Universidade Federal de São Carlos
Campus Sorocaba

Bacharelado em Ciência da Computação
Estruturas de Dados 2

Árvores-B e suas operações

1 Introdução

Árvores-B são estruturas de dados usadas para armazenamento e recuperação eficiente de valores. A estrutura de uma árvore-B é uma generalização do conceito de árvores binária de pesquisa balanceadas, na qual os nós da árvore podem armazenar vários valores simultaneamente e ramificar para várias sub-árvores. Dessa maneira, a capacidade de armazenamento de cada nó não fica restrita a um valor fixo, oferecendo maior liberdade para que o desenvolvedor distribua os valores entre os nós da forma mais apropriada para a resolução de um dado problema. Essa característica tornou a árvore-B muito popular em aplicações que envolvem indexação de grandes volumes de dados, uma vez que permite otimizar o acesso a memória secundária.

Entre as principais aplicações de árvores-B, podemos citar os sistemas gerenciadores de bancos de dados, que utilizam essa estrutura de dados para indexar informação armazenada no sistema de arquivos de seus servidores. Praticamente todos os sistemas comerciais implementam alguma variação da árvore-B, confirmando seu alto índice de eficiência em operações que envolvem a troca de dados entre as memórias primária e secundária.

A introdução do conceito de árvore-B é atribuído a Bayer e McCreight, no trabalho intitulado *"Organization and Maintenance of Large Ordered Indexes"*, publicado em 1972 [1]. A escolha da terminologia *"Árvore-B"* ainda é desconhecida, abrindo margem para algumas hipóteses não confirmadas. Alguns autores sugerem que o termo faz referência ao laboratório no qual os autores estavam vinculados à época da publicação do primeiro trabalho sobre o tema (*Boeing Scientific Research Laboratories*). Há quem afirme que a árvore-B seja uma referência ao trabalho de Rudolf Bayer, o primeiro autor do trabalho citado anteriormente. Outra possibilidade seria que o termo indicasse uma árvore balanceada, indicando uma de suas características mais marcantes. Seja lá qual for a origem do termo *"Árvore-B"*, o que realmente se sabe é que os trabalhos de Bayer e McCreight são considerados um marco importante na Ciência da Computação, principalmente na área de Banco de Dados, uma vez que fundamentou muitas otimizações nos algoritmos usados em sistemas de armazenamento e recuperação de dados.

Este texto tem, como objetivo principal, apresentar as principais características de uma árvore-B, bem como seus algoritmos de busca e de inserção. As informações descritas nas seções seguintes são uma compilação dos capítulos correlatos dos livros de Cormen *et al.* [2], Drozdek [3] e Ziviani [4].

2 Árvore-B

A árvore-B, a exemplo da árvore binária de pesquisa, utiliza uma característica particular dos valores armazenados, de modo a acelerar a busca pelos mesmos elementos em operações futuras. Essa característica é a relação de ordem existente entre um dado valor e os demais valores armazenados na árvore. Por este motivo, a primeira providência a ser tomada antes de construir uma árvore-B é garantir que os valores armazenados na árvore-B mantem, entre si, uma relação de *ordem parcial*. Chamaremos esses valores, no restante do texto, de chaves.

Um dos diferenciais da árvore-B é que seus nós podem armazenar várias chaves ao mesmo tempo. Para garantir um nível razoável de armazenamento de chaves dos nós, definiremos uma

medida de preenchimento dos nós, que chamaremos de **ordem** da árvore-B. A ordem da árvore-B é a quantidade máxima de filhos que um nó não-folha pode armazenar.

