B-Tree

Busca em Memória Externa

Animação
https://www.cs.usfca.edu/~galles/visualization/BTree.html

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

Motivação

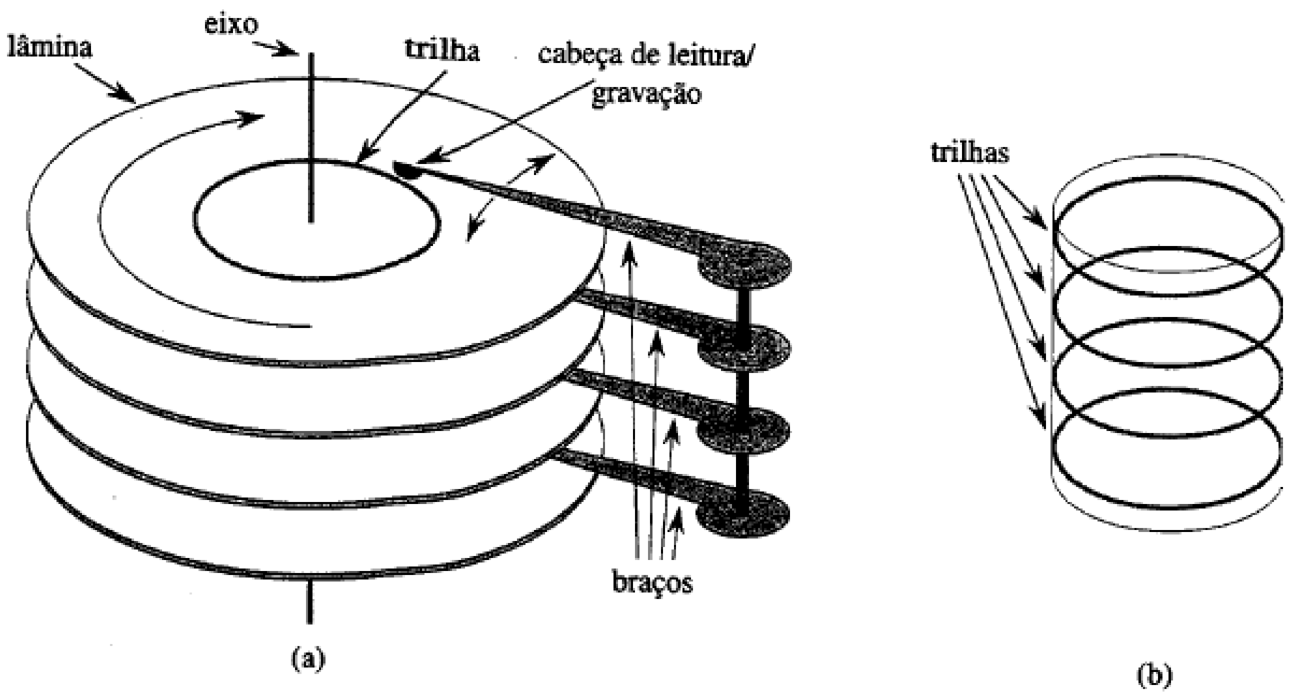


FIGURA 18.2 **(a)**Uma unidade de disco típica. Ela é composta por várias lâminas que giram em torno de um eixo. Cada lâmina é lida e gravada com uma cabeça na extremidade de um braço. Os braços são associados de modo a moverem suas cabeças em conjunto. Aqui, os braços giram em torno de um eixo pivô comum. Uma trilha é a superfície que passa sob a cabeça de leitura/gravação quando ela é estacionária. **(b)** Um cilindro consiste em um conjunto de trilhas concêntricas

Memória Virtual

- Normalmente implementado como uma função do sistema operacional

- Modelo de armazenamento em dois níveis, devido à necessidade de grandes quantidades de memória e o alto custo da memória principal.

- Uso de uma pequena quantidade de memória principal e uma grande quantidade de memória secundária.

- Programador pode endereçar grandes quantidades de dados, deixando para o sistema a responsabilidade de trasferir o dado da memória secundária para a principal.

Sistema de Paginação

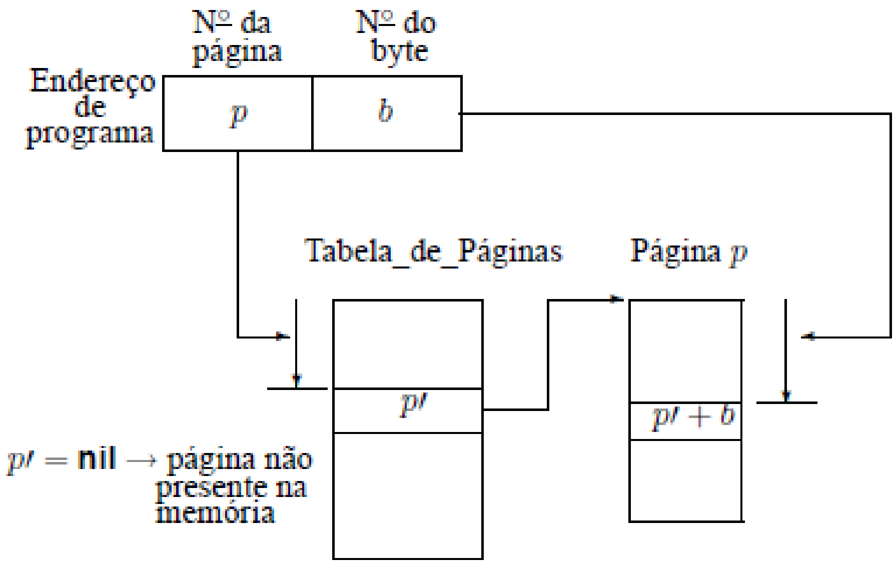
- **O espaço de endereçamento é dividido em páginas de tamanho igual, em geral, múltiplos de 512 *bytes*.**
- **A memória principal é dividida em molduras de páginas de tamanho igual.**
- **As molduras de páginas contêm algumas páginas ativas enquanto o restante das páginas estão residentes em memória secundária (páginas inativas).**

Sistemas de Paginação

- **O mecanismo possui duas funções:**
  - 1. Mapeamento de endereços: Determinar qual página um programa está endereçando, encontrar a moldura, se existir, que contenha a página.
  - 2. Transferência de páginas: transferir páginas da memória secundária para a memória primária e transferí-las de volta para a memória secundária quando não estão mais sendo utilizadas.
- **Endereçamento da página: uma parte dos bits é interpretada como um número de página e a outra parte como o número do byte dentro da página (*offset*).**

Sistema de Paginação

- **Mapeamento de endereços ! Realizado através de uma Tabela de Páginas.**
  – a p-ésima entrada contém a localização p0 da Moldura de Página contendo a página número p desde que esteja na memória principal.



