Estruturas de Dados II

Árvore B: propriedades, busca e inserção

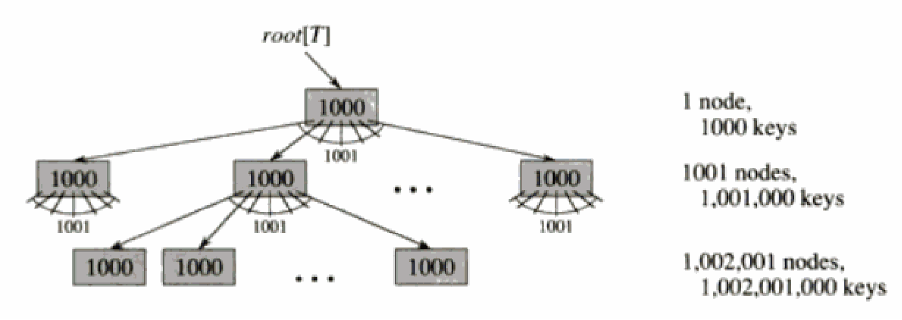
Prof[a]. Juliana de Santi
Prof. Rodrigo Minetto
Universidade Tecnológica Federal do Paraná
Material compilado de: Cormen, Notas de aula IC-UNICAMP, IME-USP

Sumário

Árvores B: **são árvores de pesquisa balanceadas**, projetadas para minimizar operações de E/S de disco. Muitos sistemas de banco de dados usam árvores B ou variações (**B+**) para armazenar informações (funcionam bem em discos magnéticos).

A árvore B foi criada em 1972 por **Bayer** e **Mc-Creight**. Foi desenvolvida no *Boeing Scientific Research Labs*. A razão da letra B é desconhecida. **Árvores B** **diferem** significativamente das árvores AVL (e Rubro-Negras) pelo fato de que seus nós podem ter **muitos filhos**, desde uma dezena até milhares.

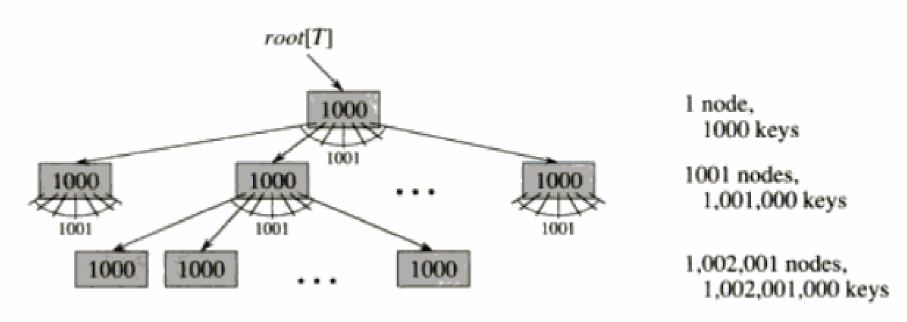
Exemplo: uma árvore B com **altura 2** contendo **mais de 1 bilhão de chaves**.



Introdução

Fatores de **ramificação** (grau da árvore) **entre 50 e 2000** são usados com frequência. Um grande fator de ramificação reduz drasticamente tanto a altura da árvore quanto o número de acessos ao disco necessários para encontrar qualquer chave.

Como o nó raiz pode ser mantido permanentemente na memória principal, apenas **2** acessos ao disco são exigidos para encontrar qualquer chave nessa árvore.



Árvores B **assemelham árvores AVL** (e Rubro-Negras) no fato de que toda árvore B de n nós tem altura $\mathcal{O}(\log n)$ (embora essa altura possa ser consideravelmente menor devido a seu fator de ramificação).

Sumário

Árvores B - Definição

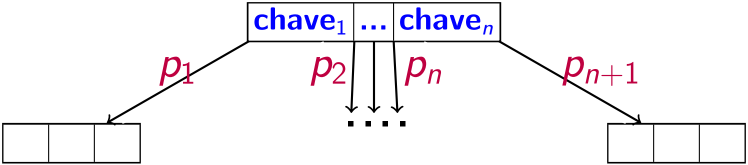
Propriedades de uma árvore **B**:

1) Todo nó x tem os seguintes campos:

- $n[\mathbf{x}]$, o número de chaves atualmente armazenadas no nó **x**;
- **as próprias $n[\mathbf{x}]$ chaves**, armazenadas em ordem *crescente*;
- **folha[x]**, um booleano que é **true** se **x** é folha e **false** se **x** é um nó interno.

Propriedades de uma árvore **B**:

2) Cada nó interno **x** também contém $n[\mathbf{x}] + 1$ **ponteiros** $p_1[\mathbf{x}], p_2[\mathbf{x}], \ldots, p_{n[\mathbf{x}]+1}[\mathbf{x}]$ para seus filhos. Os nós folhas não têm filhos, e assim seus campos p_i são indefinidos.

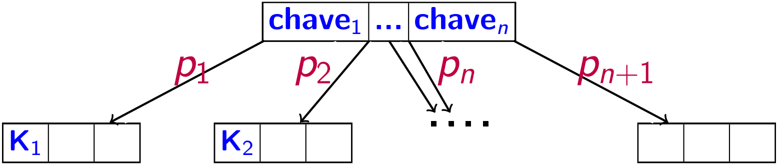


Árvores B - Definição

Propriedades de uma árvore **B**:

3) As chaves $chave_i[\mathbf{x}]$ **separam** os **intervalos de chaves** armazenadas **em cada subárvore**: se k_i é qualquer chave armazenada na subárvore com raiz $chave_i[\mathbf{x}]$, então:

$$k_1 \leq chave_1[\mathbf{x}] \leq k_2 \leq chave_2[\mathbf{x}] \leq \ldots$$



Propriedades de uma árvore **B**:

4) Sua construção garante que **toda folha** tem a **mesma profundidade** (mesmo nível), que é a altura h da árvore.

5) Existem **limites** inferiores e superiores sobre o número de chaves que um nó pode conter. Esses limites são expressos em termos de um inteiro fixo $t \geq 2$ chamado **grau mínimo** da árvore B.

Árvores B - Definição

Propriedades de uma árvore **B**:

5a) Todo nó diferente da raiz deve ter pelo menos $t - 1$ chaves. Assim, todo **nó interno** diferente da raiz tem pelo menos t filhos. Uma árvore não vazia tem raiz com pelo menos uma chave.

5b) Todo nó pode ter no máximo $2t - 1$ chaves, assim um nó interno tem no máximo $2t$ filhos.