Seja $m > 2$ um número natural. Uma árvore-B de ordem m é uma árvore de busca com as seguintes propriedades (acompanhe pela Figura 1):

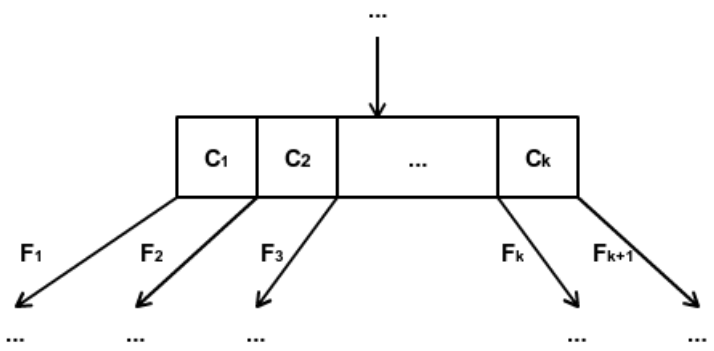

Figura 1: Organização de um nó interno: disposição de chaves e referências para nós-filhos.

1. Cada página tem:
   - no máximo, m descendentes e $m - 1$ registros
   - no mínimo $\lceil m/2 \rceil$ descendentes (exceto raiz e folhas)
2. A raiz tem, no mínimo, dois descendentes
   - A não ser que seja uma folha (único nó da árvore)
3. Todas as folhas estão no mesmo nível
4. Uma página não folha com k descendentes contém $k - 1$ chaves
5. Uma página folha contém, no mínimo $\lceil m/2 \rceil - 1$ e, no máximo, $m - 1$ chaves

São grandes vantagens da árvore-B:

- É uma árvore perfeitamente balanceada;
- A quantidade de chaves em cada nó é flexível (apesar de fixa), permitindo que desenvolvedores possam adaptá-la de acordo com suas necessidades. Esta característica, em particular, é que tornou a árvore-B muito popular em sistemas com gerenciamento de memória secundária;
- Quanto maior a quantidade de chaves em cada nó, menor é a altura da árvore-B, permitindo estratégias para acelerar operações de pesquisa.

Neste trabalho exploraremos apenas as operações de busca e de inserção de chaves. Antes de apresentarmos os algoritmos para as operações citadas, é oportuno mostrar o significado de algumas expressões usadas:

- $raiz[T]$: raiz da árvore T;
- $n[X]$: número de chaves do nó X;
- $c_i[X]$: a i-ésima chave do nó X (considerando uma ordenação não-decrescente), $1 \leq i < m$;
- $f_i[X]$: o i-ésimo filho do nó X, $1 \leq i \leq m$. Para nós folha, assumiremos que $f_i[X]$ é $NULO$;
- $folha[X]$: valor booleano que indica se o nó X é uma folha (verdadeiro) ou não (falso).

3 Busca de chaves na árvore-B

O objetivo de um algoritmo de busca é verificar a presença (ou ausência) de uma determinada chave entre todas as chaves armazenadas na árvore-B. Para realizar esta tarefa, é útil usar a ordenação das chaves nos nós internos da árvore. O algoritmo apresentado a seguir busca recursivamente por uma chave, devolvendo um nó que contém essa chave ou $NULO$, se chave não estiver na árvore.

O Algoritmo 1 segue a mesma lógica dos algoritmos de busca em árvores binárias de pesquisa. Primeiro, busca pela chave procurada entre as chaves armazenadas em um determinado nó da árvore, nas linhas 2–5. Nessa posição, o índice i indica a posição na qual a chave procurada deveria estar no nó X. Se a chave estiver na posição i, então o algoritmo devolve X e finaliza, linhas 7–9. Se a chave não estiver em i, então ela deve ser procurada em algum filho. Se não houver filhos para procurar, então a chave não está na árvore, e o algoritmo devolve $NULO$ (linha 12). Se houver filhos, repita o algoritmo no nó-filho (linha 14).

Algoritmo 1 BUSCA (X, k)

Entrada: Nó não-vazio X e uma chave k
Saída: Nó X que possui a chave k

```
1:
2: i ← 1
3: while i ≤ n[X] e k > c_i[X] do
4:     i ← i + 1
5: end while
6:
7: if i ≤ n[X] e k = c_i[X] then
8:     return X
9: end if
10:
11: if folha[X] then
12:     return NULO
13: else
14:     return BUSCA(f_i[X], k)
15: end if
```

4 Inserção de chaves na árvore-B

A inserção de chaves é uma operação usada para incluir novas chaves na árvore-B. A inserção em uma árvore-B, comparada à inserção em árvores binárias de pesquisa, pode ser considerada complexa, uma vez que envolve tratamentos especiais para manter as propriedades da árvore. Nesta seção, apresentaremos um algoritmo de inserção de chaves em uma árvore-B, comentando suas principais características. Os passos realizados estão ilustrados na Figura 2, que foi adaptada de Bayer e McCreight [1].

