo de transferência?
15. Como se estima o desempenho de algoritmos que lidam com tabelas de busca armazenadas em memória secundária?
16. Por que as estruturas de dados apresentadas em capítulos anteriores não são convenientes para implementação de tabelas de busca residentes em memória secundária?
17. Que dificuldades um programador inexperiente em implementação de algoritmos que manipulam dados em memória secundária pode encontrar?

18. Como se interpreta o conceito de *ponteiro* em memória secundária?

19. Como se representa ponteiro nulo em memória secundária?

Árvores Multidirecionais de Busca em Memória Secundária (Seção 6.3)

20. Por que o tamanho de um nó de uma árvore multidirecional de busca implementada em memória secundária deve ser o mais próximo possível do tamanho do bloco utilizado pelo sistema de arquivos no qual a árvore será implementada, mas o tamanho desse nó não deve ser maior do que o tamanho desse bloco?

21. Como é dimensionado o grau de uma árvore multidirecional de busca implementada em memória secundária?

22. (a) Para que serve a função `EncontraNoMultiMS()` utilizada na implementação de árvores multidirecionais descendentes de busca? (b) Em que difere essa função da função `BuscaEmNoMultiMS()`?

23. Se o tamanho de bloco varia de acordo com o sistema de arquivo, como ele pode ser considerado constante como nas implementações apresentadas neste capítulo?

24. Explique a necessidade da constante simbólica `POSICAO_NULA` em implementações de árvores multidirecionais de busca em memória secundária.

25. Qual é a importância da função `LeNoMultiMS()` em implementações de árvores multidirecionais de busca em memória secundária.

26. Explique o funcionamento da função `CompactaNoMultiMS()` definida na **Seção 6.3.6**.

27. (a) O que é preenchimento de estruturas? (b) Por que muitos compiladores preenchem uma estrutura com espaços vazios?

28. (a) Por que não existe preenchimento de arrays? (b) É possível haver preenchimento de uma união?

29. Como é possível evitar que preenchimentos de estruturas (i.e., espaços vazios) sejam copiados para um arquivo?

30. Quais abordagens podem ser utilizadas para dimensionamento do grau de uma árvore multidirecional de busca que levem em consideração preenchimento de estruturas?

Árvores B (Seção 6.4)

31. O que é uma árvore multidirecional de busca balanceada?

32. (a) O que é uma árvore B? (b) Quais são as semelhanças entre árvores B e árvores de busca multidirecionais descendentes? (c) Quais são as diferenças entre árvores B e árvores de busca multidirecionais descendentes?

33. Mostre que, numa árvore B, todas as semifolhas são folhas.

34. Por que não faz sentido armazenar todas as chaves de um conjunto de registros na raiz de uma árvore B de modo que apenas um acesso ao meio de armazenamento externo seja necessário?

35. (a) Quais são os passos coincidentes nos algoritmos de inserção em árvores multidirecionais descendentes e em árvores B? (b) Em que diferem esses algoritmos?

36. (a) Por que o algoritmo de inserção para árvores B utiliza uma pilha para armazenar todos os nós no caminho que vai da raiz até o nó onde deve ser feita uma inserção? (b) Por que essa pilha também armazena os endereços desses nós? (c) Por que essa pilha também armazena a posição de cada um desses nós com relação aos seus irmãos?

37. Descreva a divisão de nós que ocorre quando é feita a inserção de uma chave numa árvore B.

38. (a) O que significa inserção com tendência esquerda em árvores B? (b) O que é uma árvore B com tendência direita?

39. Apresente em forma diagramática a árvore B resultante da inserção das chaves *C*, *N*, *G*, *A* e *H* numa árvore B de ordem *5* inicialmente vazia.

40. Apresente o resultado da inserção das chaves *F*, *S*, *Q*, *K*, *C*, *L*, *H*, *T*, *V*, *W*, *M*, *R*, *N*, *P*, *A*, *B*, *X*, *Y*, *D*, *Z* e *E* (nessa ordem) numa árvore B de ordem *5* inicialmente vazia.

41. Explique como encontrar numa árvore B (a) a menor chave e (b) a maior chave.

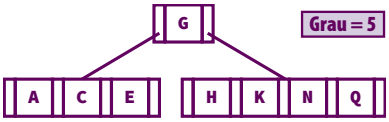
42. Suponha que se saiba que uma dada chave possui filho esquerdo não-vazio. Qual dessas operações é facilitada: encontrar a chave sucessora ou encontrar a chave antecessora da chave dada?