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

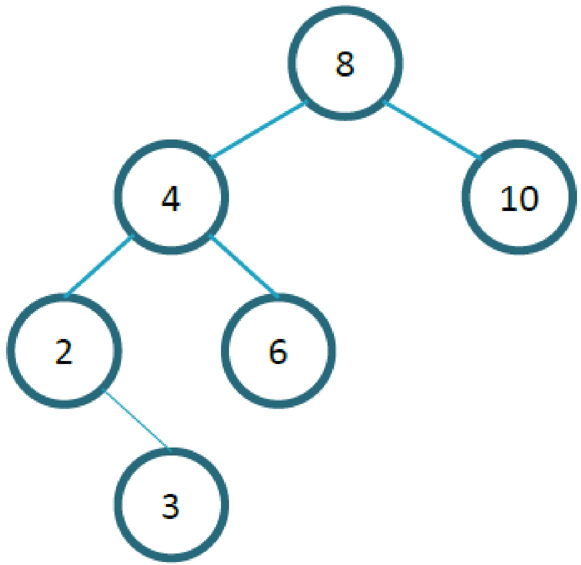
O que são Árvores B

- **Proposta em 1972 por Bayer e McCreight, desenvolvido no Laboratório de Pesquisas Científicas Boeing**
  - Não se sabe se o B é de Bayer ou Boeing