Sumário

Sumário

Árvore B

Uma árvore **B** pode ser definida pela **estrutura**:

```
#define T 2
typedef struct _node {
  int n;            /*Número de chaves!*/
  int folha;        /*Booleano*/
  int chaves[2 * T - 1];
  struct _node *filhos[2 * T];
} No, Arvore;
```

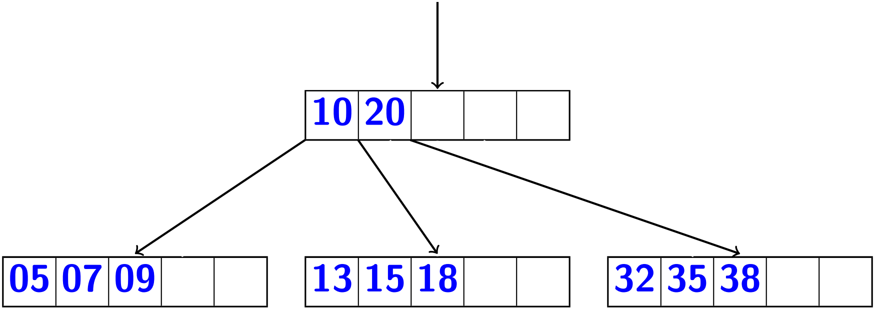
Sumário

Árvores B - Busca

Pesquisar em uma árvore B é **semelhante** a pesquisar em uma **árvore binária**, exceto que ao invés de uma bifurcação em cada nó, tomamos uma decisão de ramificação de várias vias, de acordo com o número de filhos do nó (generalização).

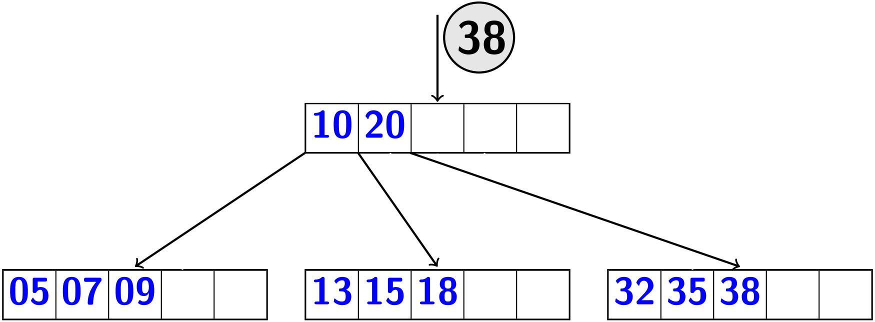
Árvores B - Busca

Suponha uma busca pelo elemento: **‘38’**



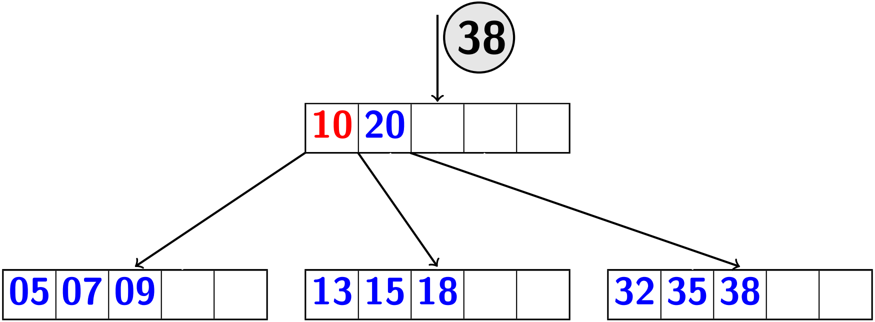
Árvores B - Busca

Suponha uma busca pelo elemento: **‘38’**



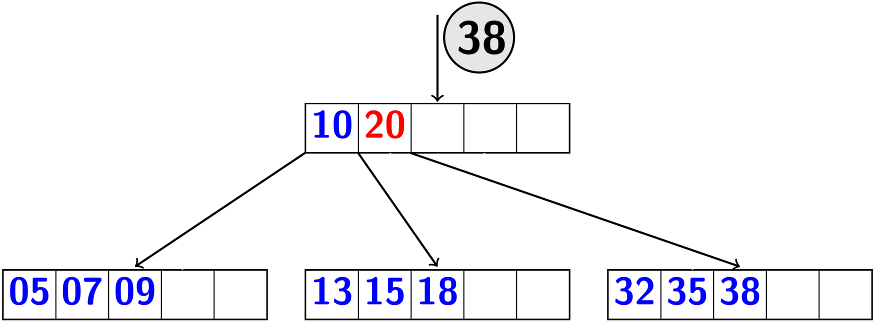
Árvores B - Busca

Suponha uma busca pelo elemento: **‘38’**

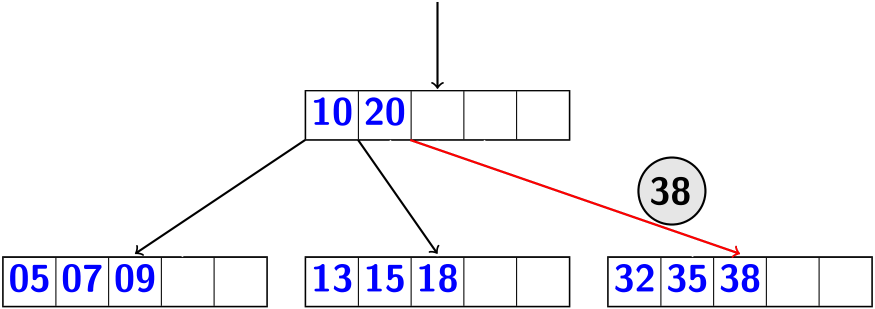


Árvores B - Busca

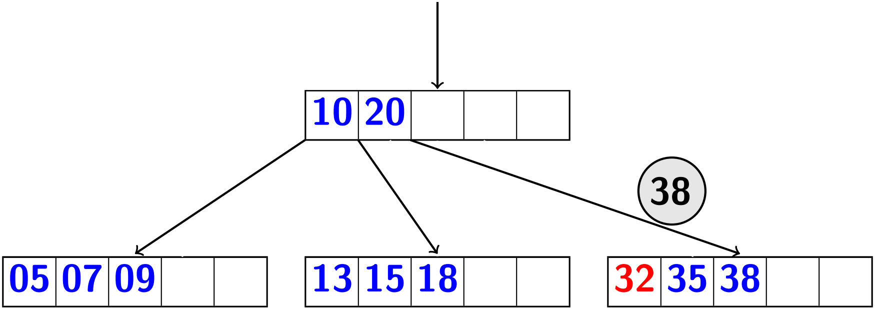
Suponha uma busca pelo elemento: **‘38’**



