ALGORITMOS E ESTRUTURAS DE DADOS II

Árvores B

Prof. Rafael Fernandes Lopes

http://www.dai.ifma.edu.br/~rafaelf

rafaelf@ifma.edu.br

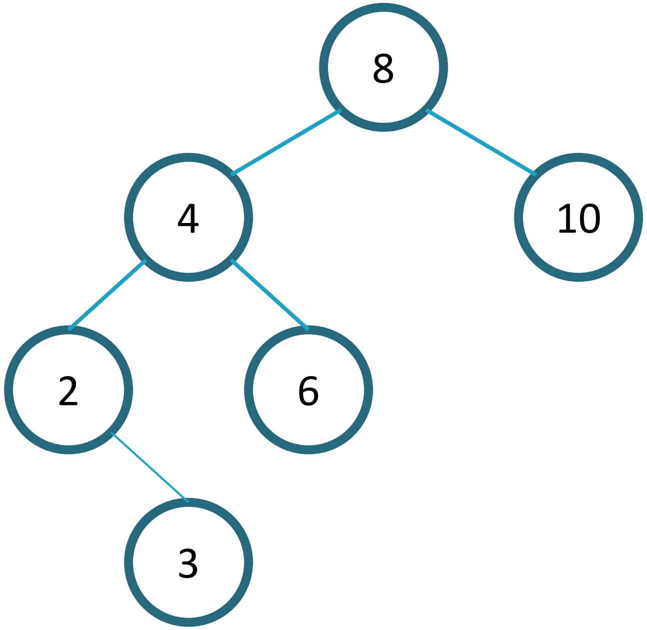
Motivação

- Memória dos sistemas de computadores consistem largamente em duas partes:
  - Memória primária: utiliza chips de memória de silício
  - Memória secundária: baseada em discos magnéticos
- Discos magnéticos são baratos e tem grande capacidade
- Porém, são lentos, pois possuem partes mecânicas que se movimentam
- Precisamos de meios eficientes de acesso aos dados (que minimizem o número de acessos ao disco)
  - Acesso a um disco de 7.200 RPM leva em média 8,33 ms → quase cinco ordens de magnitude mais lento que os tempos de acesso de 100 ns encontrados em memória de silício

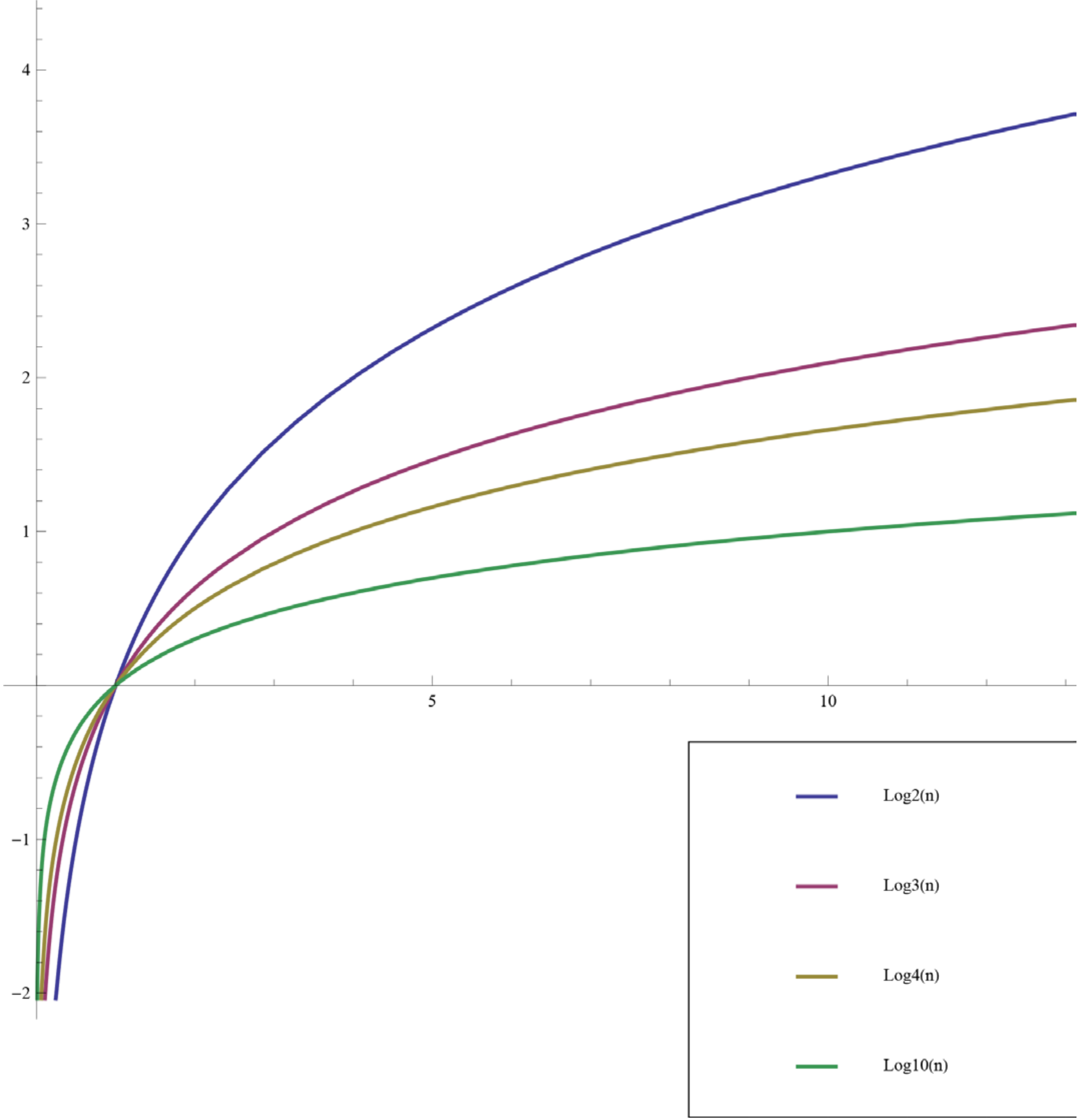
O que são Árvores B

- Árvores B são árvores de busca balanceadas: altura = O(log(n)) no pior caso
- Elas foram projetadas para trabalhar bem com dispositivos de armazenamento secundário e acesso direto (discos magnéticos)
- Similares às Árvores Rubro-Negras, mas apresentam melhor desempenho em operações de disco de E/S
- Árvores B (e suas variantes como a B^+ e a B^*) são largamente utilizadas em sistemas de bancos de dados

- Número de acessos em uma árvore binária com n nós:
  $h = \log_2(n)$



- Como reduzir o número de acessos?
  - Reduzindo a altura!
  - Como???
  - $\text{Log}_2(n) \geq \text{Log}_3(n) \geq \text{Log}_4(n) \geq \ldots \geq \lim_{k \to \infty} \text{Log}_k(n)$, $n \geq 1$



4
3
2
1
−1
−2
5
10
Log2(n)
Log3(n)
Log4(n)
Log10(n)

O que são Árvores B

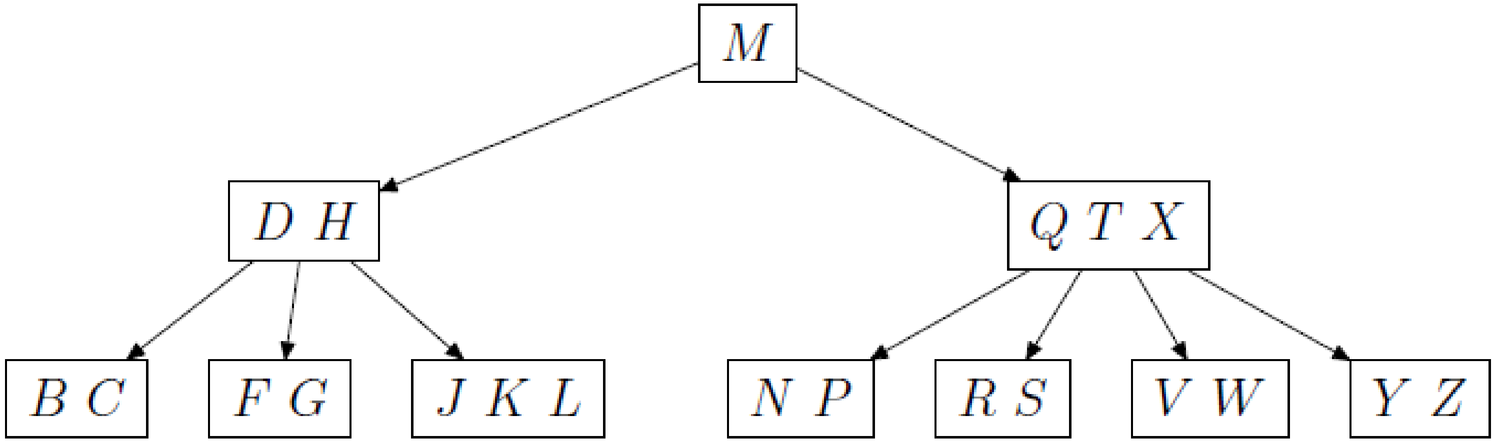
- Proposto em 1972 por Bayer e McCreight, desenvolvido no Laboratório de Pesquisas Científicas Boeing
    - Não se sabe se o B é de Bayer ou Boeing
- Usado no armazenamento em memória secundária
- É uma árvore n-ária

Características

- O que afeta o desempenho de uma árvore é a altura
- A altura diminui a medida que a aridade aumenta
- Uma árvore de ordem m tem uma aridade m, onde cada nó pode ter m filhos
- Todas as folhas estão no mesmo nível

Um exemplo

- As 21 consoantes do alfabeto como chaves de uma Árvore B:



- Todo nó interno x contém n[x] chaves e tem n[x]+1 filhos
- Todas as folhas tem a mesma profundidade na árvore

Definição

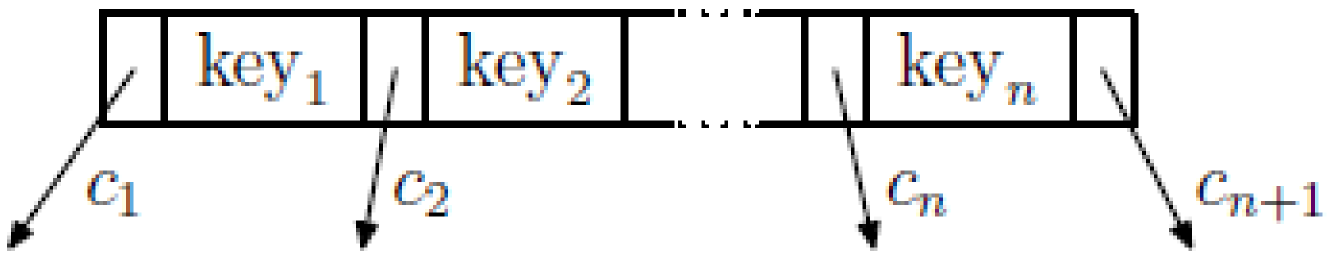
- Uma árvore B *T* é uma árvore (com raiz *raiz[T]*) com as seguintes propriedades:
  - Todo nó *x* tem quatro campos
    1. O número de chaves que são armazenados em *x*, *n[x]*
    2. As *n[x]* chaves são armazenadas em ordem crescente:

       $chave_1[x] \leq chave_2[x] \leq \ldots \leq chave_{n[x]}[x]$
    3. Um valor booleano

       folha[x] = { TRUE, se *x* é uma folha, FALSE, se *x* é um nó interno }
    4. *n[x]*+1 ponteiros $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ para seus filhos (os nós folhas não tem filhos e seus campos c_i são indefinidos)