- **Usada no armazenamento em memória secundária**
- **É uma árvore n-ária**

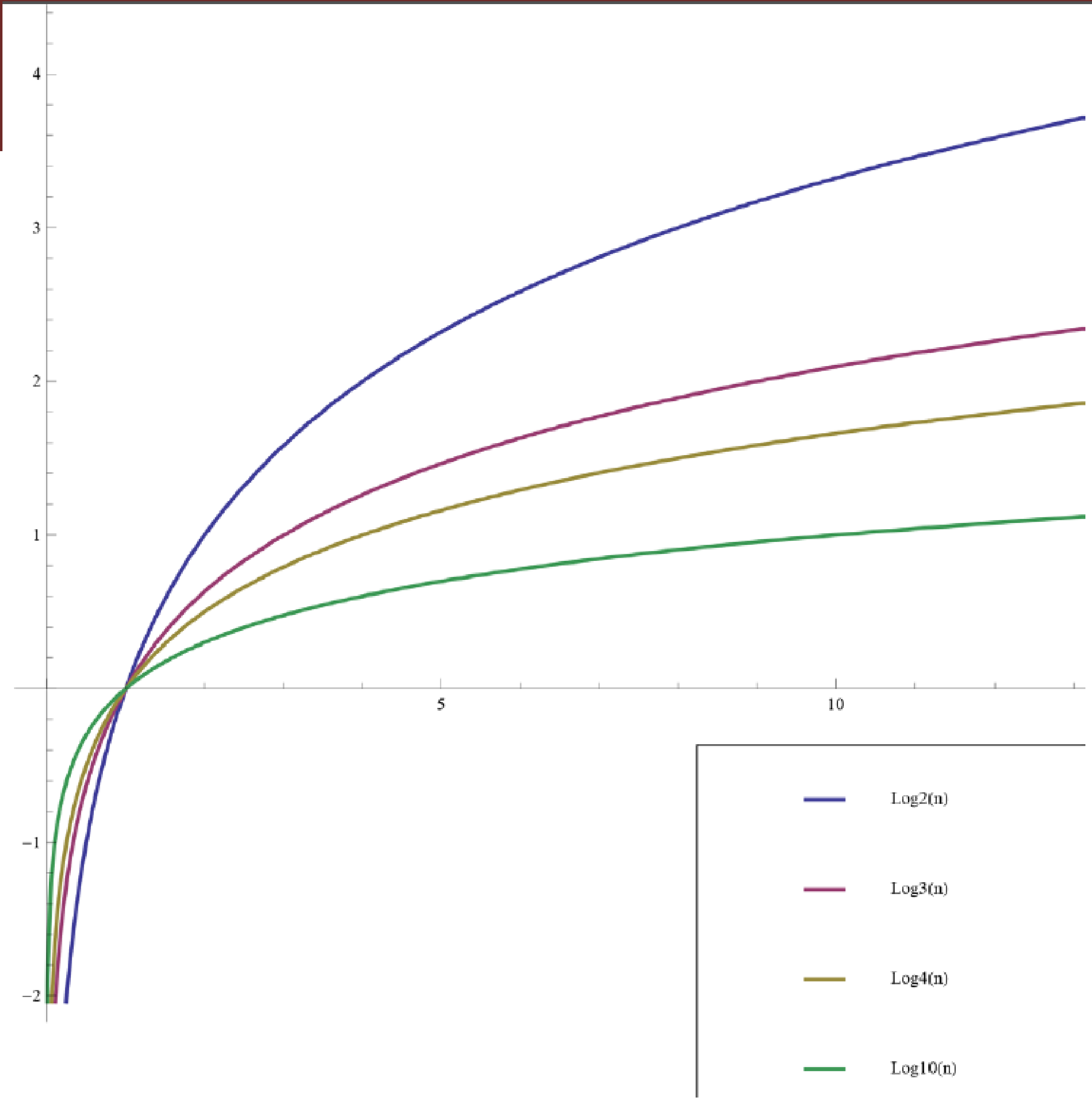
B-Tree

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



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

B-Tree

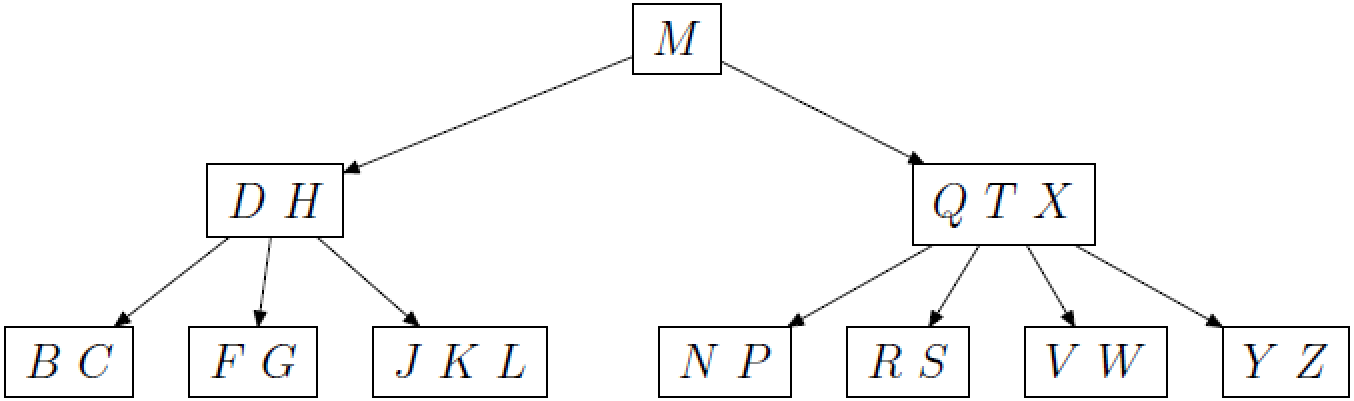


Características

- **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**

Exemplo

- **Os registros aparecem em ordem crescente da esquerda para a direita.**
- **Os nodos folha tem os seus campos ponteiro com valores nulos.**
- **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**

Estrutura do nodo da B-tree

- **Considerando *X* como um nodo interno da B-Tree e *n* o número de chaves armazenadas em *X***
  - as ***n*** chaves ***k1, k2...kn*** são armazenadas em ordem crescente;
  - e ***X*** possui ***n+1*** ponteiros ***f1, f2...fn+1*** para seus filhos

f1	k1	f2	k2	f3	k3	f4	k4	...	...	fn	kn	fn+1

  - As chaves separam os intervalos de chaves armazenados em cada subárvore: se ki é qualquer chave armazenada na subárvore com raiz ci[x], então:

$$k_1 \leq chave_1[x] \leq k_2 \leq chave_2[x] \leq \ldots \leq chave_{n[x]}[x] \leq k_{n[x]+1} \, .$$

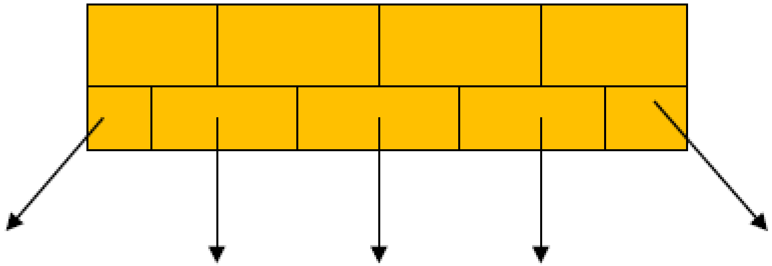
Definição

- 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

      $$k_1 \leq chave_1[x] \leq k_2 \leq chave_2[x] \leq \ldots \leq k_{n[x]} \leq chave_{n[x]+1}[x]$$
  - 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 $t \geq 2$, o **grau mínimo** da árvore B
    - Limitante inferior: todo nó diferente da raiz deve ter pelo menos t-1 chaves (e t filhos)
    - Limitante superior: cada nó pode conter no máximo $2t$-1 chaves (e $2t$ filhos)