Suponha uma busca pelo elemento: **'38'**

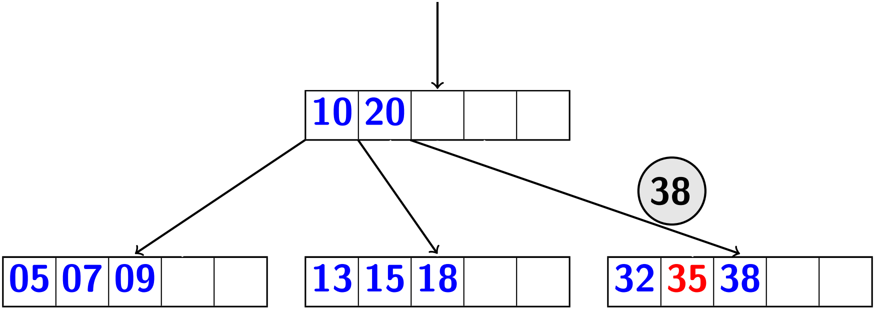


Suponha uma busca pelo elemento: **'38'**

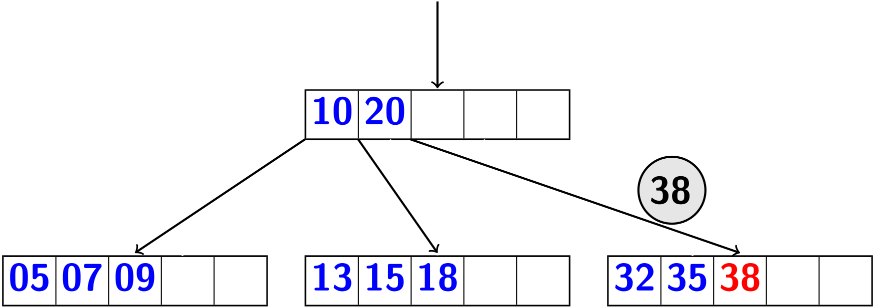


Árvores B - Busca

Suponha uma busca pelo elemento: **‘38’**

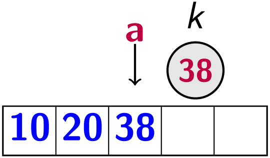


Suponha uma busca pelo elemento: **‘38’**



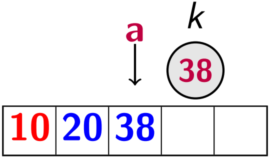
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



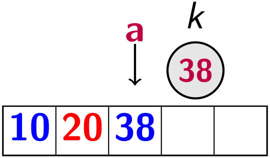
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



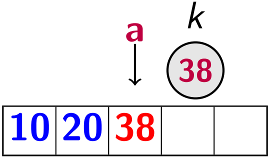
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



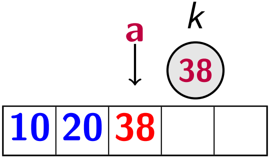
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



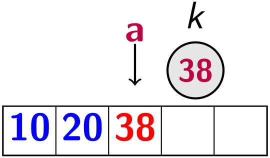
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



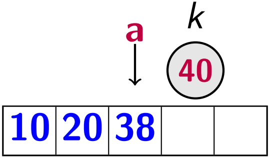
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



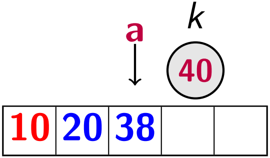
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



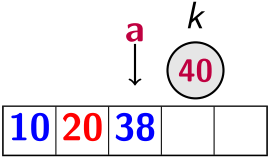
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



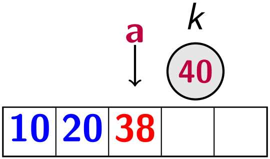
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



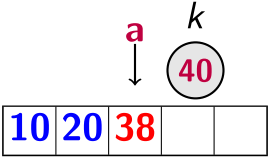
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



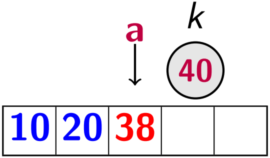
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



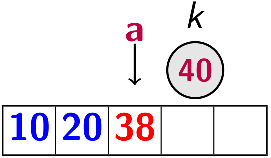
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



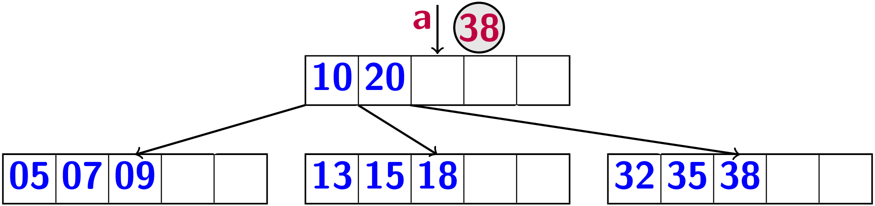
Busca (Arvore *a, Chave k = **40**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



Busca (Arvore *a, Chave k = **38**)

$i = 0$

while ($i < $ **a→n** && $k >$ **a→chave[i])**

    $i = i + 1$**;**

if (($i <$ a→n) && ($k ==$ a→chave[i]))

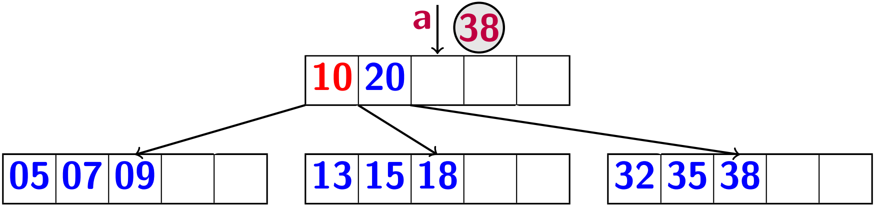
    return FOUND;

else if (a→folha)

    return NOT_FOUND;

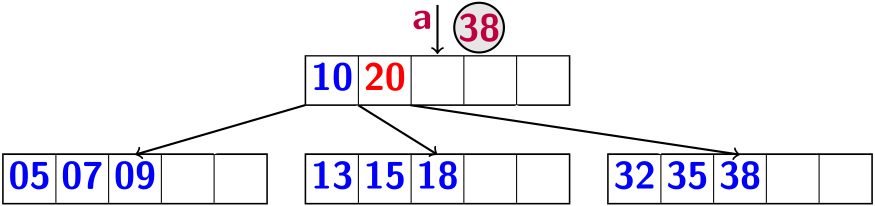
else

    return **Busca** (a→filhos[i], k);