Definição

- Ponteiros e chaves de um nó



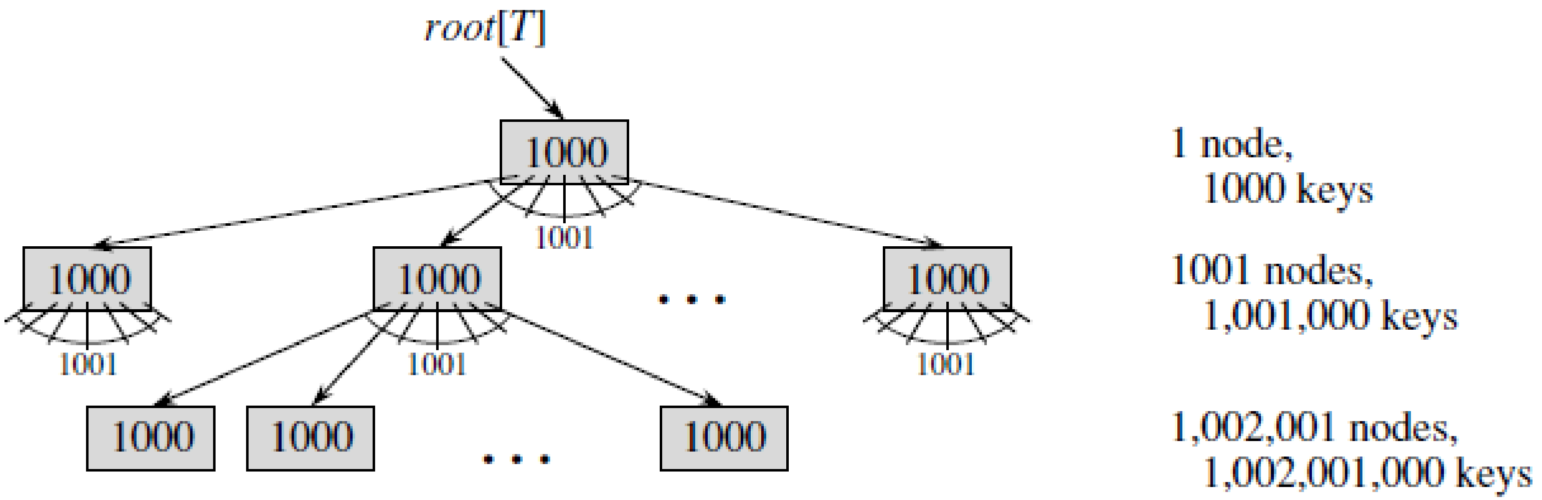


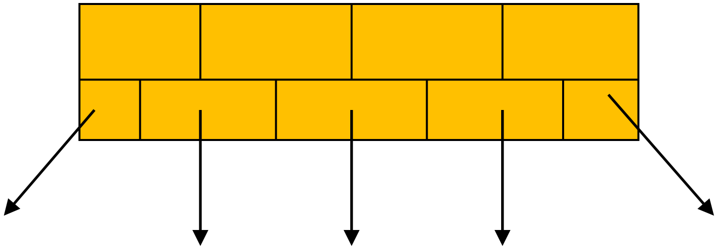
Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Definição

- Propriedades (cont.)
  - As chaves *$chave_i[x]$* separam os intervalos de chaves armazenadas em cada sub-árvore:
    - Se k_i é qualquer chave armazenada na sub-árvore com raiz $c_i[x]$, então

      $$k_1 \le chave_1[x] \le k_2 \le chave_2[x] \le \ldots \le k_{n[x]} \le chave_{n[x]+1}[x]$$
  - Todas as folhas têm a mesma altura, que é a altura da árvore, *h*
  - Existem limitantes superiores e inferiores para o número de chaves em um nó
    - A especificação desses limitantes utiliza um inteiro fixo $t \ge 2$, o **grau mínimo** da árvore B
    - Limitante inferior: todo nó diferente da raiz deve ter pelo menos $t\text{-}1$ chaves (e t filhos)
    - Limitante superior: cada nó pode conter no máximo $2t\text{-}1$ chaves (e $2t$ filhos)

- Nó com *t = 5*

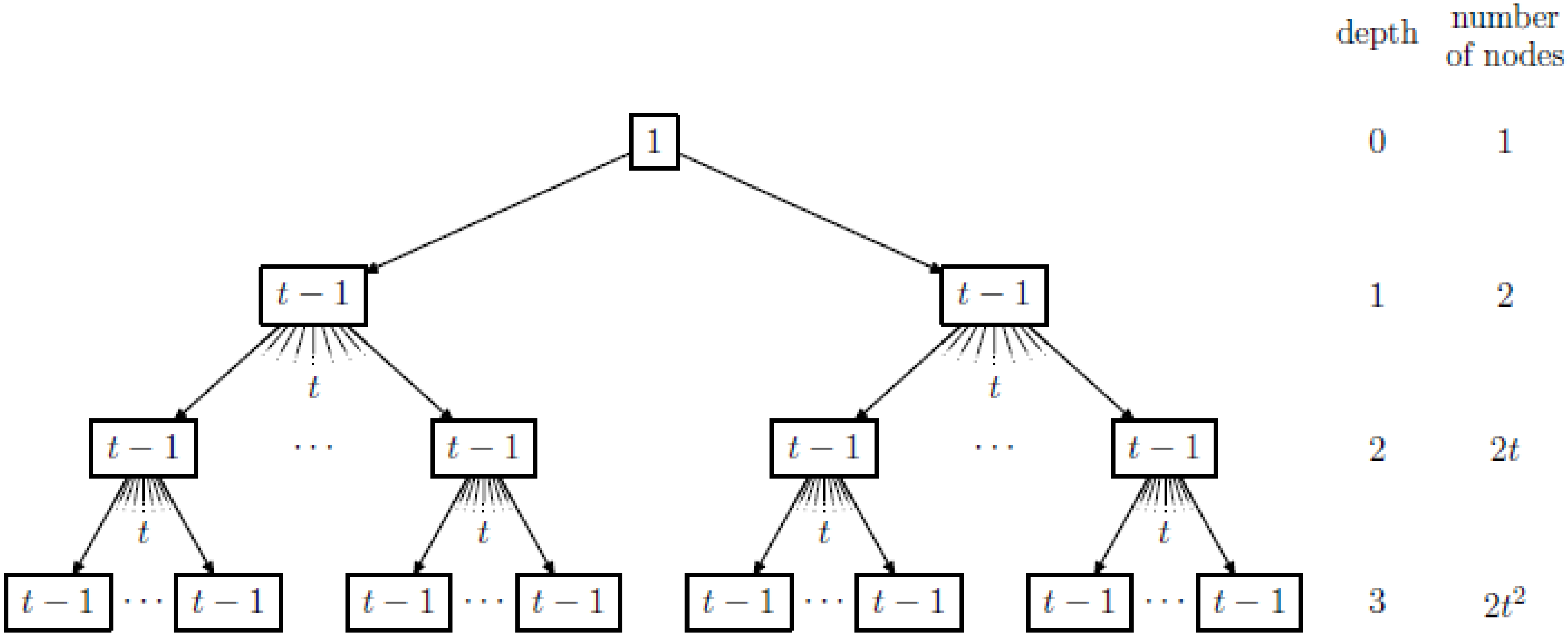


Nó de uma árvore B de ordem = 5

```
typedef struct NO_Btree_ {
  int               n;
  boolean           folha;
  Tipo              chave[4];
  struct NO_Btree_  c[5];
} NO_Btree;
```

Altura de uma Árvore B

- Exemplo (pior caso): uma Árvore B contendo o número mínimo possível de chaves



- Dentro de cada nó *x* é apresentado o número de chaves *n[x]* contidas

Altura de uma Árvore B

- Número de acesso ao disco é proporcional à altura da Árvore B
- No pior caso: $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$

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

$$t^h \leq \frac{(n+1)}{2}$$

$$h \leq \log_t\left(\frac{n+1}{2}\right) \sim O(\log_t n)$$

- Principal vantagem das Árvores B comparadas às rubro-negras:
  - A base do logaritmo *t* pode ser muito maior
  - Economizam um fator ~ *log(t)* em número de nós examinados em operações na árvore
  - **Número de acessos ao disco é substancialmente reduzido!**

Operações básicas em Árvores B

- Árvores B fornecem as seguintes operações:
  - `B-Tree-Search`
  - `B-Tree-Create`
  - `B-Tree-Insert`
  - `B-Tree-Delete`

- Convenções:
  - A raiz da árvore é sempre mantida na memória principal (operação `Disk-Read` nunca é necessária na raiz)
  - Qualquer nó passado como parâmetro deve ter uma operação `Disk-Read` realizada nele

- Os procedimentos são todos algoritmos de “uma passagem”, que prosseguem em sentido descendente a partir da raiz, sem ter que subir novamente (uma exceção é feita à exclusão)

Pesquisa em uma Árvore B

- 2 entradas:
  - o *x*, um ponteiro para a raiz de uma sub-árvore
  - o *k*, uma chave a ser pesquisada na sub-árvore

function B-TREE-SEARCH(x, k) **returns** (y, i) such that $\text{key}_i[y] = k$ or NIL
    $i \leftarrow 1$
    while $i \leq n[x]$ and $k > \text{key}_i[x]$
        do $i \leftarrow i + 1$
    if $i \leq n[x]$ and $k = \text{key}_i[x]$
        then return (x, i)
    if $\text{leaf}[x]$
        then return NIL
        else DISK-READ$(c_i[x])$
                return B-TREE-SEARCH$(c_i[x], k)$

- Em cada nó interno x, é necessário realizar uma decisão entre *(n[x]+1)* chaves

Pesquisa em uma Árvore B - Complexidade

- Número de páginas do disco acessadas em uma árvore B
  - $\Theta(h) = \Theta(\log_t n)$
- O tempo do laço "while" dentro de cada nó é O(t), portanto o tempo total de CPU gasto é
  - $O(th) = O(t \log_t n)$

Criando uma Árvore B vazia

$$
\begin{array}{l}
\text{B-TREE-CREATE}(T) \\
\quad x \leftarrow \text{ALLOCATE-NODE}() \\
\quad \text{leaf}[x] \leftarrow \text{TRUE} \\
\quad n[x] \leftarrow 0 \\
\quad \text{DISK-WRITE}(x) \\
\quad \text{root}[T] \leftarrow x
\end{array}
$$