Nó B-Tree

- 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;
```

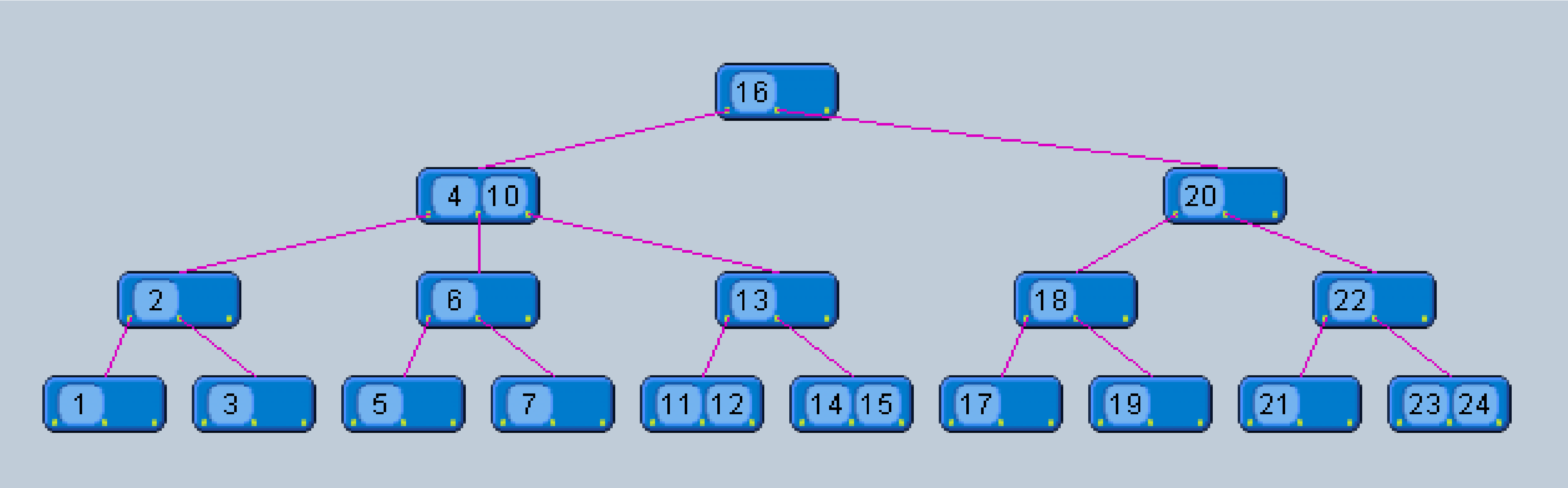
Disposição dos elementos ordenados na B-tree

- **Em um nodo, cada campo chave tem associado a ele um campo ponteiro para um nodo filho**
  - Esse nodo filho é a raiz de uma sub-árvore que contém nodos com todos os valores de chave menores ou iguais ao valor da chave em questão no nodo pai; e maiores do que o valor da chave anterior no nodo pai.

- **Cada nodo tem também um campo ponteiro adicional que indica o seu filho mais à direita**
  - Esse nodo é a raiz de uma sub-árvore que contém nodos com todos os valores de chave maiores do que todas as chaves do nodo pai.

Exemplo de uma B-tree de grau 3

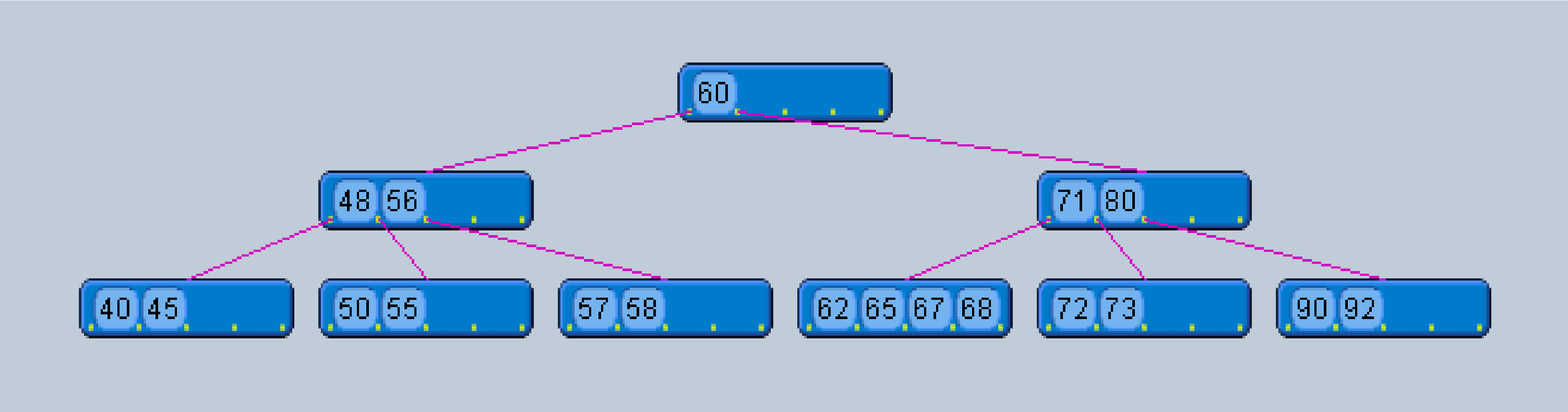
- grau= 3 = t (número de filhos em cada nodo ou grau)



- Alguns autores utilizam a palavra "grau" de uma B-tree para indicar o número máximo de chaves num nó. Outros autores utilizam a palavra "grau" para indicar o número máximo de filhos.

Exemplo de uma B-tree de grau 5

- **m = 5 (grau)**
- **n = 4 (chaves)**
- **T = 3 (grau mínimo)**
- **Nenhum nodo pode ter menos que 3 filhos ou 2 chaves.**

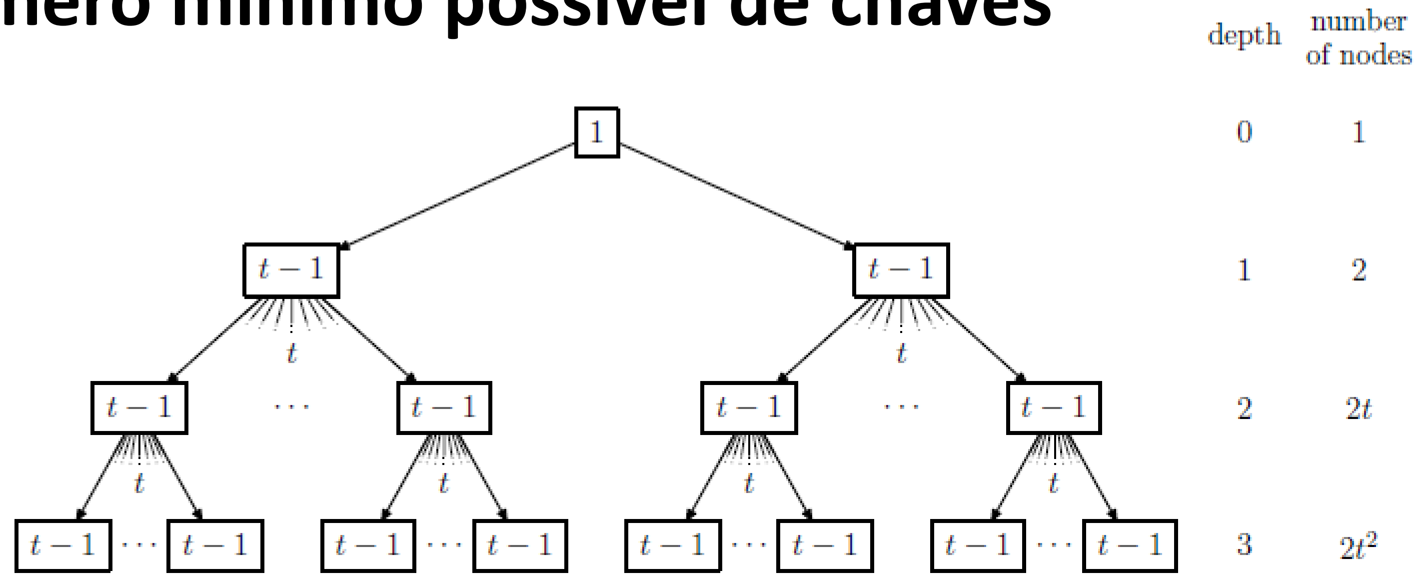


Aplicações

- Como cada nodo tende a ter um número grande de filhos, tipicamente é necessário passar por um número pequeno de nodos até localizar a chave desejada.
- Se o acesso a cada nodo requer um acesso a disco, então uma b-tree irá minimizar o número de acessos a disco necessários.
- O grau mínimo (fator de minimização) é usualmente escolhido de forma que o tamanho total de cada nodo corresponda a um múltiplo do tamanho de bloco do meio de armazenamento. Esta escolha simplifica e otimiza os acessos a disco.
- Consequentemente, uma b-tree é uma estrutura de dados apropriada para situações onde o conjunto completo de dados não podem residir na memória primária e acessos à memória secundária são muitos custosos.

Altura da B-tree

- **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 da B-tree

- 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)**

Busca em B-tree

- **Processo similar à busca em uma árvore binária.**
- **Comparação entre os valores da chave de busca e das chaves armazenadas na B-tree.**
- **Como as chaves estão ordenadas, pode-se realizar uma busca binária nos elementos de cada nó.**
  - Custo a busca binária: ***O(lg(t))***.
- **Se a chave não for encontrada no nó em questão, continua-se a busca nos filhos deste nó, realizando-se novamente a busca binária.**
- **Custo da busca completa**
  - $O(lg(t).log_t(n))$

Busca em B-tree

- **Considerando que os elementos dentro de um nó x da B-tree esteja organizado de forma linear, e que:**
  - n[x] = número de chaves carregadas correntemente no nó x
  - $chave_i[x]$ = valor da chave do nó x na posição x
  - folha[x] = retorna verdadeiro caso o nó seja folha
  - Disk-Read = operação de leitura do nó em disco
- **Um pseudo-código que implementa uma busca numa B-tree:**

Busca em B-tree

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

Procura pelo elemento k no nó

Se a chave k foi encontrada, então retorna

Caso não tenha sido encontrada, e nó igual a folha, então o elemento não está na B-Tree

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] e k > chave_i[x]
    do i ← i + 1
if i ≤ n[x] e k = chave_i[x]
    then return (x, i)
if folha[x]
    then return NIL
    else DISK-READ(c_i[x])
        return B-TREE-SEARCH(c_i[x], k)
```