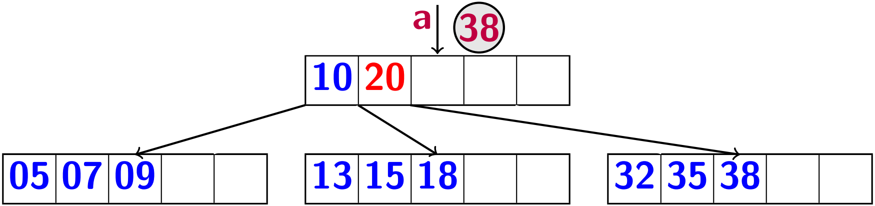
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



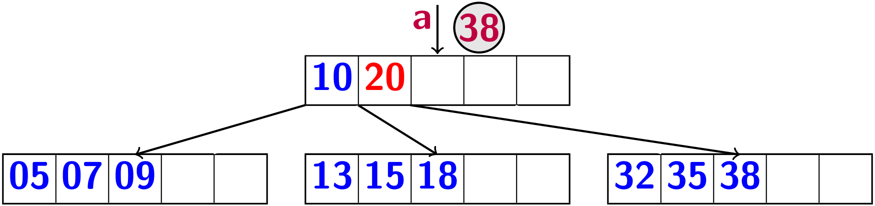
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while (i < a→n && k > a→chave[i])
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



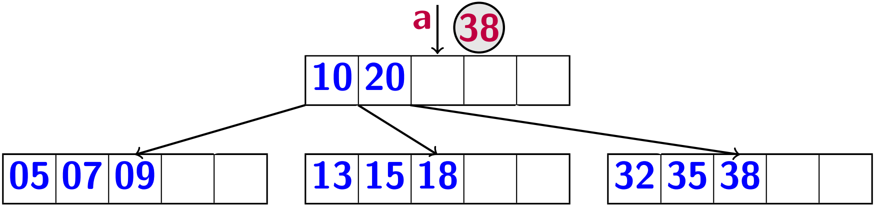
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



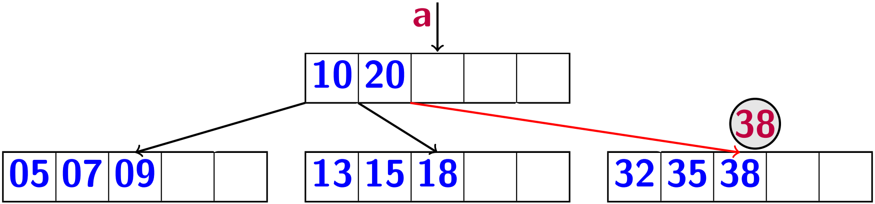
Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



Busca (Arvore *a, Chave k = **38**)

```
i = 0
while ( (i < a→n) && (k > a→chave[i]) )
    i = i + 1;
if ( (i < a→n) && (k == a→chave[i]) )
    return FOUND;
else if (a→folha)
    return NOT_FOUND;
else
    return Busca (a→filhos[i], k);
```



Complexidade: para cada nó, tem-se uma busca linear em **t** chaves (dado que o máximo de chaves é $2t-1$), então tem-se um custo $\mathcal{O}(\mathbf{t})$. A busca através dos nós tem custo em função da altura **h** da árvore. Logo, a busca na árvore B tem custo $\mathcal{O}(\mathbf{th})$.

Sumário

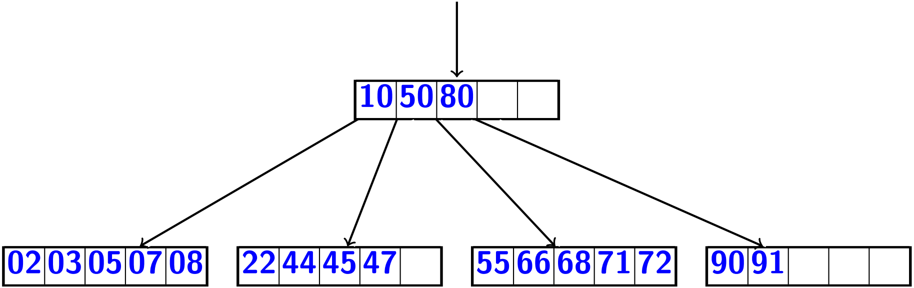
Árvores B - Inserção

Inserção: a operação de inserção em uma árvore B é relativamente mais complicada, pois, precisamos inserir a nova chave no nó correto da árvore **sem violar** suas propriedades. Mas como proceder caso o nó já esteja **cheio**? Ou seja, já tem $2t - 1$ **chaves**.

Árvores B - Inserção

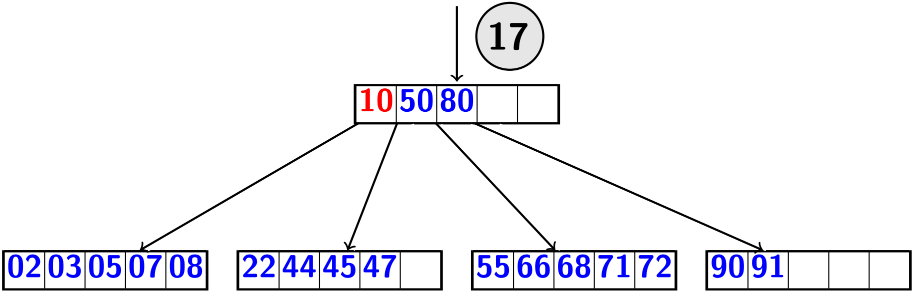
Devemos **separar** o nó cheio ao redor de sua chave **mediana**, **criando dois novos nós** que não violam as definições da árvore. O elemento **mediano é** então **promovido**, passando a fazer parte do nó pai daquele nó. A **chave é inserida** em um dos novos nós respeitando as definições da árvore.