A inserção de uma chave sempre a direciona para um nó-folha. A operação de inserção procura o nó-folha mais apropriado para receber a nova chave. Quando houver espaço para incluir a chave, ela é colocada no conjunto de chaves da folha e o algoritmo termina. A situação muda quando a chave deve ser colocada em um nó-folha cheio, ou seja, com exatamente $m - 1$ chaves. Nesse caso, dizemos que ocorreu um *overflow* e teremos que fazer alterações na estrutura da árvore para abrir espaço para a nova chave. A alteração da árvore se processa por meio de uma operação que chamaremos *divisão do nó*, onde um nó cheio é "dividido" em dois nós preenchidos até a metade.

A divisão de um nó é a peça-chave do algoritmo de inserção e está ilustrado pelo Algoritmo 2. É bom salientar que é vista semanticamente como a divisão de um nó, porém sua implementação é construída por meio da criação de um novo nó na árvore e movimentação de metade das chaves do nó cheio para esse novo criado, inicialmente vazio.

O algoritmo recebe três parâmetros: o nó X, a chave k e o nó $filho\_direito$, que será $NULO$

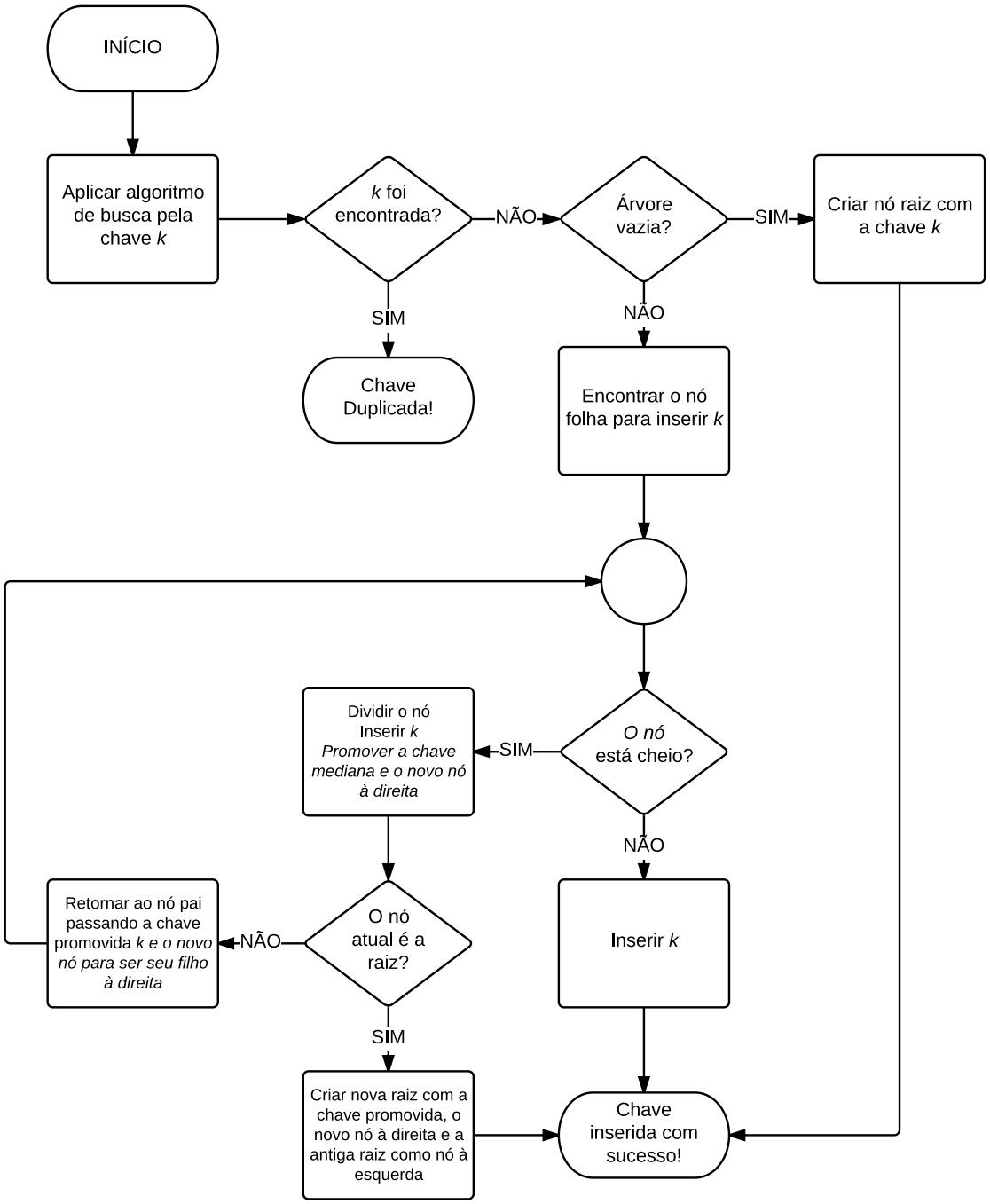

Figura 2: Operação de inserção.

quando X for folha. A criação do novo nó Y ocorre nas linhas 5–7 e a movimentação de metade das chaves de X para este nó nas linhas 9–19. Caso k não tenha sido alocado em Y (quando k é menor que as demais chaves), X deve ser manipulado para abrir espaço para a nova chave, linhas 21–29.