- `Allocate-Node()` aloca uma página do disco para ser utilizada como um novo nó

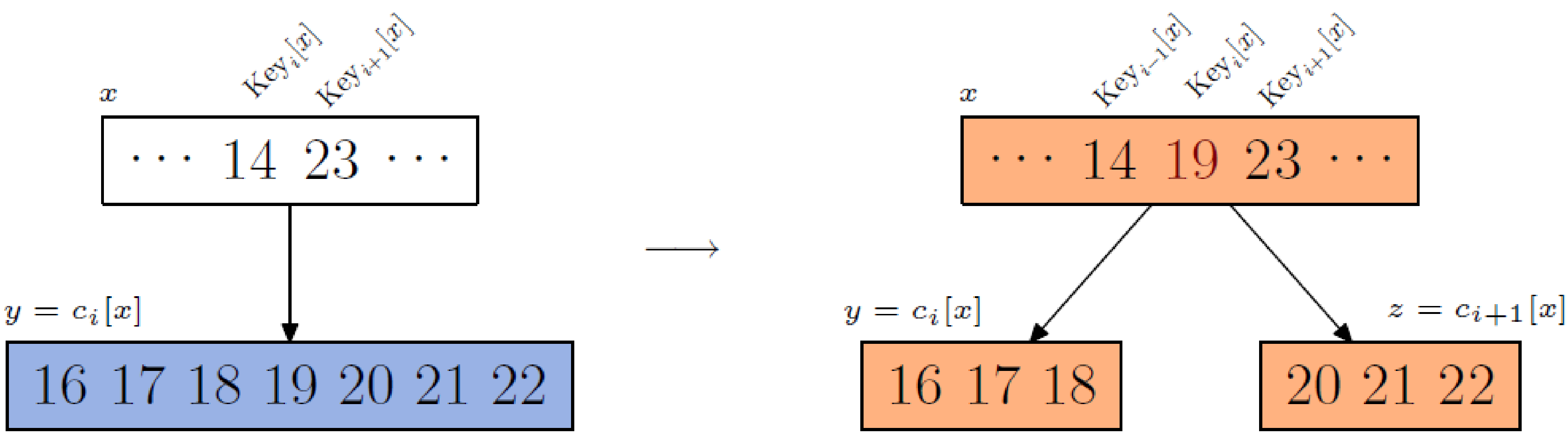
- Requer O(1) operações de disco em um tempo de CPU O(1)

Inserção de Elementos

- As chaves são inseridas em ordem crescente
- Inserção é sempre realizada em nós folhas
- Inserção é sempre feita em uma única passagem pela árvore
- Requer $O(h) = O(\log_t n)$ acessos ao disco
- Requer um tempo de CPU de $O(th) = O(t \log_t n)$ acessos ao disco
- Quando o número de elementos é ultrapassado a quantidade de elementos deve ser dividida entre o novo nó e o nó antigo
  - Utiliza a operação `B-Tree-Split-Child` para garantir que a recursão nunca alcance um nó cheio

Divisão de nós em Árvores B

- Inserir uma chave em uma árvore B é mais complicado que em uma árvore binária de busca
- Divisão de um nó *y* completo (ou seja, com *2t-1* chaves) é uma operação fundamental durante a inserção
- Divisão ocorre em torno de um elemento chave mediana $chave_t[y]$ em dois sub-nós
  - Chave mediana é movida para o nó pai (que não pode estar completo!)
  - Se *y* é a raiz, a altura aumenta em 1



Divisão de nós em Árvores B

- 3 entradas:
  - o *x*, um nó interno não completo
  - o *i*, um índice
  - o *y*, um nó dado que $y = c_i[x]$ é um nó completo, filho de *x*

```
B-TREE-SPLIT-CHILD(x, i, y)
    z ← ALLOCATE-NODE()
    leaf[z] ← leaf[y]
    n[z] ← t −1
    for j ← 1 to t − 1
        do key_j[z] ← key_{j+t}[y]
    if not leaf[y]
        then for j ← 1 to t
            do c_j[z] ← c_{j+t}[y]
    n[y] ← t − 1
    for j ← n[x] + 1 downto i + 1
        do c_{j+1}[x] ← c_j[x]
    c_{i+1}[x] ← z
    for j ← n[x] downto i
        do key_{j+1}[x] ← key_j[x]
    key_i[x] ← key_t[y]
    n[x] ← n[x] + 1
    DISK-WRITE(y)
    DISK-WRITE(z)
    DISK-WRITE(x)
```

- Tempo de CPU usado pela função `B-Tree-Split-Child` é Θ(t) devido aos laços

Inserção de Elementos

- Tente inserir a nova chave em um nó folha (na posição adequada)
- Se isso fizer com que o nó fique cheio, divida a folha em duas partes e suba o elemento central para o nó pai
- Se isso fizer com que o pai fique cheio repita o processo
- A estratégia poderá ser repetida até o nó raiz
- Se necessário o nó raiz deverá ser também divido e o elemento central será transformado em nova raiz (fazendo com que a árvore fique mais alta)

Inserção de Elementos

- 2 entradas:
  - root[T], a raiz da árvore
  - k, a chave a ser inserida

```
B-TREE-INSERT(T, k)
  r ← root[T]
  if n[r] = 2t − 1
     then s ← ALLOCATE-NODE()
          root[T] ← s
          leaf[s] ← FALSE
          n[s] ← 0
          c1[s] ← r
          B-TREE-SPLIT-CHILD(s,1,r)
          B-TREE-INSERT-NONFULL(s,k)
     else B-TREE-INSERT-NONFULL(r,k)
```

  - Usa a função `B-Tree-Insert-Nonfull` para inserir a chave *k* em um nó não completo *x*

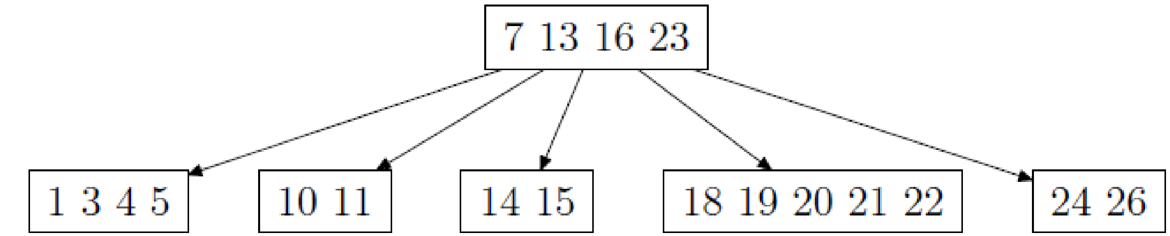
Inserção de Elementos em nós não completos

```
B-TREE-INSERT-NONFULL(x, k)
  i ← n[x]
  if leaf[x]
    then while i ≥ 1 and k < key_i[x]
            do key_{i+1}[x] ← key_i[x]
               i ← i − 1
         key_{i+1}[x] ← k
         n[x] ← n[x] + 1
         DISK-WRITE(x)
    else while i ≥ 1 and k < key_i[x]
            do i ← i − 1
         i ← i + 1
         DISK-READ(c_i[x])
         if n[c_i[x]] = 2t − 1
            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                 if k > key_i[x]
                    then i ← i + 1
         B-TREE-INSERT-NONFULL(c_i[x], k)
```

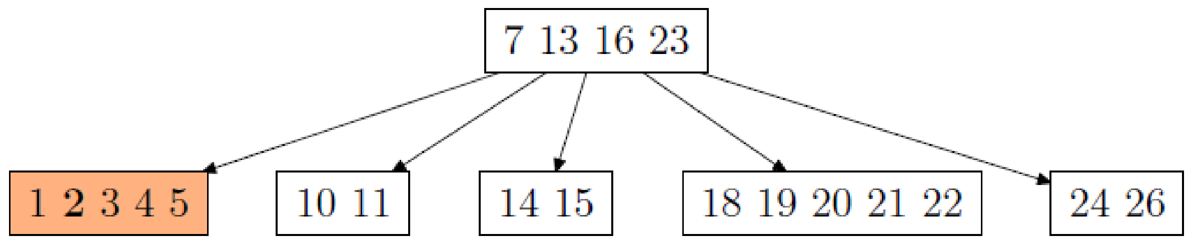
Inserção de Elementos – Exemplos

Initial tree:

$t = 3$

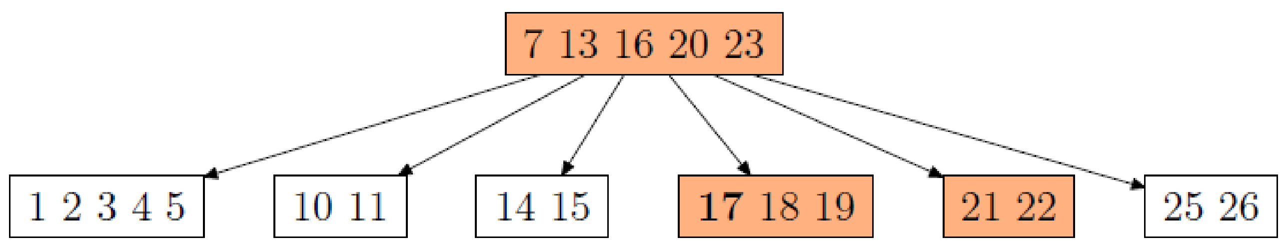


2 inserted:



17 inserted:

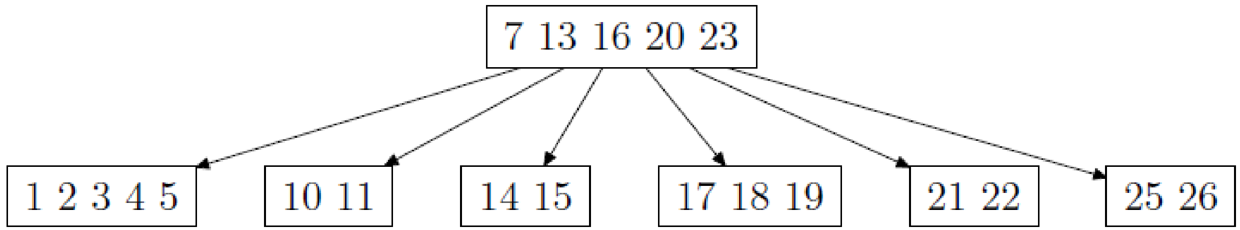
(to the previous one)



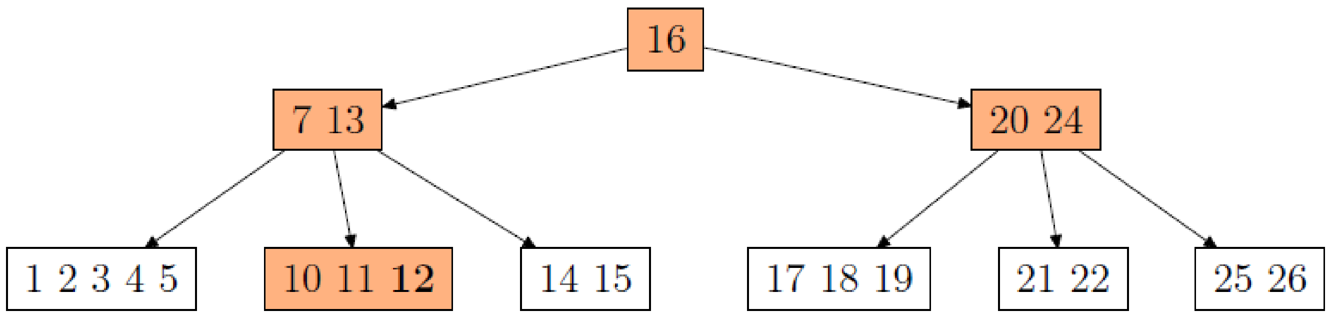
Inserção de Elementos – Exemplos

Initial tree:

$t = 3$

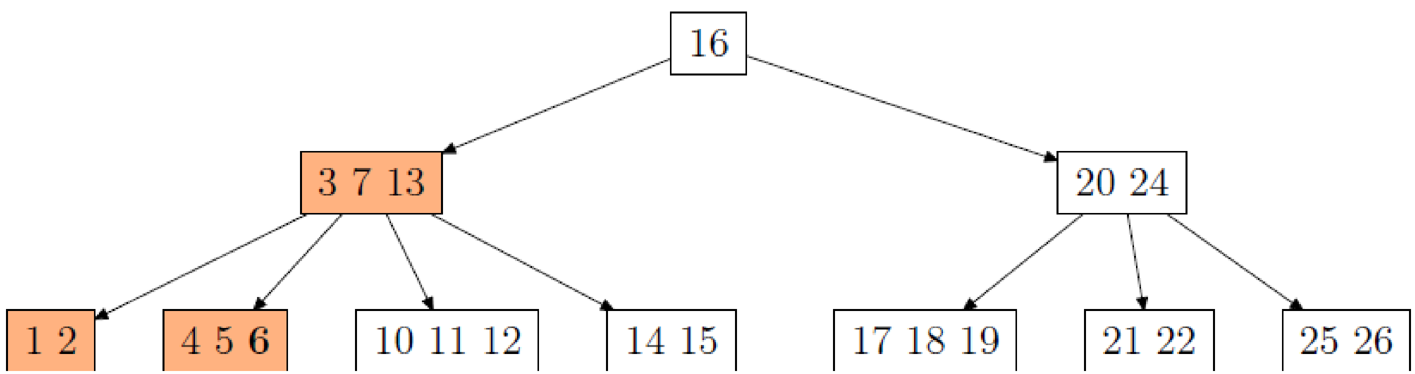


12 inserted:



6 inserted:

(to the previous one)



Mais exemplos de inserção...