Suponha a inserção do elemento: **'17'**



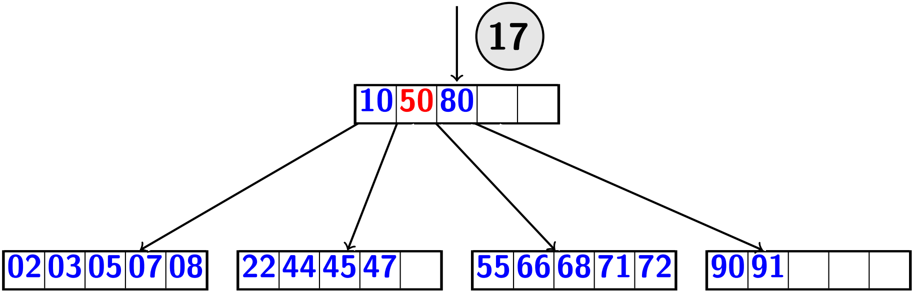
Árvores B - Inserção

Suponha a inserção do elemento: **'17'**



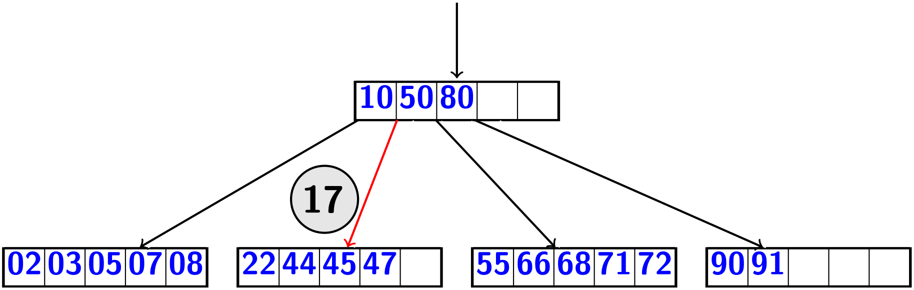
Árvores B - Inserção

Suponha a inserção do elemento: **‘17’**



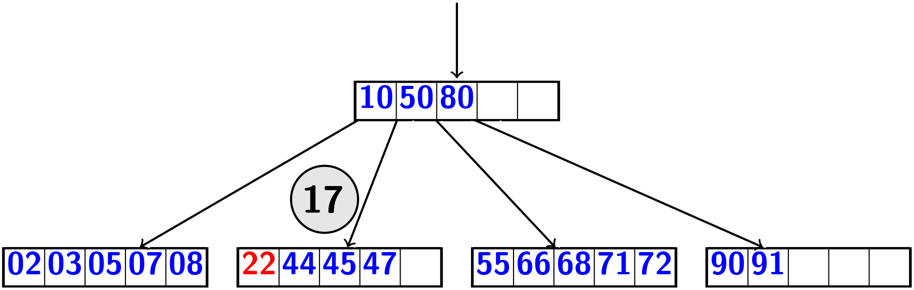
Árvores B - Inserção

Suponha a inserção do elemento: **'17'**



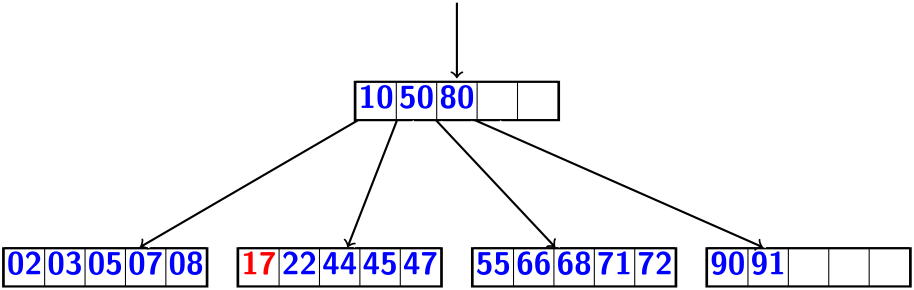
Árvores B - Inserção

Suponha a inserção do elemento: **'17'**

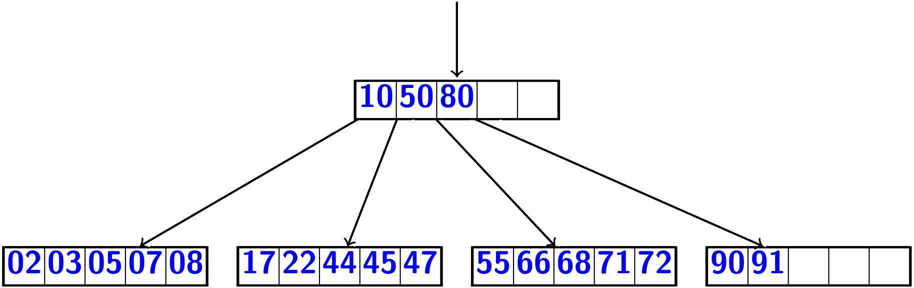


Árvores B - Inserção

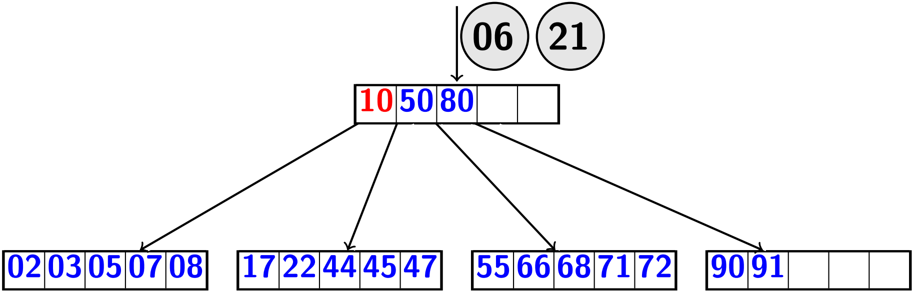
Suponha a inserção do elemento: **‘17’**



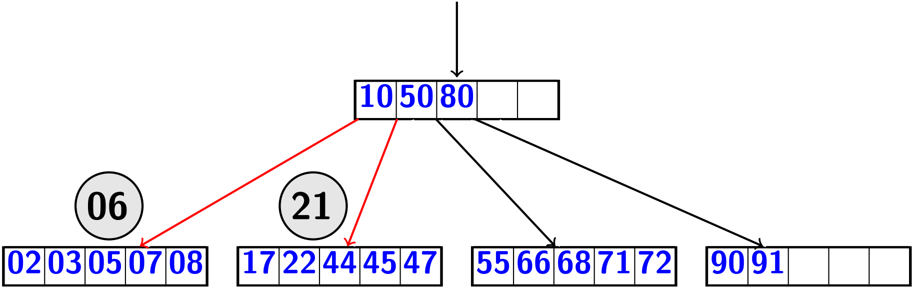
Suponha a inserção dos elementos: **‘06’** e **‘21’**