Os apontadores para os filhos das chaves também são movimentados, linhas 12, 16, 24, 28 e 32, para quando a divisão ocorre em um nó não-folha (quando nós internos também sofrem *overflow*). Quando a divisão ocorrer em um nó folha, todos os descendentes serão $NULO$, inclusive o parâmetro $filho\_direito$, que é utilizado nas linhas 12 e 28.

A chave mediana é promovida (linha 31) e o número de chaves de X é reduzido pela metade

Algoritmo 2 DIVIDE_NO $(X, k, filho\_direito)$

Entrada: Nó não-vazio X, uma chave k e um nó $filho\_direito$ que pode ser nulo
Saída: Chave promovida e o novo nó Y

```
 1:
 2: i ← n[X]
 3: chave_alocada ← FALSO
 4:
 5: Y ← CRIA_NO()
 6: folha[Y] ← folha[X]
 7: n[Y] ← ⌊(m − 1)/2⌋
 8:
 9: for j ← n[Y] decresce até 1 do
10:     if !chave_alocada e k > c_i[X] then
11:         c_j[Y] ← k
12:         f_{j+1}[Y] ← filho_direito
13:         chave_alocada ← VERDADEIRO
14:     else
15:         c_j[Y] ← c_i[X]
16:         f_{j+1}[Y] ← f_{i+1}[X]
17:         i ← i − 1
18:     end if
19: end for
20:
21: if !chave_alocada then
22:     while i ≥ 1 e k < c_i[X] do
23:         c_{i+1}[X] ← c_i[X]
24:         f_{i+2}[X] ← f_{i+1}[X]
25:         i ← i − 1
26:     end while
27:     c_{i+1}[X] ← k
28:     f_{i+2}[X] ← filho_direito
29: end if
30:
31: chave_promovida ← c_{⌊m/2⌋+1}[X]
32: f_1[Y] ← f_{⌊m/2⌋+2}[X]
33: n[X] ← ⌊m/2⌋
34:
35: return chave_promovida, Y
```

(linha 33). É importante dizer que quando a ordem da árvore-B é par, a chave promovida será a menor chave do nó direito, que por sua vez ficará com uma chave a menos do que o nó esquerdo.