Caso esteja num intervalo representado por um nó filho, continua a busca

Criando uma Árvore B

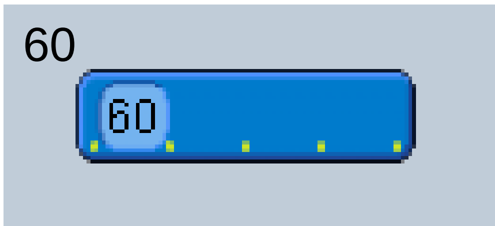
- **Allocate-Node() aloca uma página do disco para ser utilizada como um novo nó**
- **Requer O(1) operações de disco em um tempo de CPU O(1)**

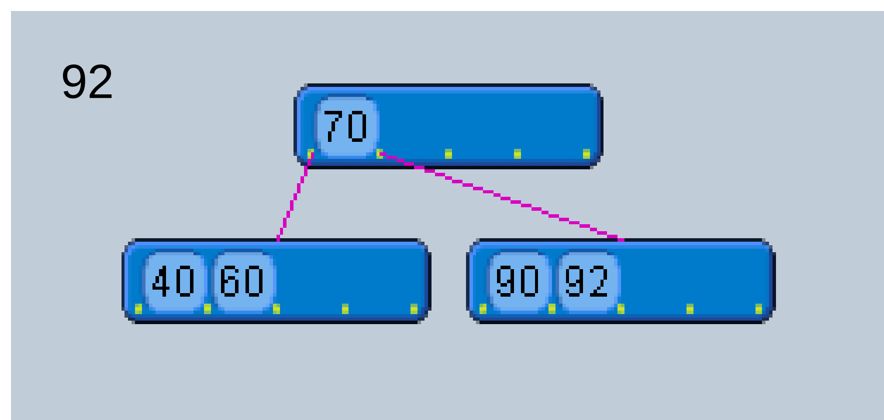
```
B-TREE-CREATE(T)
    x ← ALLOCATE-NODE()
    leaf[x] ← TRUE
    n[x] ← 0
    DISK-WRITE(x)
    root[T] ← x
```

Inserção em B-tree

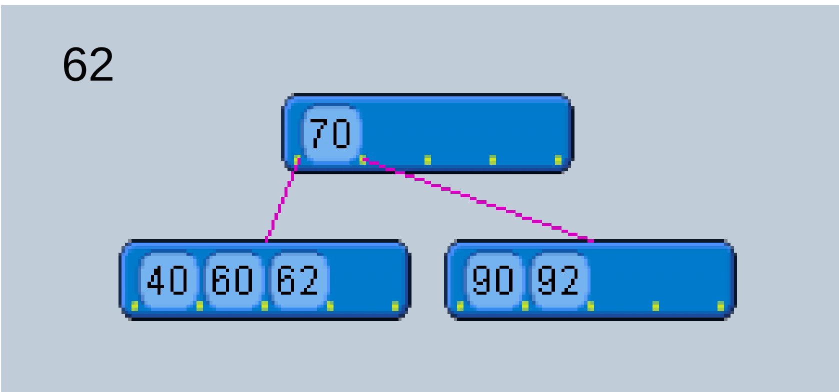
- **Localizar o nó folha *X* onde o novo elemento deve ser inserido.**
- **Se o nó *X* estiver cheio, realizar uma subdivisão de nós**
    - passar o elemento mediano de ***X*** para seu pai
    - subdividir ***X*** em dois novos nós com ***t - 1*** elementos
    - inserir a nova chave
- **Se o pai de *X* também estiver cheio, repete-se recursivamente a subdivisão acima para o pai de *X***
- **No pior caso terá que aumentar a altura da árvore B para poder inserir o novo elemento**