- Suponha que iniciemos com uma árvore B vazia e as chaves devem ser inseridas na seguinte ordem:

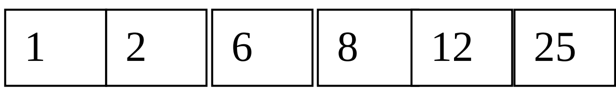
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

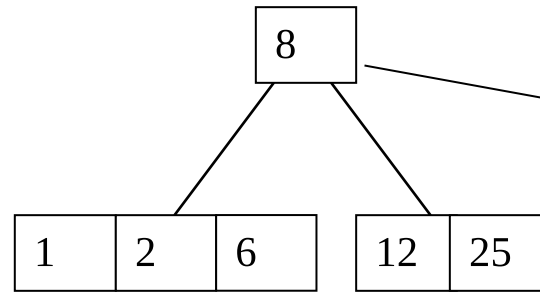
- Queremos construir uma árvore B com t = 3
- Os 5 primeiros elementos vão para a raíz:

1	2	8	12	25

- O sexto elemento extrapola o tamanho do nó
- Assim, quando inserimos o 6 devemos dividir o nó em duas partes e colocar o elemento do meio como nova raiz

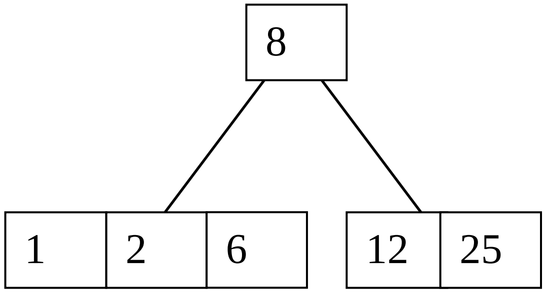
Inserindo o 6 ocorre quebra da regra de tamanho máximo



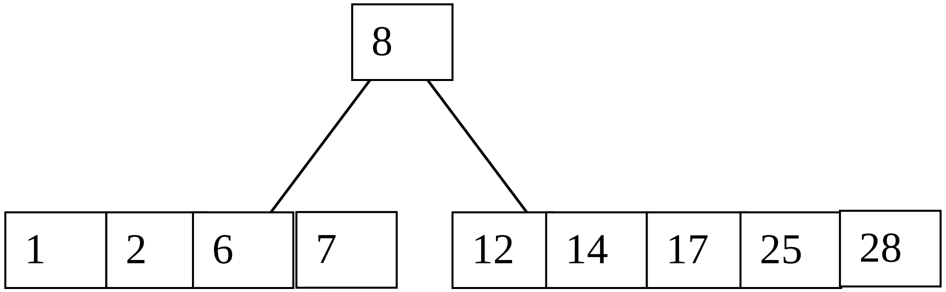


É preciso fazer o split

6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

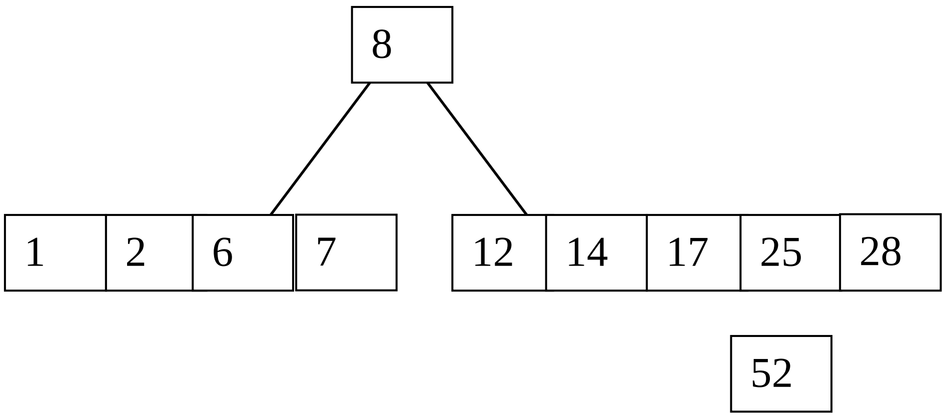


Em seguida colocamos 14, 28, 17 e 7 :

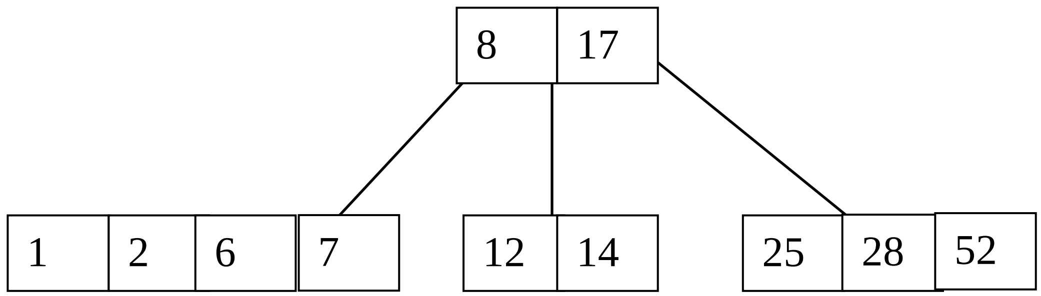


14 28 17 7 52 16 48 68 3 26 29 53 55 45

Adicionando 52 à árvore teremos outro split…

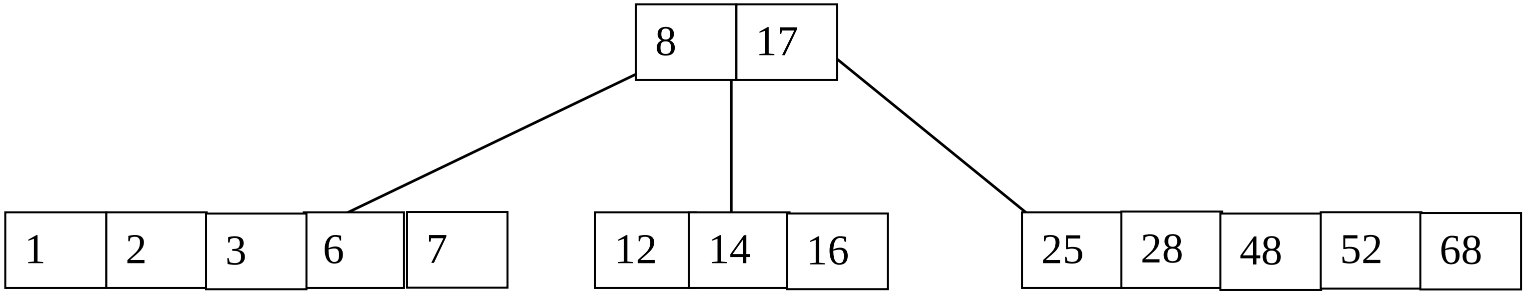


52 16 48 68 3 26 29 53 55 45



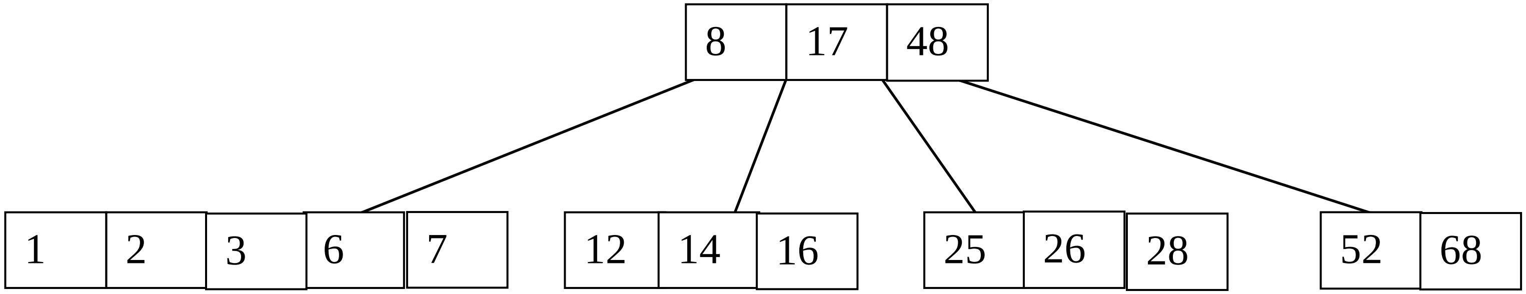
52 16 48 68 3 26 29 53 55 45

Continuando com 16, 48, 68 e 3:



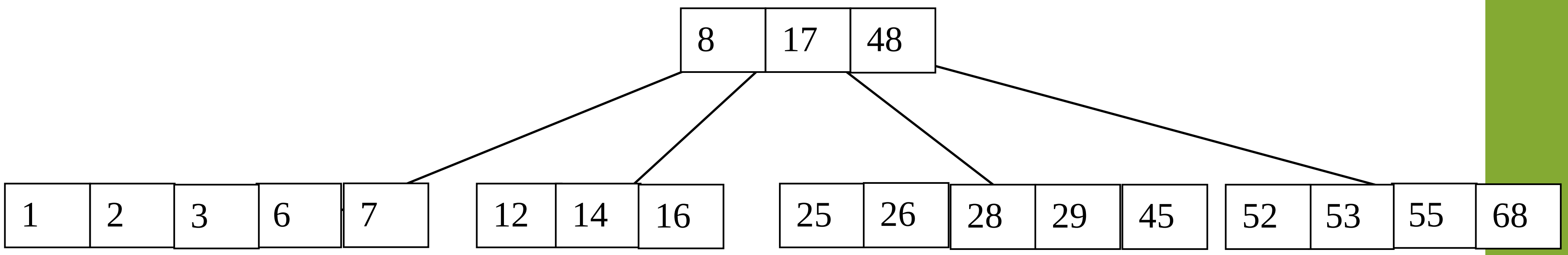
16 48 68 3 26 29 53 55 45

Adicionando 26 à árvore causa um “split” na folha mais à direita, fazendo com que o 48 suba à raiz.