A Figura 3 apresenta um exemplo no qual podemos ver o comportamento da inserção quando o nó está completamente cheio. Neste exemplo, temos uma árvore de ordem 5, completamente cheia em sua configuração original (Figura 3a). Sobre essa árvore, deseja-se inserir a chave 13. O procedimento padrão é procurar pelo nó-folha mais adequado para a inserção da chave. No exemplo, este nó-folha não tem espaço para incluir a chave, de modo que ele será dividido em dois nós, com distribuição de suas chaves (Figura 3b). Sempre que ocorre uma divisão, a chave mediana é inserida no nó-pai. Neste exemplo, vemos um caso particular, em que o nó pai também está cheio, de modo que o processo de divisão se repete no nível superior também (Figura 3c). Mais uma vez, a chave mediana é inserida no nível superior. Como não há nenhum nó no nível superior, é criado um novo nó, que passará a ser a raiz da árvore. Ao final, verifica-se que a árvore-B, após a inserção da chave 13, cresceu em altura.

A chamada do Algoritmo 2 é parte do algoritmo de inserção de chaves de um determinado nó. Seu papel é simplesmente dividir um nó cheio e retornar a chave promovida e o novo nó criado. O Algoritmo 3, a seguir, é responsável por encontrar o nó-folha correto para a inserção e tratar