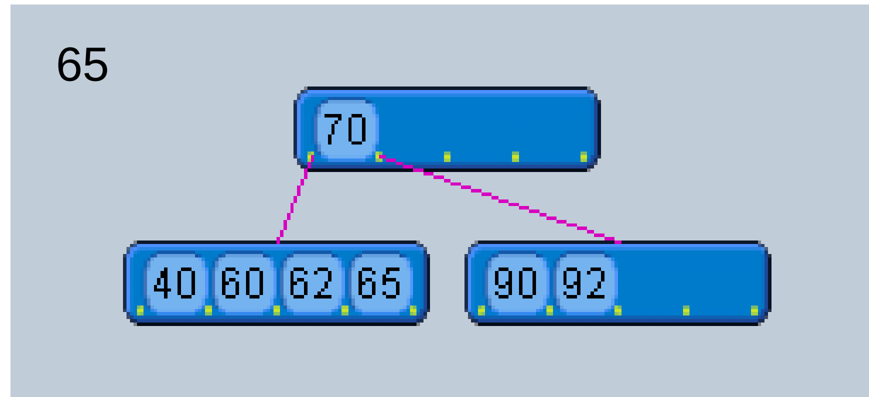
Exemplos de inserção em uma B-tree

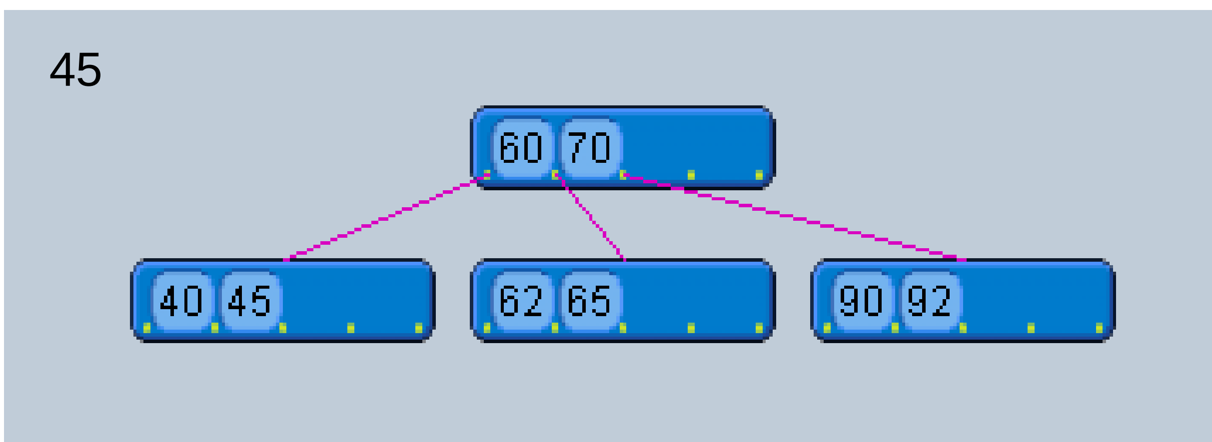




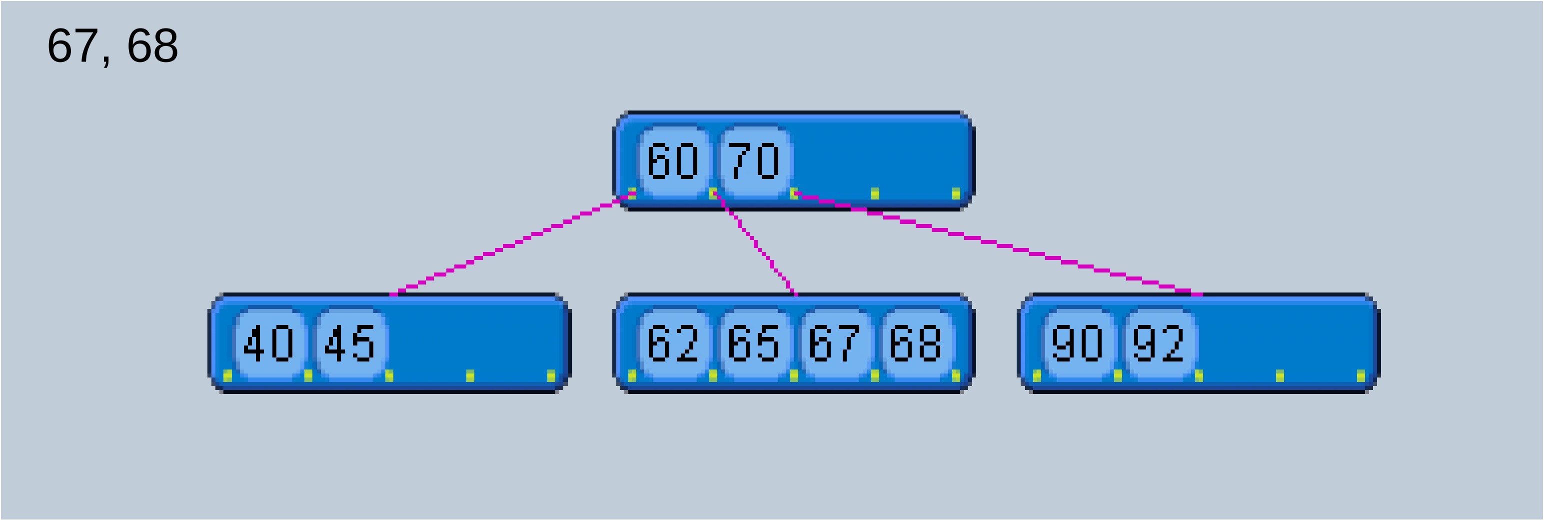
Exemplos de inserção em uma B-tree

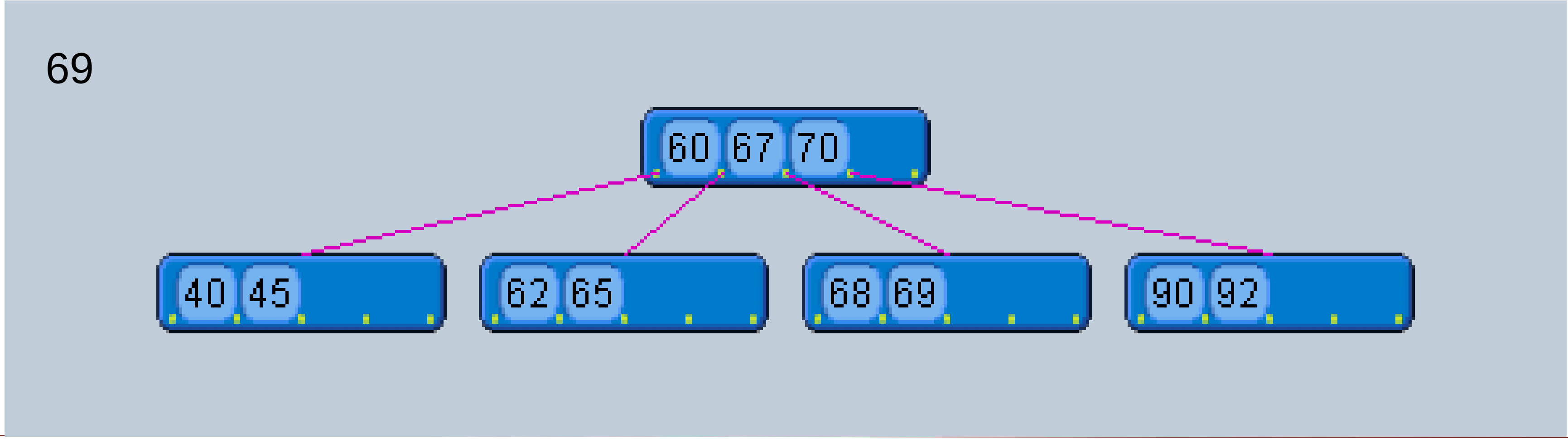






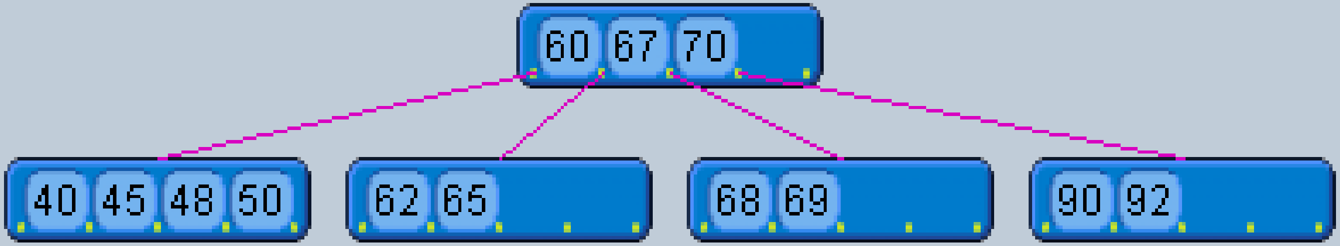
Exemplos de inserção em uma B-tree





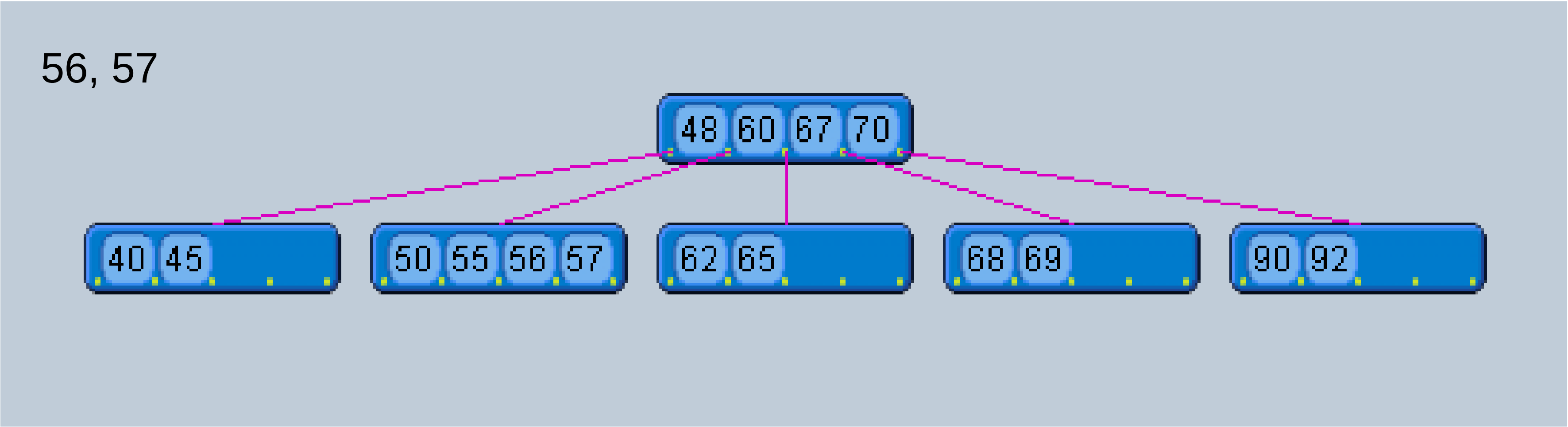
Exemplos de inserção em uma B-tree

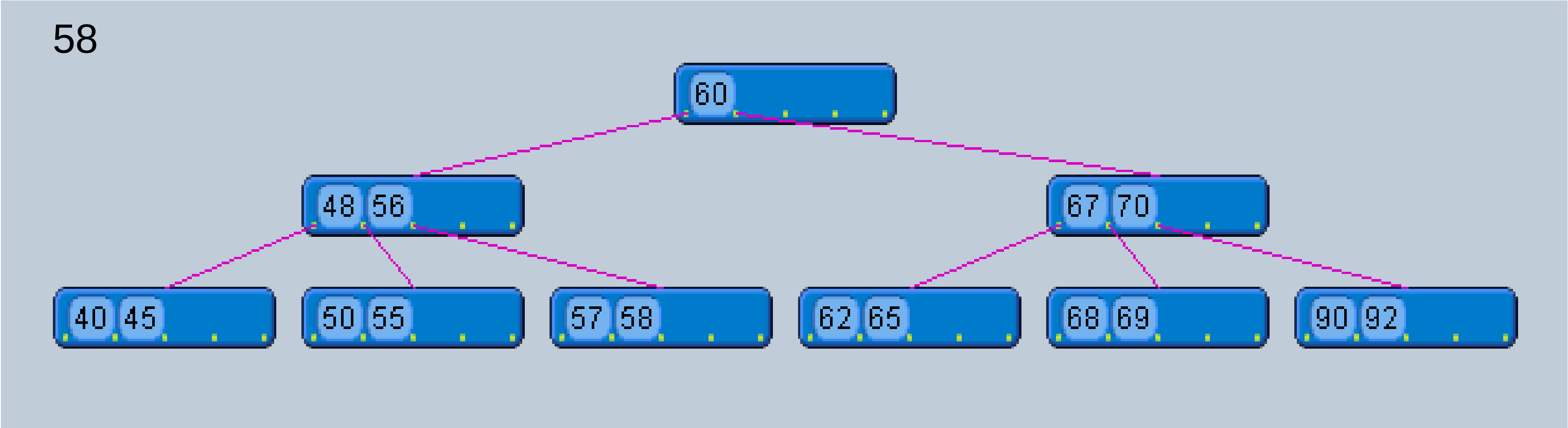
48, 50



55

Exemplos de inserção em uma B-tree





Inserção na B-Tree

```
B-TREE-INSERT(T, k)
r ← raiz[T]
if n[r] = 2t – 1
   then s ← ALLOCATE-NODE( )