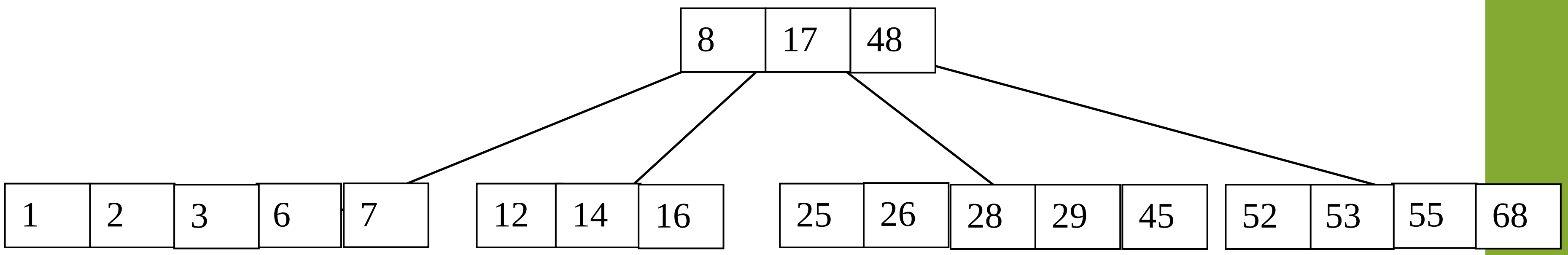
26 29 53 55 45

Por fim, inserindo o 29, 53, 55 e 45 a árvore fica da seguinte maneira



29 53 55 45

Exercício: e se forem inseridos os elementos 5, 19, 70 e 72?
Responda ilustrando a árvore resultante de cada inserção.



Remoção de Elementos

- A operação é remoção é similar à inserção, com a adição de alguns casos especiais
- Uma chave pode ser excluída de qualquer nó
- Procedimento mais complexo, porém similar em termos da análise de complexidade: $O(h)$ acessos ao disco, $O(th) = O(t \log_t n)$ tempo de CPU
- Exclusão é feita em uma única passada na árvore, mas precisa retornar ao nó em que a chave foi excluída se ele for um nó interno
- Neste último caso, a chave é primeiramente movida para uma folha abaixo. A exclusão final é sempre realizada em um folha

Remoção de Elementos – Casos

- São considerados três casos distintos para a exclusão
- Seja *k* a chave a ser excluída e *x* o nó contendo a chave

1. Se a chave *k* está no nó *x,* e *x* é uma folha, elimine a chave *k* de *x*
2. Se a chave *k* está no nó *x* e *x* é um nó interno, existem três sub-casos para considerar:
   a. Se o filho *y* que precede *k* no nó *x* tem pelo menos *t* chaves (mais que o mínimo), então encontre a chave predecessora *k'* de *k* na sub-árvore com raiz *y*. Elimine recursivamente *k'*, e substitua *k* por *k'* em *x*
   b. Simetricamente, se o filho *z* que segue *k* no nó *x* tem pelo menos *k* chaves, encontre o sucessor *k'* de *k* na sub-árvore com raiz em *z*. Elimine recursivamente *k'* e substituta *k* por *k'* em *x*, da mesma forma como discutido no sub-caso anterior
   c. Caso contrário, se tanto *y* quanto *z* tem apenas *t-1* chaves (o mínimo), intercale *k* e todos os elementos de *z* em *y*, de modo que tanto *k* quanto o ponteiro para *z* sejam removidos de *x*. Agora *y* contém *2t-1* chaves. Em seguida, libere *z* e elimine recursivamente *k* de *y*.

Remoção de Elementos – Casos

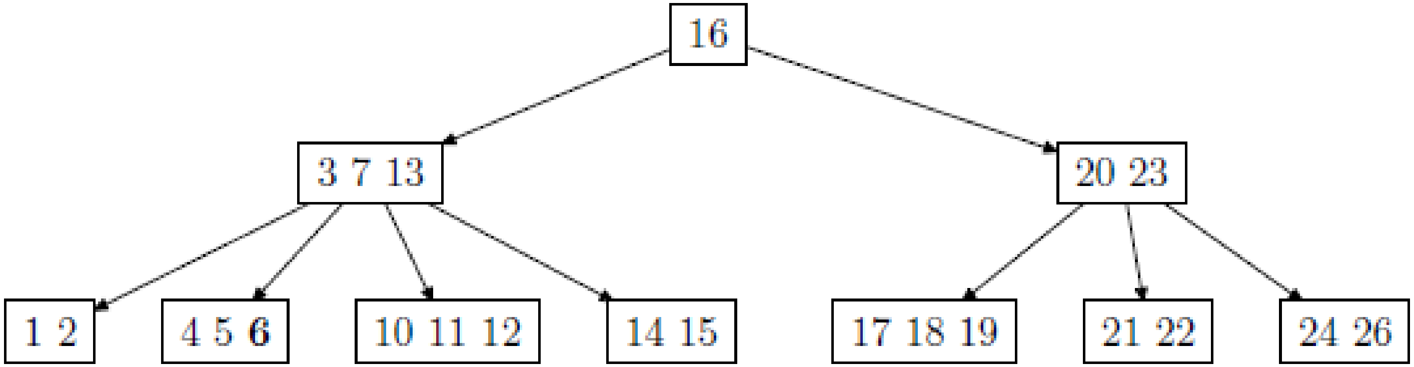
3. Se a chave *k* não estiver presente no nó interno *x*, determine a raiz $c_i[x]$ da sub-árvore apropriada que deve conter *k*, se *k* estiver absolutamente na árvore. Se a raiz $c_i[x]$ tiver apenas *t-1* chaves, execute o passo 3(a) ou 3(b) para garantir que o algoritmo descerá para um nó contendo pelo menos *t* chaves. Em seguida realize uma recursão sobre o filho apropriado de *x*

   a. Se $c_i[x]$ tiver somente *t-1* chaves, mas tiver um irmão com *t* chaves, forneça a $c_i[x]$ uma chave extra, movendo uma chave de *x* para baixo até $c_i[x]$, movendo uma chave do irmão esquerdo ou direito de $c_i[x]$ para *x*, e movendo o filho apropriado do irmão para $c_i[x]$

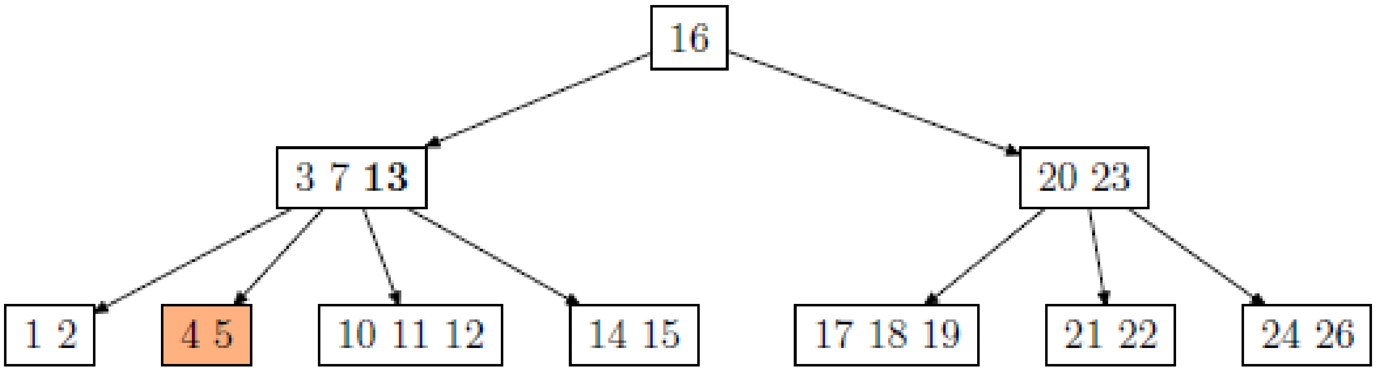
   b. Se a raiz $c_i[x]$ e todos os seus irmãos têm *t-1* chaves, faça a intercalação de $c_i[x]$ com um irmão. Isso envolve mover uma chave de *x* para baixo até o novo nó intercalado para se tornar a chave mediana desse nó

Remoção de Elementos – Exemplo Caso 1

Initial tree:



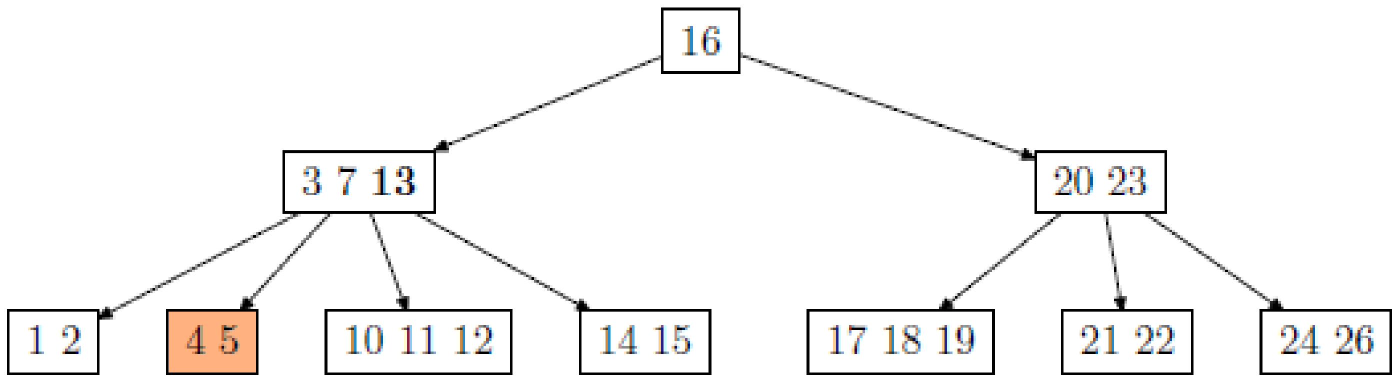
6 deleted:



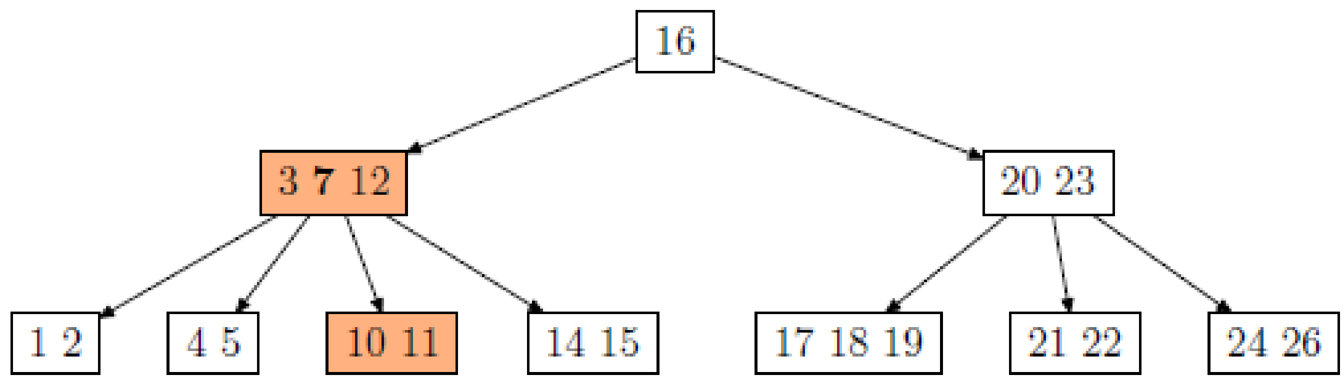
- O primeiro (e mais simples) caso envolve a remoção de uma chave em uma folha. *t-1* chaves permanecem.