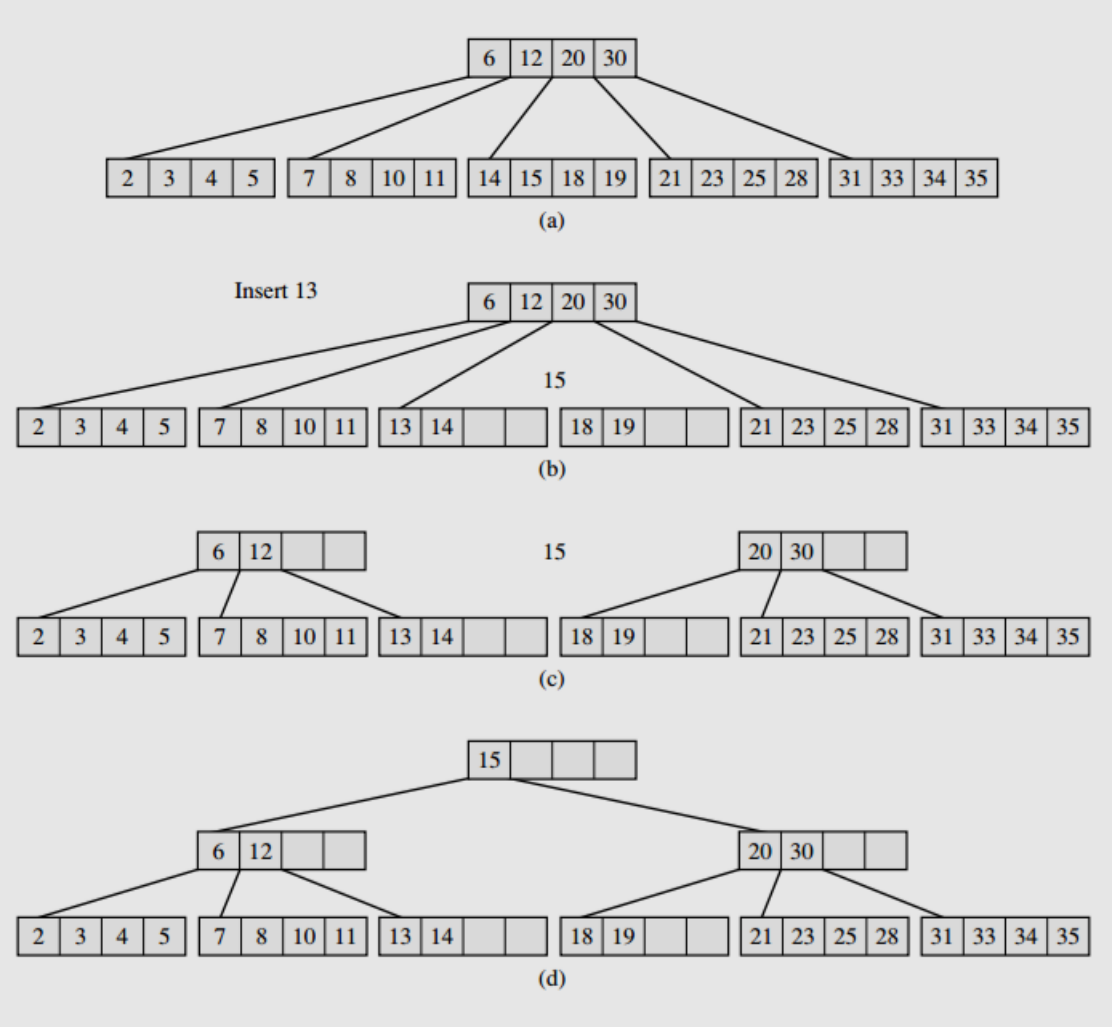

Figura 3: Exemplo de uma inserção em um nó cheio (retirado de [3]).

overflow. Os parâmetros do Algoritmo 3 são um nó X e uma chave k, de modo que deseja-se que k seja adicionado à sub-árvore de raiz X. Sua atuação pode ser dividida em duas situações: X é folha ou X não é folha.

Se X é uma folha, é preciso verificar se há espaço suficiente para a inclusão de k. Se existe, simplesmente adiciona a chave k em X, abrindo espaço entre as chaves de X se necessário, como mostram as linhas 3–12. O algoritmo retorna $NULO$ para indicar que não houve *overflow*. Por outro lado, quando não há espaço para a inclusão de k, o algoritmo de divisão de nó é chamado, linha 14, passando o argumento $filho\_direito$ como $NULO$.

Caso X não seja folha, a chave k deve ser direcionada para algum nó-folha. Assim, o algoritmo transfere a operação de inserção para seu nó-filho mais adequado, linhas 17–22. A chamada recursiva (linha 22) garante a continuidade do algoritmo até que um nó-folha seja alcançado.

Quando os retornos da chamada recursiva, $chave\_promovida$ e $filho\_direito$, são $NULOS$, quer dizer que não houve *overflow* e nada precisa ser feito. Por outro lado, quando houve *overflow*, é preciso verificar se há espaço suficiente para a inclusão da $chave\_promovida$. Se existe, simplesmente adiciona a $chave\_promovida$ em X, abrindo espaço entre as chaves de X se necessário, como mostram as linhas 26–37. Dessa vez, os apontadores para os filhos também são movimentados, pois X não é folha. Por outro lado, quando não há espaço para a inclusão de $chave\_promovida$, o algoritmo de divisão de nó é chamado, linha 39, repassando o argumento $filho\_direito$.

Uma vez que temos um algoritmo de inserção de uma chave k em um nó X de uma árvore-B, estamos preparados para construir a função global de inserção de uma chave k na árvore-B, que

Algoritmo 3 INSERE_AUX (T, k)

Entrada: Nó não-vazio X e uma chave k
Saída: Chave promovida e o nó $filho\_direito$ (quando o *overflow* é propagado)

```
 1:
 2: if folha[X] then
 3:     if n[X] < m − 1 then
 4:         i ← n[X]
 5:         while i ≥ 1 e k < c_i[X] do
 6:             c_{i+1}[X] ← c_i[X]
 7:             i ← i − 1
 8:         end while
 9:         c_{i+1}[X] ← k
10:         n[X] ← n[X] + 1
11:
12:         return NULO, NULO
13:     else
14:         return DIVIDE_NO(X, k, NULO)
15:     end if
16: else
17:     i ← n[X]
18:     while i ≥ 1 e k < c_i[X] do
19:         i ← i − 1
20:     end while
21:     i ← i + 1
22:     chave_promovida, filho_direito ← INSERE_AUX(f_i[X], k)
23:
24:     if chave_promovida ≠ NULO then
25:         k ← chave_promovida
26:         if n[X] < m − 1 then
27:             i ← n[X]
28:             while i ≥ 1 e k < c_i[X] do
29:                 c_{i+1}[X] ← c_i[X]
30:                 f_{i+2}[X] ← f_{i+1}[X]
31:                 i ← i − 1
32:             end while
33:             c_{i+1}[X] ← k
34:             f_{i+2}[X] ← filho_direito
35:             n[X] ← x[X] + 1
36:
37:             return NULO, NULO
38:         else
39:             return DIVIDE_NO(X, k, filho_direito)
40:         end if
41:     else
42:         return NULO, NULO
43:     end if
44: end if
```