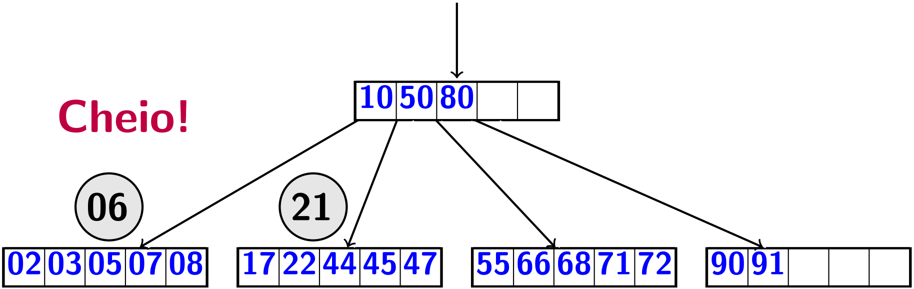
Suponha a inserção dos elementos: **‘06’** e **‘21’**



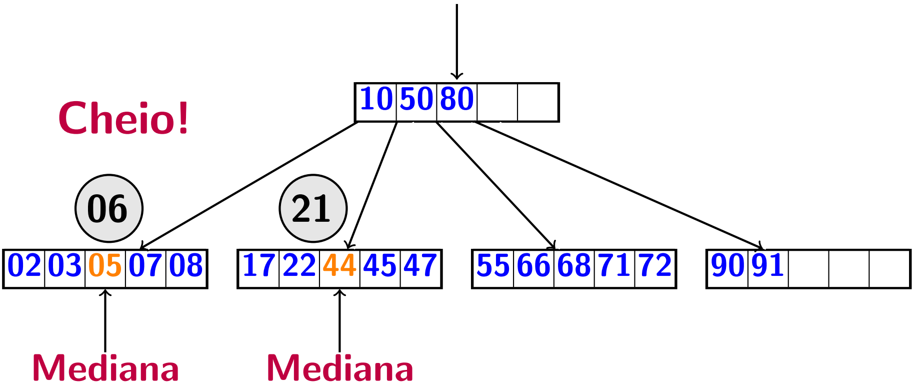
Suponha a inserção dos elementos: **‘06’** e **‘21’**



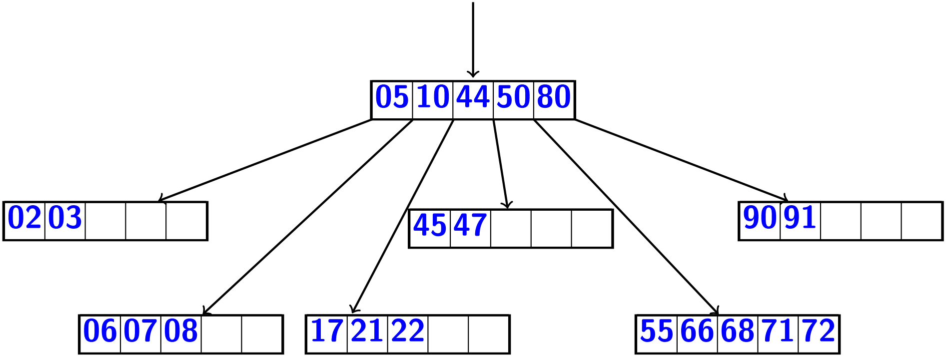
Suponha a inserção dos elementos: **‘06’** e **‘21’**