Remoção de Elementos – Exemplo Caso 2a, 2b

Initial tree:

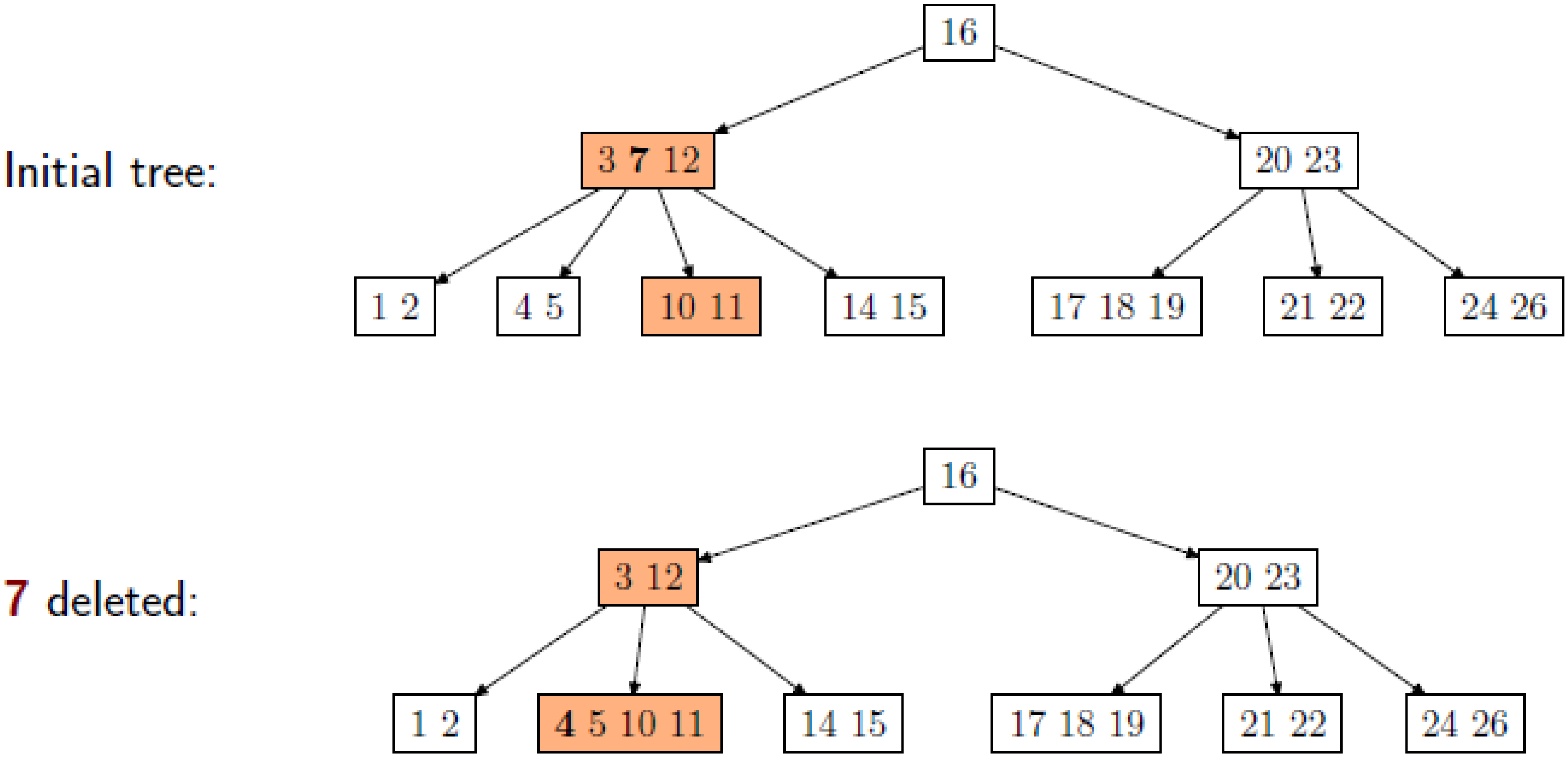


13 deleted:



- Caso 2a é ilustrado. O predecessor de 13, que se situa no filho precedente de *x*, é movido para cima e ocupa a posição de 13. O filho predecessor tinha uma chave sobrando nesse caso.

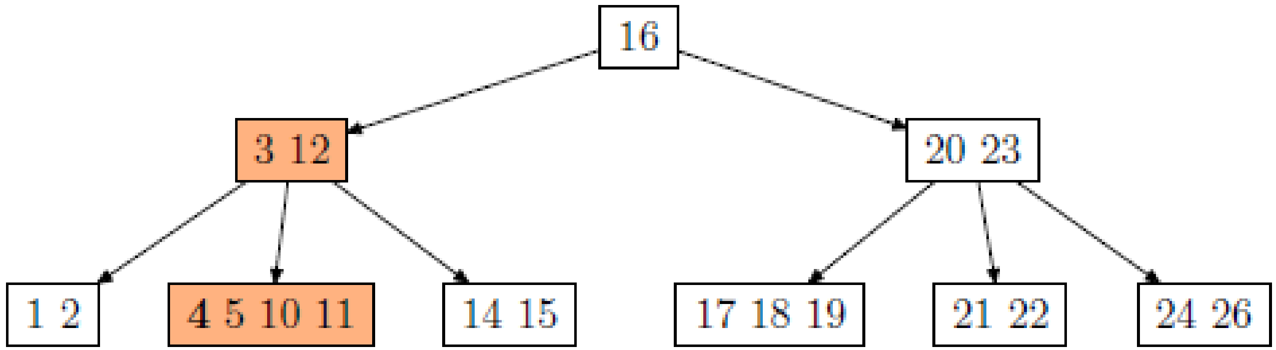
Remoção de Elementos – Exemplo Caso 2c



- Aqui, ambos os filhos sucessores e predecessores têm *t-1* chaves, o mínimo permitido. 7 é inicialmente empurrado para baixo e os nós filhos são intercalados para formar uma única folha. Subsequentemente, o valor é removido dessa folha.

Remoção de Elementos – Exemplo Caso 3b

Initial tree:
Key 4 to be deleted



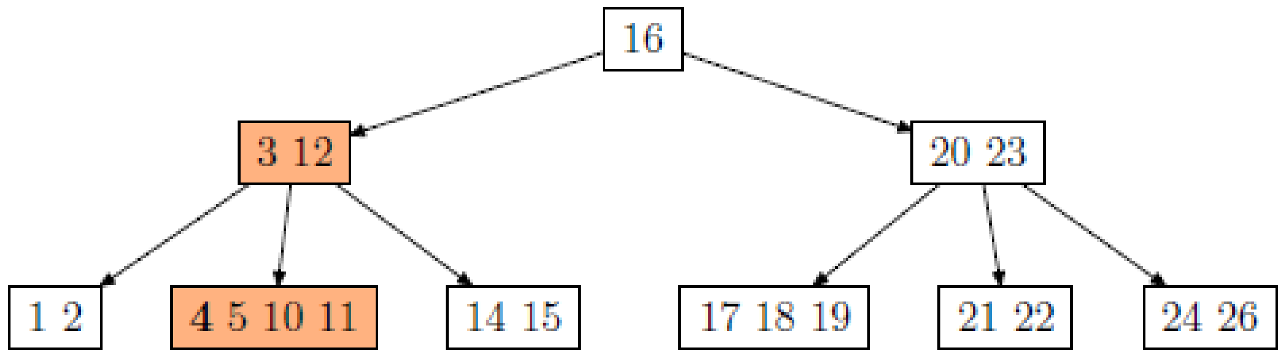
- A recursão aqui não pode descer do nó 3, 12 porque ele tem *t-1* chaves. No caso das duas folhas da esquerda e da direita terem mais que *t-1* chaves, 3, 12 poderia obter uma e o 3 ser movido para baixo

- Além disso, o irmão de 3, 12 também tem *t-1* chaves, então não é possível mover a raiz para a esquerda e tomar o elemento mais a esquerda do irmão para a nova raiz

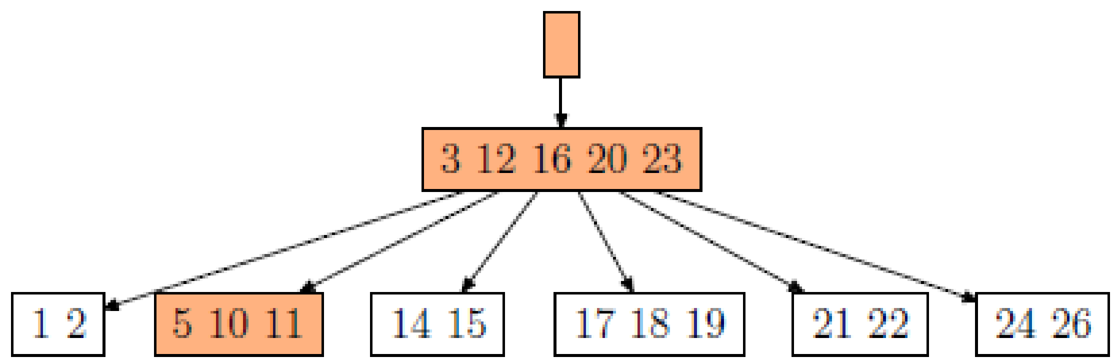
- Portanto, a raiz deve ser empurrada para baixo e intercalada com seus dois filhos, assim o 4 pode ser excluído com segurança

Remoção de Elementos – Exemplo Caso 3b

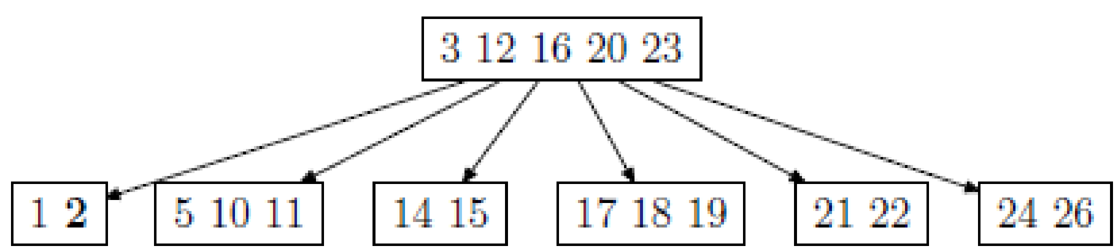
Initial tree:



4 deleted:

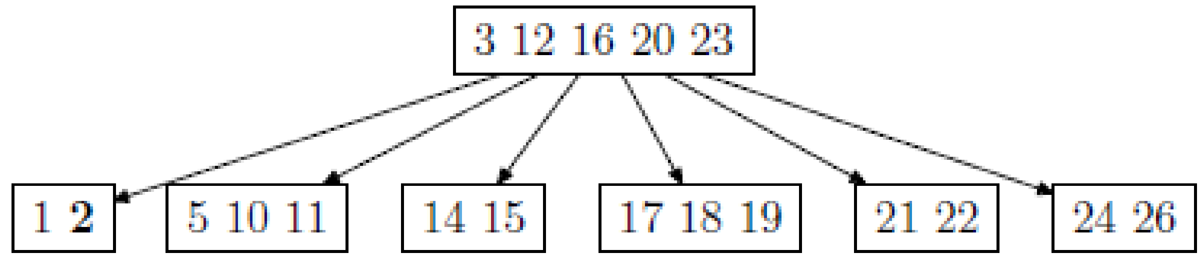


Outcome:



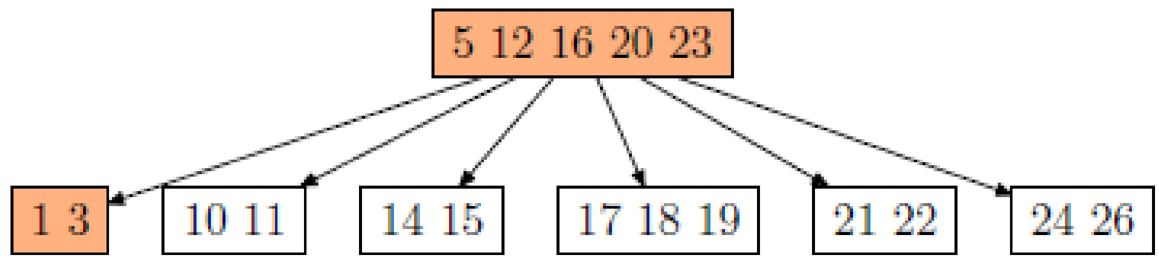
Remoção de Elementos – Exemplo Caso 3a

Initial tree:



2 deleted:

(to the previous one)



- Nesse caso, 1, 2 tem *t-1* chaves, mas o irmão da direita tem *t*. A recursão move 5 para preencher a posição de 3, que passa a ocupar a posição de 2.