43. Qual é o número máximo de nós de uma árvore B de grau *G* que armazena *n* chaves em termos de *G* e *n*?

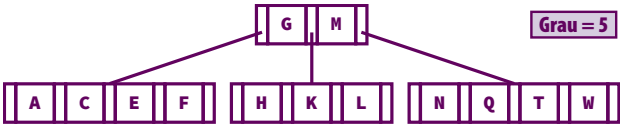
44. Sabe-se que as chaves contidas num nó de uma árvore B são dispostas em ordem crescente. Assim para efetuar a busca por uma chave dentro de um nó, é possível efetuar busca binária, em vez de busca sequencial como faz a função `BuscaEmNoMultiMS()`. Existe alguma vantagem nessa abordagem de busca binária?

45. Apresente graficamente todas as árvores B com grau igual a *6* que contenham as chaves *1*, *2*, *3*, *4* e *5*.

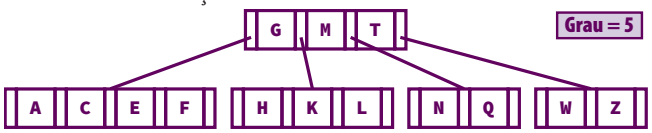
46. Mostre de forma diagramática a árvore B resultante da inserção da chave *M* na árvore B de ordem *5* mostrada na figura abaixo.



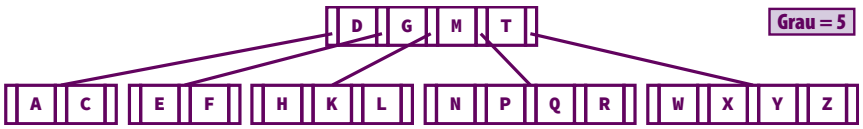
47. Mostre de modo diagramático a árvore B resultante da inserção da chave *Z* na árvore B de ordem *5* da figura abaixo.



48. Apresente a árvore B resultante da inserção da chave *D* na árvore B de ordem *5* mostrada na figura abaixo.



49. Mostre de forma diagramática a árvore B resultante da inserção da chave *S* na árvore B de ordem *5* mostrada na figura abaixo.



50. Descreva o algoritmo de remoção em árvores B.

51. Descreva o algoritmo de junção de nós após uma remoção de uma chave numa árvore B.

52. Como se efetua uma busca numa árvore B?

53. Como se dimensiona o grau de uma árvore B?

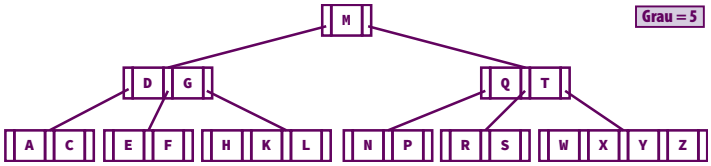
54. Em que diferem a função `EncontraCaminhoB()`, utilizada com árvores B, e a função `EncontraNoMultiMS()`, utilizada para árvores multidirecionais descendentes de busca?

55. (a) Qual é o papel desempenhado pela função `DivideNoB()` na implementação de árvores B? (b) Explique o funcionamento dessa função.

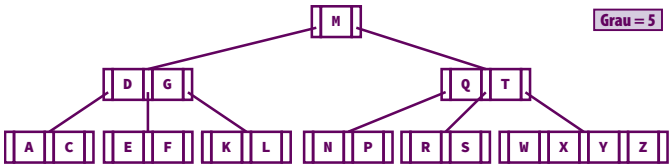
56. (a) Explique o funcionamento da função `JuntaNosB()`. (b) Para que serve essa função?

57. Por que a função `RemoveEmFolhaB()`, que remove uma chave de uma folha de uma árvore B, utiliza uma pilha como parâmetro?

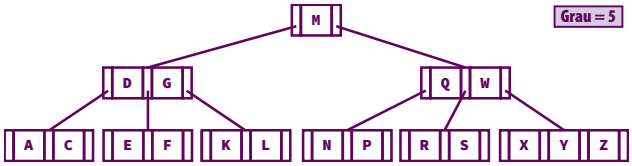
58. Apresente a árvore B resultante da remoção da chave *H* da árvore B da figura abaixo, considerando que a ordem dessa árvore é *5*.



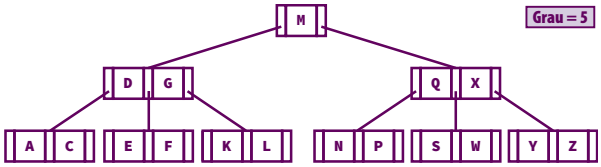
59. Apresente a árvore B resultante da remoção da chave *T* da árvore B da figura abaixo, considerando que a ordem dessa árvore é *5*.



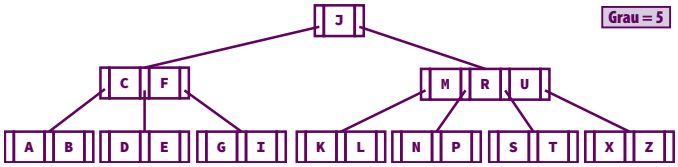
60. Apresente a árvore B resultante da remoção da chave *R* da árvore B da figura abaixo, sabendo que a ordem dessa árvore é *5*.