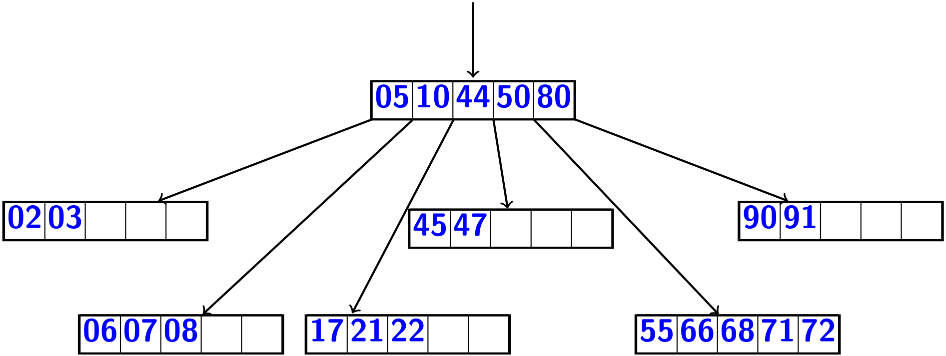
Divisão do nó completo



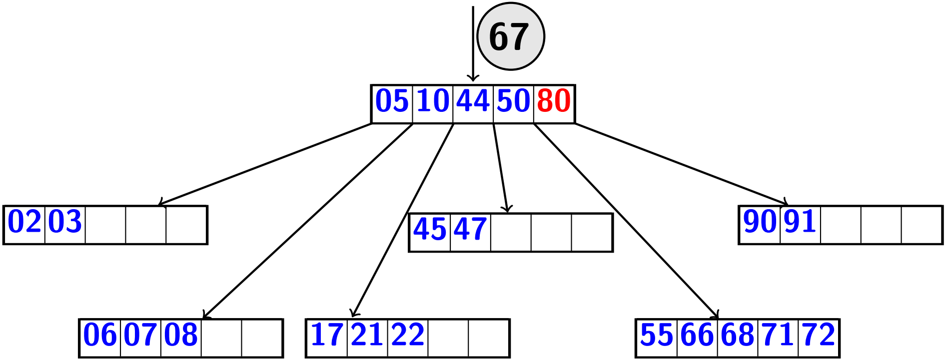
Árvores B - Inserção



Suponha a inserção do elemento: **‘67’**

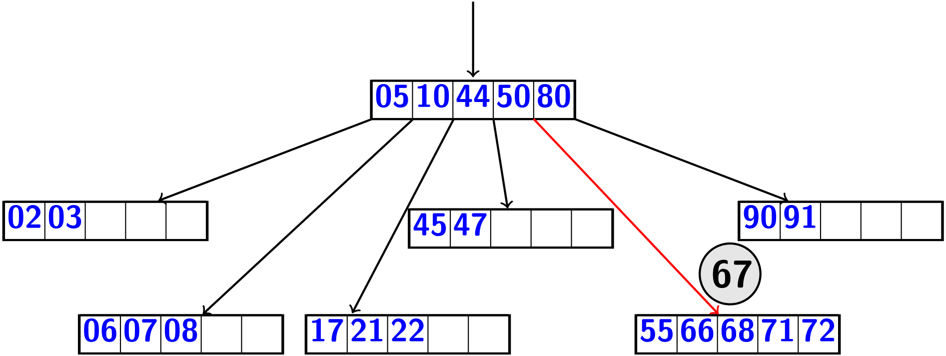


Suponha a inserção do elemento: **'67'**

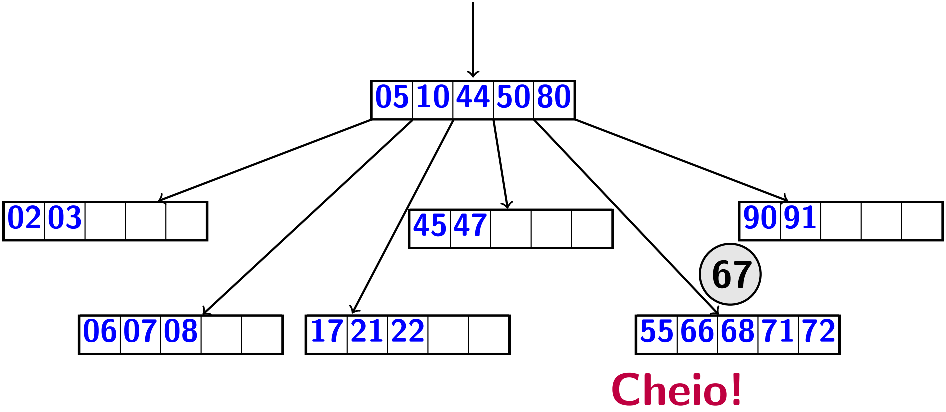


Árvores B - Inserção

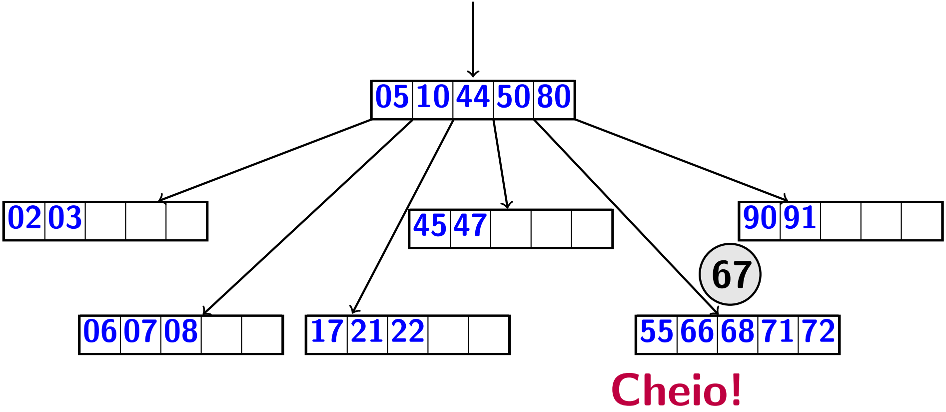
Suponha a inserção do elemento: **'67'**



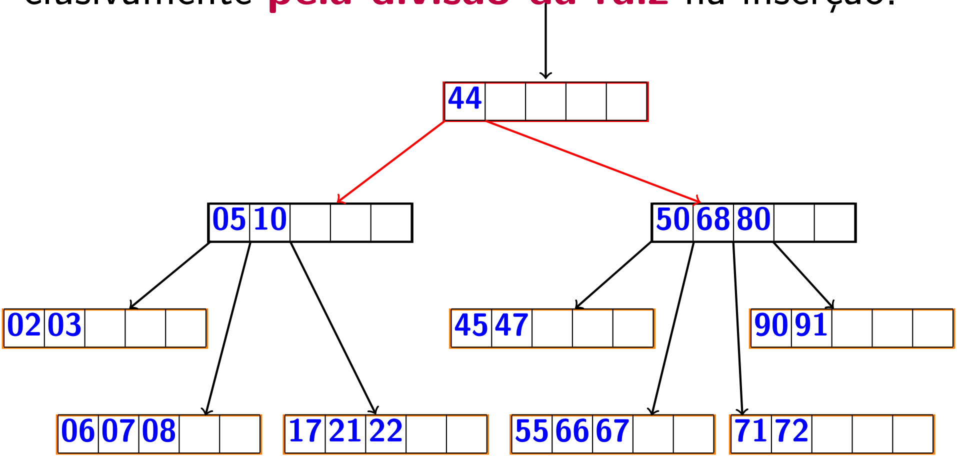
Suponha a inserção do elemento: **'67'**



Exercício: mostre a árvore após a inserção.



A **altura** de uma árvore B **aumenta** única e exclusivamente **pela divisão da raiz** na inserção.



Complexidade: a inserção de uma chave k em um árvore B de altura **h** é feita em uma única passagem descendente na árvore, exigindo $\mathcal{O}(\mathbf{h})$ acessos ao disco. A ordenação dos elementos é da ordem de $\mathcal{O}(\mathbf{t})$. As divisões dos nós podem se propagar até a raiz, exigindo tempo da ordem de $\mathcal{O}(\mathbf{th})$. Portanto a operação de inserção tem um custo de $\mathcal{O}(\mathbf{th})$.

Sumário

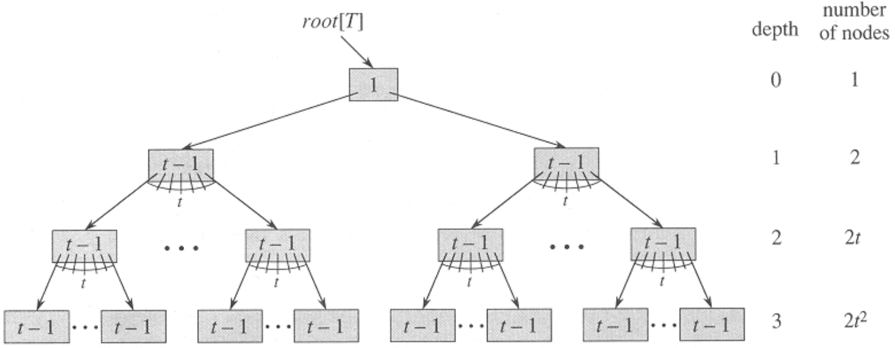
Árvores B - Altura

O número de acessos ao disco exigidos para a maioria das operações em uma árvore B é proporcional à altura da árvore B.