Remoção de Elementos – Pseudocódigo (1)

```
B-Tree-Delete-Key(x, k)
  if not leaf[x] then
    y ← Preceding-Child(x)
    z ← Successor-Child(x)
    if n[y] > t − 1 then             // Caso 2a
        k′ ← Find-Predecessor-Key(k, x)
        Move-Key(k′, y, x)
        Move-Key(k, x, z)
        B-Tree-Delete-Key(k, z)
    else if n[z] > t − 1 then        // Caso 2b
        k′ ← Find-Successor-Key(k, x)
        Move-Key(k′, z, x)
        Move-Key(k, x, y)
        B-Tree-Delete-Key(k, y)
    else                             // Caso 2c
        Move-Key(k, x, y)
        Merge-Nodes(y, z)
        B-Tree-Delete-Key(k, y)
```

Remoção de Elementos – Pseudocódigo (2)

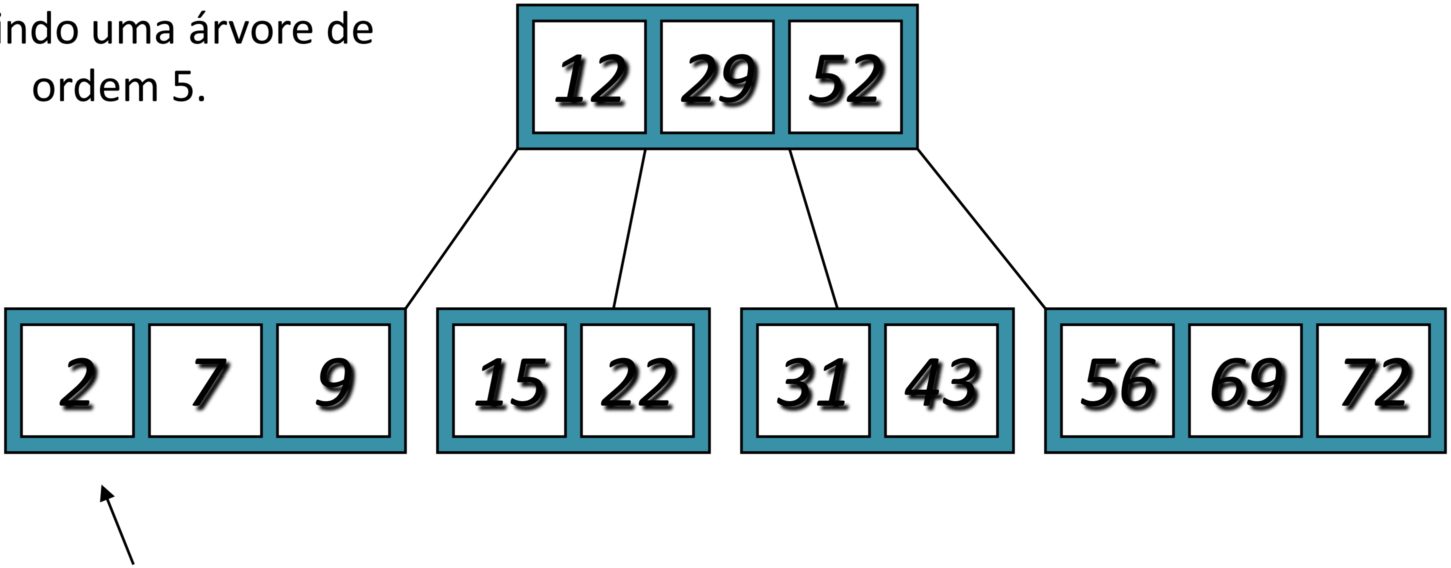
```
else (leaf node)
  y ← PRECEDING-CHILD(x)
  z ← SUCCESSOR-CHILD(x)
  w ← root(x)
  v ← RootKey(x)
    if n[x] > t − 1 then REMOVE-KEY(k, x)      // Caso 1
    else if n[y] > t − 1 then                   // Caso 3a
        k' ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k', y, w)
        k' ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k', w, x)
        B-TREE-DELETE-KEY(k, x)
    else if n[w] > t − 1 then                   // Caso 3a
        k' ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k', z, w)
        k' ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k', w, x)
        B-TREE-DELETE-KEY(k, x)
```

Remoção de Elementos – Pseudocódigo (3)

```
else                                          // Caso 3b
    s ← FIND-SIBLING(w)
    w' ← root(w)
        if n[w'] = t − 1 then
            MERGE-NODES(w', w)
            MERGE-NODES(w, s)
            B-TREE-DELETE-KEY(k, x)
        else
            MOVE-KEY(v, w, x)
            B-TREE-DELETE-KEY(k, x)
```

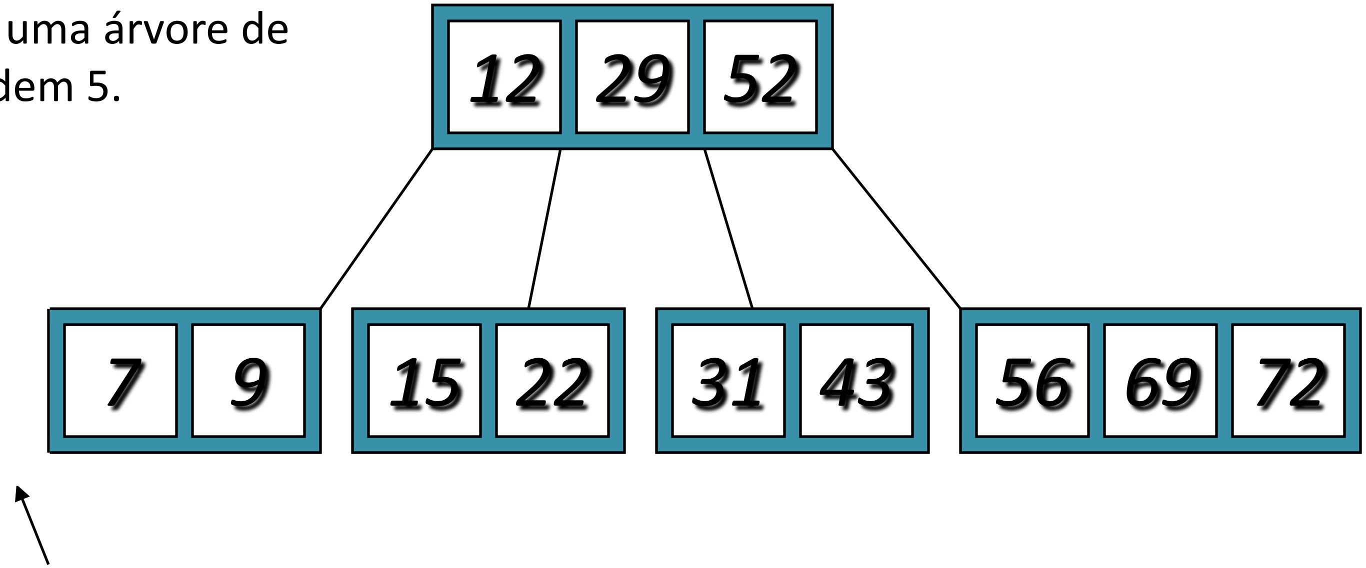
- PRECEDING-CHILD(x) Returns the left child of key x.
- MOVE-KEY(k, n_1, n_2) Moves key k from node n_1 to node n_2.
- MERGE-NODES(n_1, n_2) Merges the keys of nodes n_1 and n_2 into a new node.
- FIND-PREDECESSOR-KEY(n, k) Returns the key preceding key k in the child of node n.
- REMOVE-KEY(k, n) Deletes key k from node n. n must be a leaf node.

Assumindo uma árvore de ordem 5.



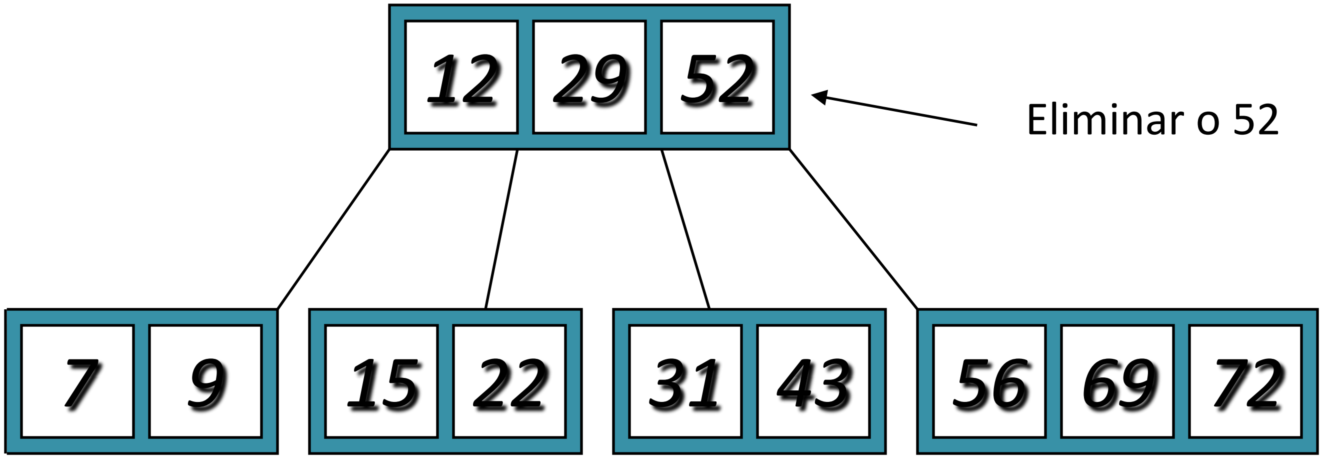
Eliminar o 2: Há chaves suficientes

Assumindo uma árvore de ordem 5.