chamaremos T. O Algoritmo 4 mostra como essa operação é realizada.

A operação realizada pelo algoritmo é, basicamente, inserir uma chave k a partir de sua raiz. Se a árvore ainda não possuir um nó raiz, ele é criado e a chave k é inserida como a primeira da árvore, linhas 2–8. Se o nó-raiz existir, o Algoritmo 3 é chamado e todo o processo recursivo se inicia, até que se alcance um nó-folha. Se o Algoritmo 3 retornar que houve *overflow*,, é preciso criar uma nova raiz com a $chave\_promovida$, o $filho\_direito$ e a antiga raiz como filho esquerdo, linhas 12–21.

Algoritmo 4 INSERE (T, k)

Entrada: Árvore-B T e uma chave k

```
1:
2: if raiz[T] = NULO then
3:     X ← CRIA_NO()
4:     folha[X] ← VERDADEIRO
5:     n[X] ← 1
6:     c1[X] ← k
7:
8:     raiz[T] ← X
9: else
10:    chave_promovida, filho_direito ← INSERE_AUX(raiz[T], k)
11:
12:    if chave_promovida ≠ NULO then
13:        X ← CRIA_NO()
14:        folha[X] ← FALSO
15:        n[X] ← 1
16:        c1[X] ← chave_promovida
17:
18:        f1[X] ← raiz[T]
19:        f2[X] ← filho_direito
20:
21:        raiz[T] ← X
22:    end if
23: end if
```

Podemos acompanhar o processo global de inserção em uma árvore-B a partir da Figura 4. Nela, consideramos que a árvore-B possui ordem 5 e encontra-se inicialmente vazia. A inserção das chaves 8, 14, 2 e 15 preenche o nó raiz da árvore (Figura 4a). A próxima inserção, da chave 3, provocará uma divisão de nó e consequente crescimento da altura da árvore (Figura 4b). O processo de inserção segue como de costume, sempre procurando posições disponíveis nas folhas. As próximas inserções, das chaves 1, 16 e 6 mantém a mesma configuração da árvore. A inserção da chave 5 produz uma nova divisão da folha mais à esquerda (Figura 4c), aumentando o número de chaves da raiz. Nesse caso, como a raiz não estava cheia, não houve crescimento da altura da árvore. O mesmo ocorre na inserção de 27 e 37 (Figura 4d). A inserção das chaves 18, 25, 7, 13 e 20 preenche todas as posições da raiz da árvore (Figura 4e). Nas inserções seguintes (22, 23 e 24), provocam uma divisão na raiz, aumentando a altura da árvore (Figura 4f).

As inserções de chaves, em uma árvore-B, seguem sempre essa rotina, buscando posições vazias nas folhas e preenchendo-as. Quando as folhas não possuem posições disponíveis, elas são divididas, produzindo novas folhas com mais espaço para futuras inserções, sempre respeitando o limite mínimo de $\lceil m/2 \rceil - 1$. Cada divisão de nó produz, como efeito colateral, a inserção de uma chave no nó pai, que também se divide quando está completamente cheio. Em última análise, a árvore-B aumenta de altura somente quando uma chave é inserida em uma folha e todos os nós no caminho entre a raiz e essa folha estão cheios.