Pergunta: qual a **altura máxima** h de uma árvore B com n nós no **pior caso?**

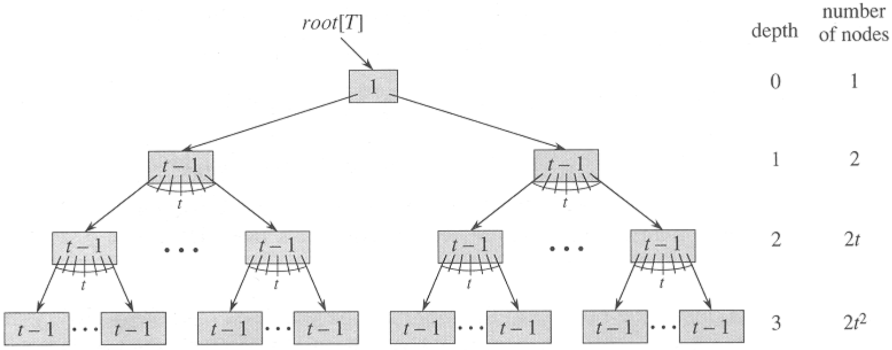
Árvores B - Altura

Prova: seja T uma árvore B de altura h, com $n \geq 1$ nós (não vazia) e grau mínimo $t \geq 2$.



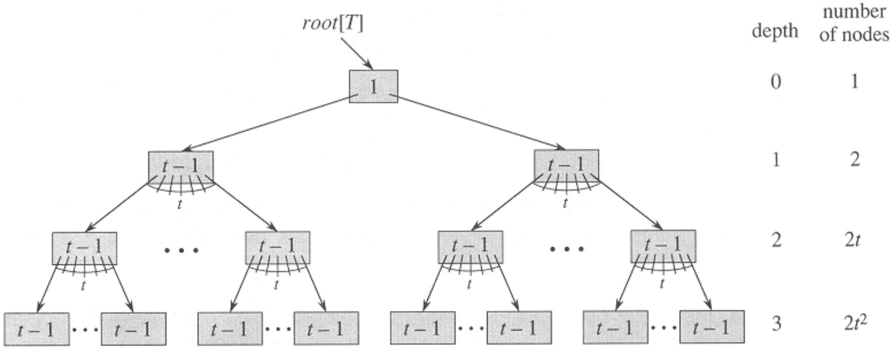
Árvores B - Altura

O número de nós é minimizado, em uma árvore B, quando a raiz contém uma chave e todos os outros nós contêm $t - 1$ chaves.



Árvore B - Altura

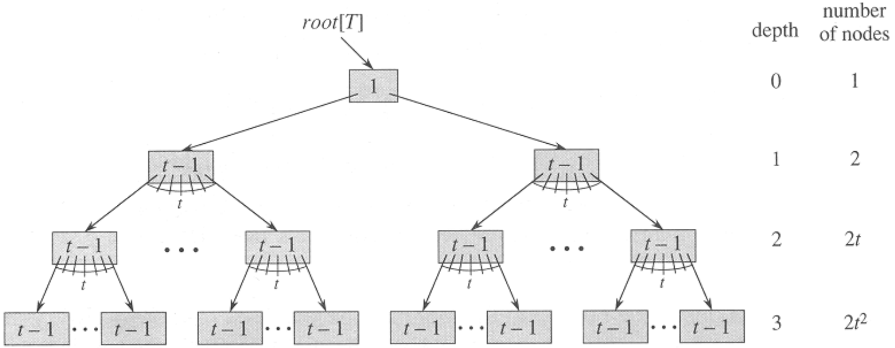
Nesse caso, existem 2 nós na profundidade 1, $2t$ nós na profundidade 2, $2t^2$ nós na profundidade 3, e $2t^{h-1}$ nós na profundidade h. Então há $\sum_{i=1}^{h} 2t^{i-1}$ nós nesta àrvore B de altura h.



Árvore B - Altura

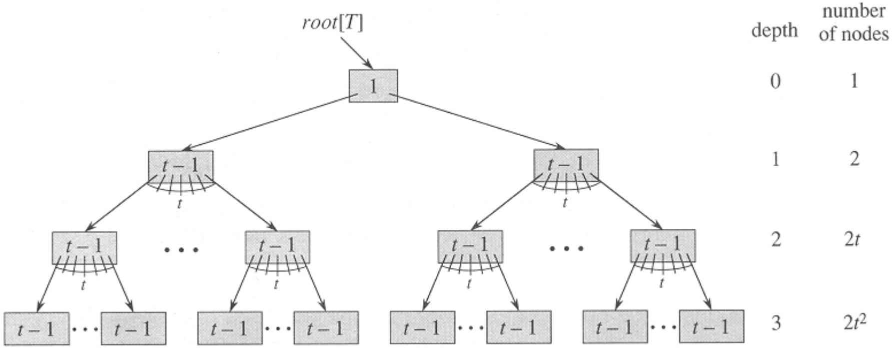
Logo, o número de chaves neste árvore é:

$1+(t-1)\sum_{i=1}^{h} 2t^{i-1}$: cada nó tem $(t-1)$ chaves; e $+1$ pois a raíz tem uma chave.



Árvore B - Altura

Desse modo, o número n de chaves satisfaz a desigualdade: $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$



Árvore B - Altura

Por definição (A.5 Cormen):

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \tag{1}$$

Ex:

$$\sum_{k=0}^{4} 2^k = 2^0 + 2^1 + 2^2 + 2^3 + 2^4$$

$$= 1 + 2 + 4 + 8 + 16 = 31$$

Com a fórmula fechada (1) temos: $\frac{2^5-1}{2-1} = 31$

Árvore B - Altura

No caso da altura temos:

$$\sum_{i=1}^{h} t^{i-1} = t^0 + t^1 + t^2 + ... + t^{h-1}$$

Então de (1) temos:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \rightarrow \frac{t^h - 1}{t - 1}$$

Árvore B - Altura

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$n \geq 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

$$n \geq 2t^h - 1$$

$$(n+1)/2 \geq t^h$$

$$\log_t((n+1)/2) \geq \log_t t^h$$

$$\log_t((n+1)/2) \geq h$$

Árvore B - Altura

Essa prova **demonstra a capacidade das árvores B**, quando comparadas a árvores AVL e Rubro-Negras. Embora a **altura** da árvore **cresça** na proporção $\mathcal{O}(\log n)$ em ambos os casos, **para as árvores B, a base do logaritmo pode ser muitas vezes maior**.

Árvore B - Altura

Dessa modo, as **árvores B poupam** um fator de aproximadamente $\mathbf{\log} t$ sobre as árvores AVL e Rubro-Negras **no número de nós examinados** para a maioria das operações de árvores.

Árvore B - Altura

Tendo em vista que o exame de um nó arbitrário em uma árvore normalmente exige um acesso ao disco, o **número de acessos ao disco é substancialmente reduzido**.

Sumário

Referências

[1] Algoritmos: Teoria e prática. Cormen.