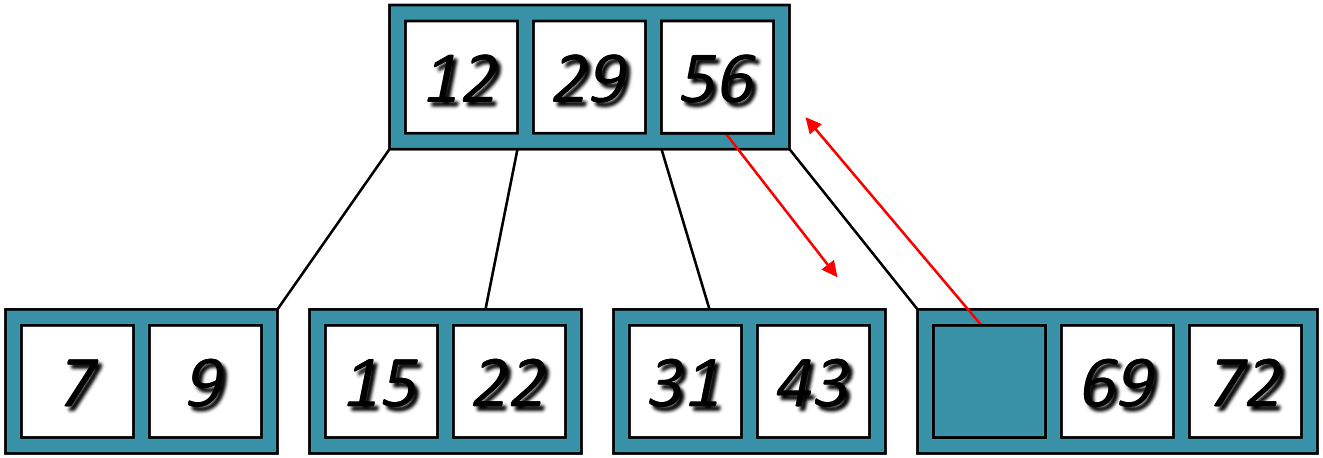


Eliminar o 2: Há chaves suficientes

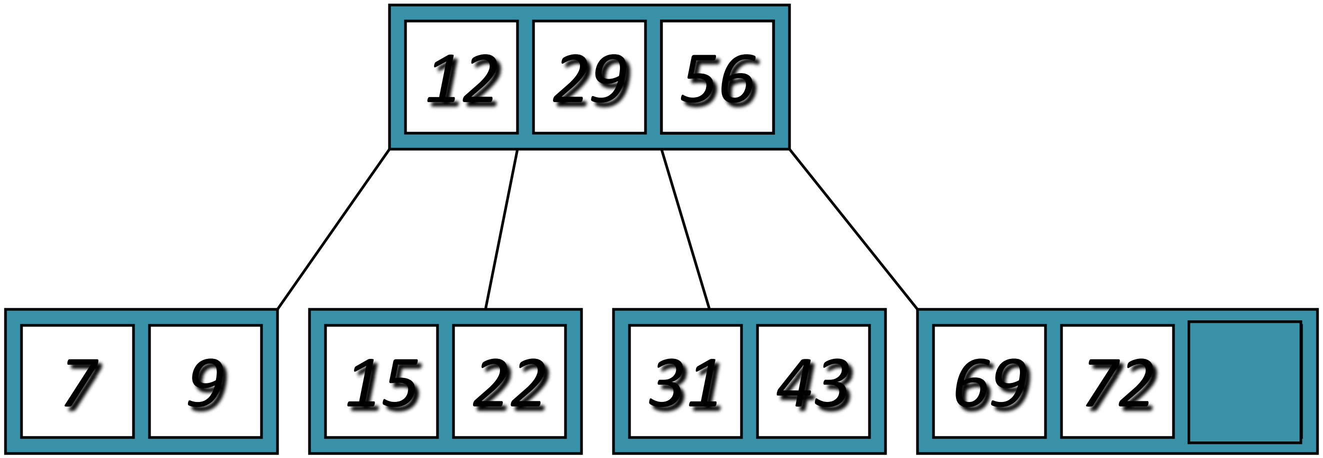
Árvore B
(remoção de nó não folha)



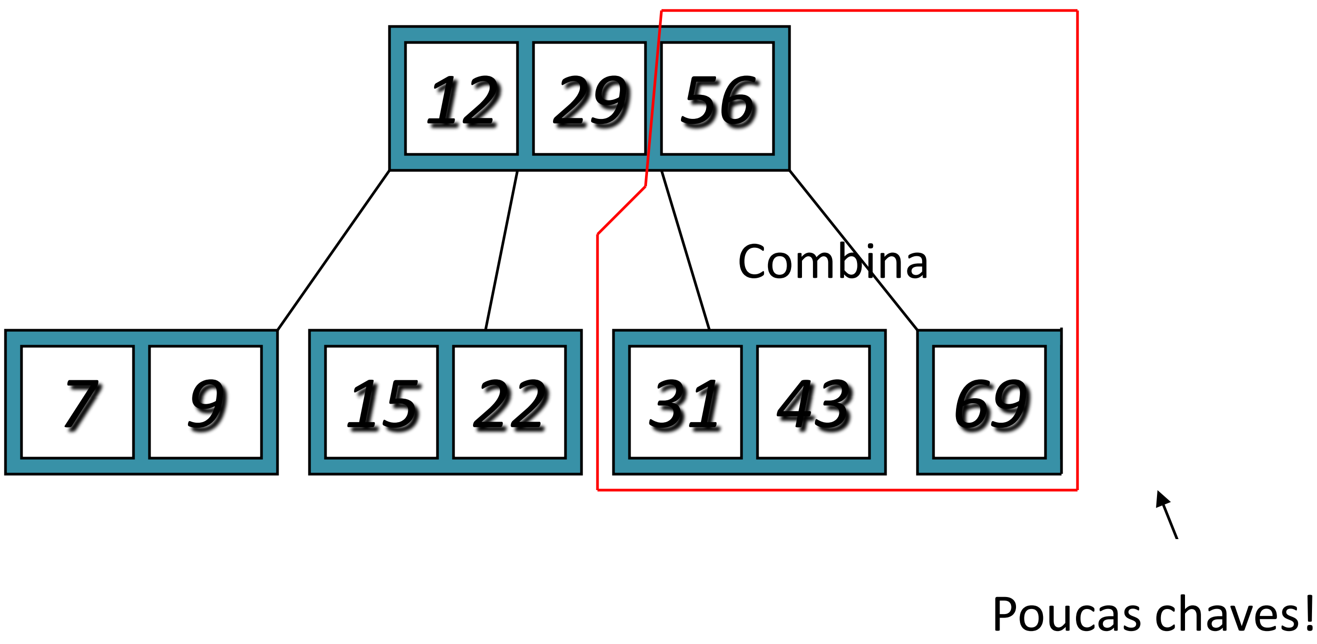
Árvore B
(remoção de nó não folha)



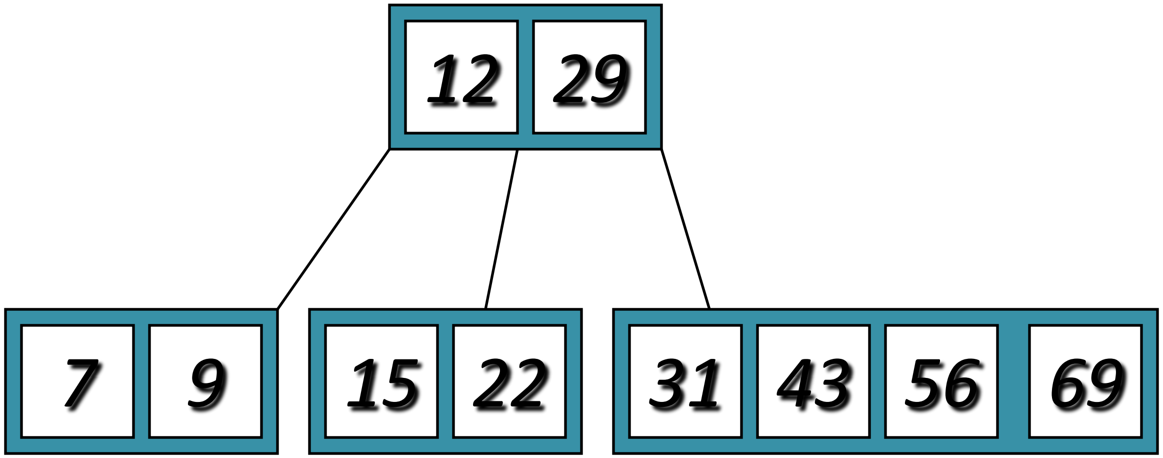
Remoção de nó não folha



Remoção de nó com poucas chaves

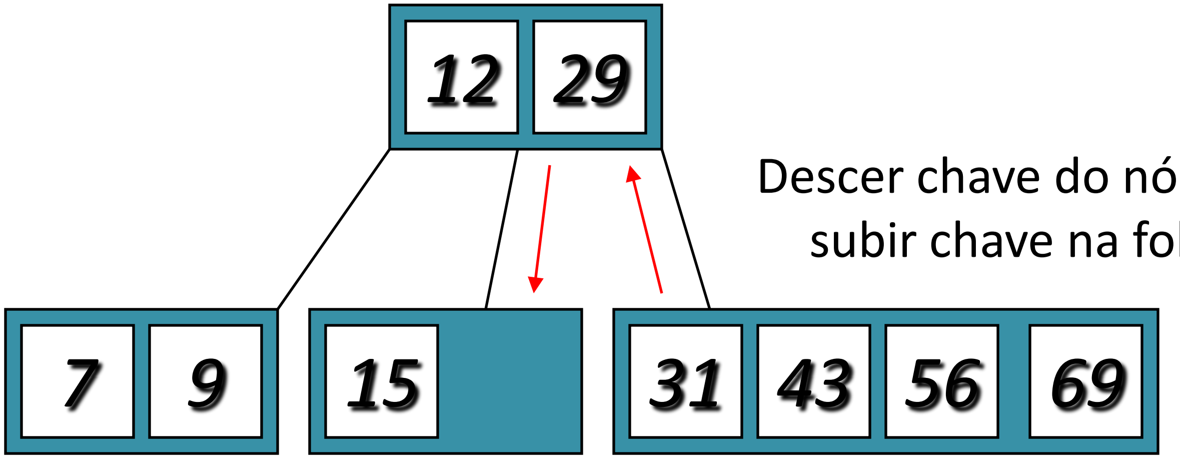


Remoção - Poucas chaves nos nós irmãos



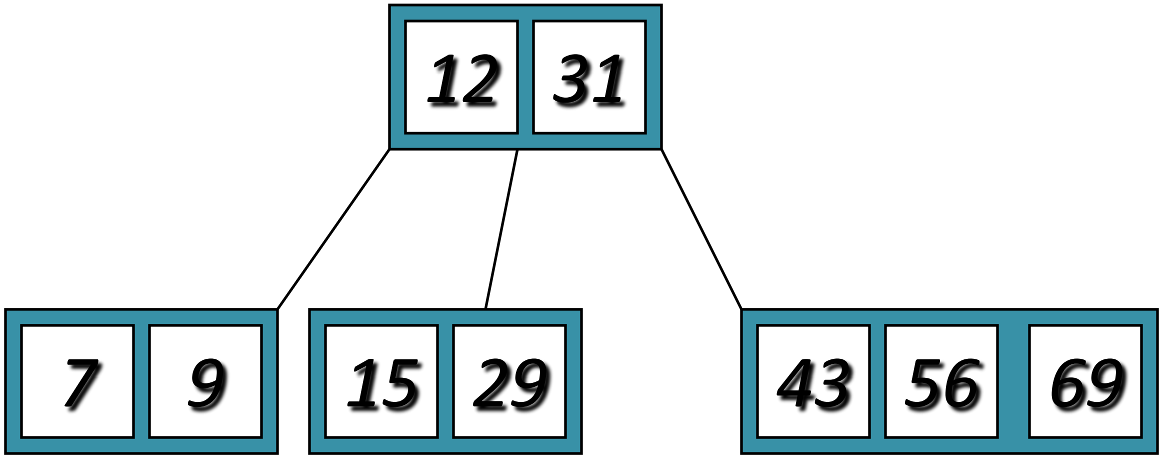
Eliminar o 22

Remoção - Poucas chaves nos nós irmãos



Descer chave do nó pai e
subir chave na folha

Remoção - Poucas chaves nos nós irmãos



Animação

http://slady.net/java/bt/view.php?w=600&h=450

Exercícios

- Insira os seguintes números em uma árvore B com grau mínimo 2:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Faça a inserção dos elementos agora em uma árvore B com grau mínimo 4