61. Apresente a árvore B resultante da remoção da chave *E* da árvore B da figura abaixo, considerando que a ordem dessa árvore é *5*.



62. Apresente a árvore B resultante da remoção da chave *C* da árvore B da figura abaixo, considerando que a ordem dessa árvore é *5*.



63. (a) Como se calcula a altura de uma árvore B? (b) Por que é mais fácil calcular a altura de uma árvore B do que a altura de uma árvore multidirecional descendente?

64. Existe diferença entre busca em árvore B e busca em árvore multidirecional descendente?

65. O que é persistência de dados?

66. Descreva em linhas gerais o papel desempenhado por cada função utilizada em persistência de dados.

67. Por que existe um arquivo que armazena a posição da raiz de uma árvore B, enquanto isso não ocorre no caso de árvore multidirecional de busca implementada em arquivo?

68. O que é grau mínimo de uma árvore B?

69. Qual é o custo de transferência de uma árvore B numa operação de busca, inserção ou remoção?

Árvores B+ (Seção 6.5)

70. (a) O que é uma árvore B+? (b) Que vantagens uma árvore B+ apresenta com relação a outras árvores de busca multidirecionais?

71. (a) O que é conjunto de índices de uma árvore B+? (b) O que é conjunto sequencial de uma árvore B+?

72. Qual é a diferença entre uma folha de uma árvore B e uma folha de uma árvore B+?

73. (a) Descreva o procedimento de busca numa árvore B+. (b) Qual é a diferença entre busca em árvore B e busca em árvores B+?

74. Em que diferem nós internos e nós-folhas numa árvore B+?

75. Descreva o algoritmo de inserção numa árvore B+.

76. (a) O que é busca de intervalo? (b) Como árvores B+ facilitam busca de intervalo?

77. Em que diferem o dimensionamento de grau de uma árvore B e o dimensionamento de grau de uma árvore B+?

78. Qual é o papel desempenhado por enumerações do tipo `tTipoDoNo` na implementação de uma árvore B+?

79. Por que chaves removidas de uma árvore B+ podem permanecer no conjunto de índices dessa árvore?

80. Qual é o custo de transferência de uma busca de intervalo numa árvore B+?

81. Seja G o grau de uma árvore B+ e a a altura de seu conjunto de índices. Supondo que o número máximo de chaves numa folha seja igual a G, mostre que:

(a) O número mínimo de pares chave/índice armazenados em suas folhas é dado por: $2d^a$, sendo $d = \lceil G/2 \rceil$.

(b) O número máximo de pares chave/índice armazenados em suas folhas é dado por: G^{a+1}.

(c) O número mínimo de chaves no conjunto de índices é dado por: $2d^{a-1} - 1$, sendo $d = \lceil G/2 \rceil$.

(d) O número máximo de chaves no conjunto de índices é dado por: $G^a - 1$.

82. Mostre que o espaço requerido para armazenar uma árvore B+ contendo n registros num arquivo tem custo $\theta(n)$.

83. Mostre que uma busca numa árvore B+ armazenada em arquivo contendo n chaves e com grau G tem custo de transferência $\theta(\log_G n)$.

84. Mostre que a inserção de um par chave/índice numa árvore B+ armazenada em arquivo contendo n chaves e com grau G tem custo de transferência $\theta(\log_G n)$.

85. Mostre que a remoção de um par chave/índice numa árvore B+ contendo n chaves e com grau G tem custo de transferência $\theta(\log_G n)$.

86. Como se calcula a altura de uma árvore B+?

87. (a) Como se encontra a menor chave de uma árvore B+? (b) Como se encontra a maior chave de uma árvore B+?

88. Suponha que as chaves de um arquivo de registros estejam ordenadas em ordem crescente e que elas são acrescentadas na ordem em que se encontram numa árvore B+. Apresente um argumento que comprove que quase todos os nós da árvore resultante serão preenchidos apenas pela metade.

Outras Variantes de Árvores B (Seção 6.6)

89. O que é uma árvore B*?

90. (a) Como uma árvore B* adia uma divisão de nós após uma operação de inserção? (b) Como ocorre essa divisão de nós?

91. Existe diferença entre busca em árvore B* e busca em árvore B?

92. (a) Quais são as vantagens oferecidas por árvores B* com relação a árvores B? (b) Qual é a principal desvantagem de árvores B*?

93. O que é uma árvore B#? (**NB**: Este livro apenas menciona essas árvores. Se for de seu interesse, pesquise sobre elas na internet.)