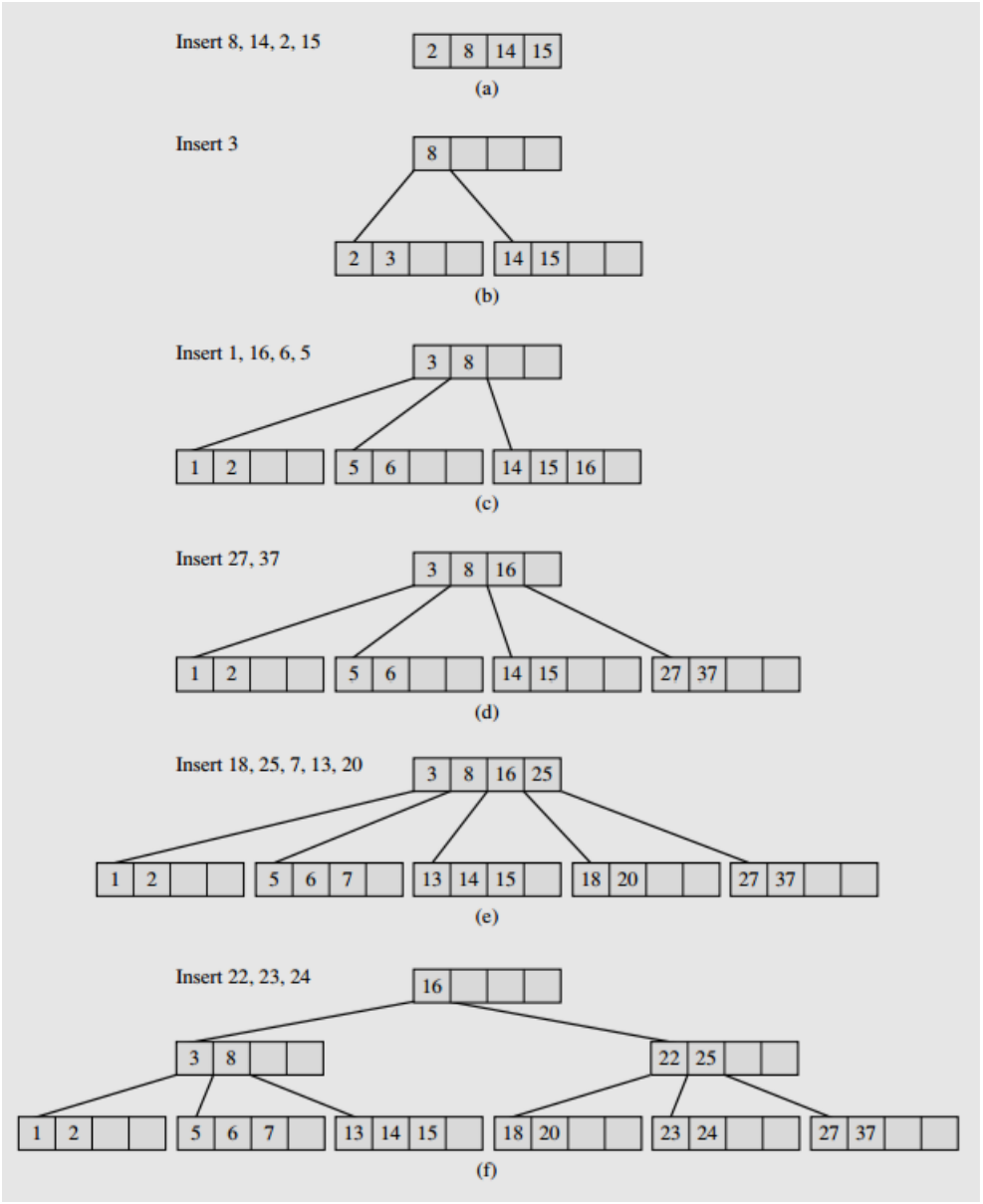

Figura 4: Exemplo de inserção de chaves em uma árvore-B (retirado de [3]).

Referências

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Algoritmos - Teoria e Prática.* Campus, 2002.

[3] A. Drozdek. *Estruturas de Dados e Algoritmos em C++.* Cengage, 2009.

[4] N. Ziviani. *Projeto de Algoritmos com Implementação em Java e C++.* Thomson, 2006.