Comparando Árvores Multidirecionais de Busca (Seção 6.7)

94. Dentre as árvores de busca apresentadas neste capítulo, quais delas possuem as seguintes propriedades:
(a) Divisão de nós
(b) Combinação de nós
(c) Facilidade para inserção massiva
(d) Facilidade para pesquisa de intervalo

95. Em que tipos de programa comumente encontrados no cotidiano, árvores B e suas variantes são utilizadas?

Exemplos de Programação (Seção 6.8)

96. Qual é a melhor maneira de acessar todas as chaves de uma árvore B+ em ordem crescente?

97. (a) A função `CaminhamentoInfixoB()` apresentada na **Seção 6.8.1** pode ser utilizada para efetuar caminhamentos em árvores multidirecionais descendentes de busca? (b) Essa função pode ser usada com a mesma finalidade com árvores B+?

98. (a) A função `MaiorChaveB()` apresentada na **Seção 6.8.2** pode ser usada para encontrar a maior chave, numa árvore multidirecional descendente de busca? (b) Essa função pode ser usada com a mesma finalidade com árvores B+?

99. (a) Explique o raciocínio empregado para calcular a altura de uma árvore B. (b) Esse mesmo raciocínio pode ser usado para calcular a altura de uma árvore B+?

100. Por que o raciocínio utilizado para calcular a altura de uma árvore B apresentada na **Seção 6.8.5** não pode ser utilizado para calcular a altura de uma árvore multidirecional descendente de busca?

101. Qual é o custo de transferência da função `CaminhamentoInfixoB()` apresentada na **Seção 6.8.1**?

102. Qual é o custo de transferência da função `MenorChaveB()` apresentada na **Seção 6.8.2**?

103. Qual é o custo de transferência da função `NumeroDeNosB()` apresentada na **Seção 6.8.3**?

104. Qual é o custo de transferência da função `NumeroDeChavesB()` apresentada na **Seção 6.8.4**?

105. Qual é o custo de transferência da função `AlturaB()` apresentada na **Seção 6.8.5**?

6.10 Exercícios de Programação

EP6.1 (a) Escreva uma função em C que calcula a altura de uma árvore multidirecional de busca descendente. (b) Escreva uma função em C que calcula a altura de uma árvore B+. (c) Qual dessas funções é mais difícil (i.e., mais complexa) de implementar e por quê?

EP6.2 Escreva uma função em C que retorna a menor chave armazenada numa árvore multidirecional de busca do tipo apresentado na **Seção 6.1**.

EP6.3 Escreva uma função em C que retorna a maior chave armazenada numa árvore multidirecional de busca do tipo apresentado na **Seção 6.1**.

EP6.4 Existe outra abordagem para inserção em árvores B que consiste numa divisão prévia enquanto se faz uma busca. Isto é, quando se faz uma busca, cada nó completo é dividido, de modo que divisões de nós não se propagam para cima. Implemente uma função similar à função `InsereB()`, apresentada na **Seção 6.4.5**, que implemente a abordagem descrita aqui.

EP6.5 A função `BuscaIntervaloBM()`, apresentada na **Seção 6.8.6**, coleta num arquivo todas as chaves que estão entre duas chaves especificadas de uma árvore B+. Numa aplicação prática (e não meramente didática), faria mais sentido coletar os registros (e não apenas as chaves). Implemente uma extensão dessa função que coleta num arquivo todos os registros cujas chaves se encontram entre duas chaves especificadas de uma árvore B+.

EP6.6 Escreva uma função que escreve num arquivo de texto todas as chaves de uma árvore B+ em ordem crescente usando os tipos e constantes definidos na **Seção 6.5.6**.

EP6.7 (a) Usando os tipos e constantes definidos na **Seção 6.5.6**, escreva uma função que encontra a menor chave numa árvore B+. (b) Usando esses mesmos tipos e constantes, escreva uma função que encontra a maior chave numa árvore B+.

EP6.8 Usando os tipos e constantes definidos na **Seção 6.4.5**, escreva uma função que encontra a chave de uma árvore B que é sucessora imediata de uma chave recebida como parâmetro.

EP6.9 Usando os tipos e constantes definidos na **Seção 6.4.5**, escreva uma função que encontra a chave de uma árvore B que é antecessora imediata de uma chave recebida como parâmetro.

EP6.10 Usando os tipos e constantes definidos na **Seção 6.5.6**, escreva (a) uma função que determina o número de nós internos de uma árvore B+ e (b) uma função que determina o número de folhas de uma árvore B+.

EP6.11 Usando os tipos e constantes definidos na **Seção 6.4.5**, escreva uma função que encontra o piso de uma chave que se encontra numa árvore B.

EP6.12 Usando os tipos e constantes definidos na **Seção 6.4.5**, escreva uma função que encontra o teto de uma chave que se encontra numa árvore B.

EP6.13 Escreva uma única função que calcula, ao mesmo tempo, o número de nós e o número de chaves de uma árvore B. Utilize os tipos e constantes definidos na **Seção 6.4.5**.