        raiz[T] ← s
        folha[s] ← FALSE
        n[s] ← 0
        c_1[s] ← r
        B-TREE-SPLIT-CHILD(s, 1, r)
        B-TREE-INSERT-NONFULL(s, k)
   else B-TREE-INSERT-NONFULL(r, k)
```

Divisão de Nó

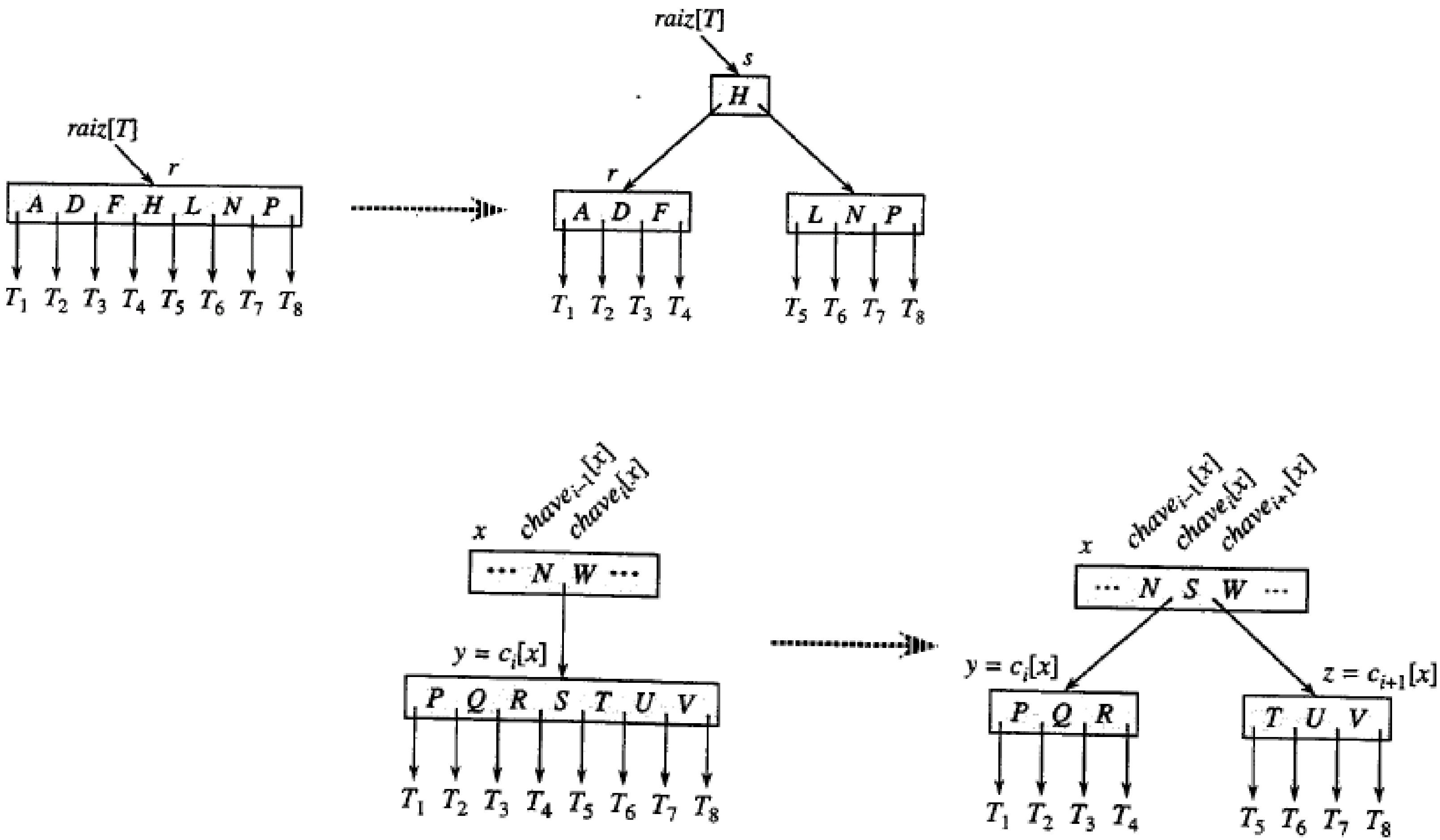


FIGURA 18.5 A divisão de um nó com $t = 4$. O nó y é dividido em dois nós, y e z, e a chave mediana S de y é movida para cima até o pai de y

Divisão de Nó

3 entradas:

- *x, um nó interno não completo*
- *i, um índice*
- *y, um nó dado que y = ci[x] é um nó completo, filho de x*

```
B-TREE-SPLIT-CHILD(x, i, y)
z ← ALLOCATE-NODE( )
folha[z] ← folha[y]
n[z] ← t – 1
for j ← 1 to t – 1
    do chave_j[z] ← chave_{j+t}[y]
if not folha[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 chave_{j+1}[x] ← chave_j[x]
chave_i[x] ← chave_t[y]
n[x] ← n[x] + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

Cria um novo nó z

Atribui os t-1 maiores elementos a z

Se o nó y, a ser dividido, não era folha, então também copia os t filhos de y para z

Translada valores de x para receber mediana de y

Atualiza a qtd de elementos do no y

Translada as chaves de x para receber z (z passa a ser filho de x)

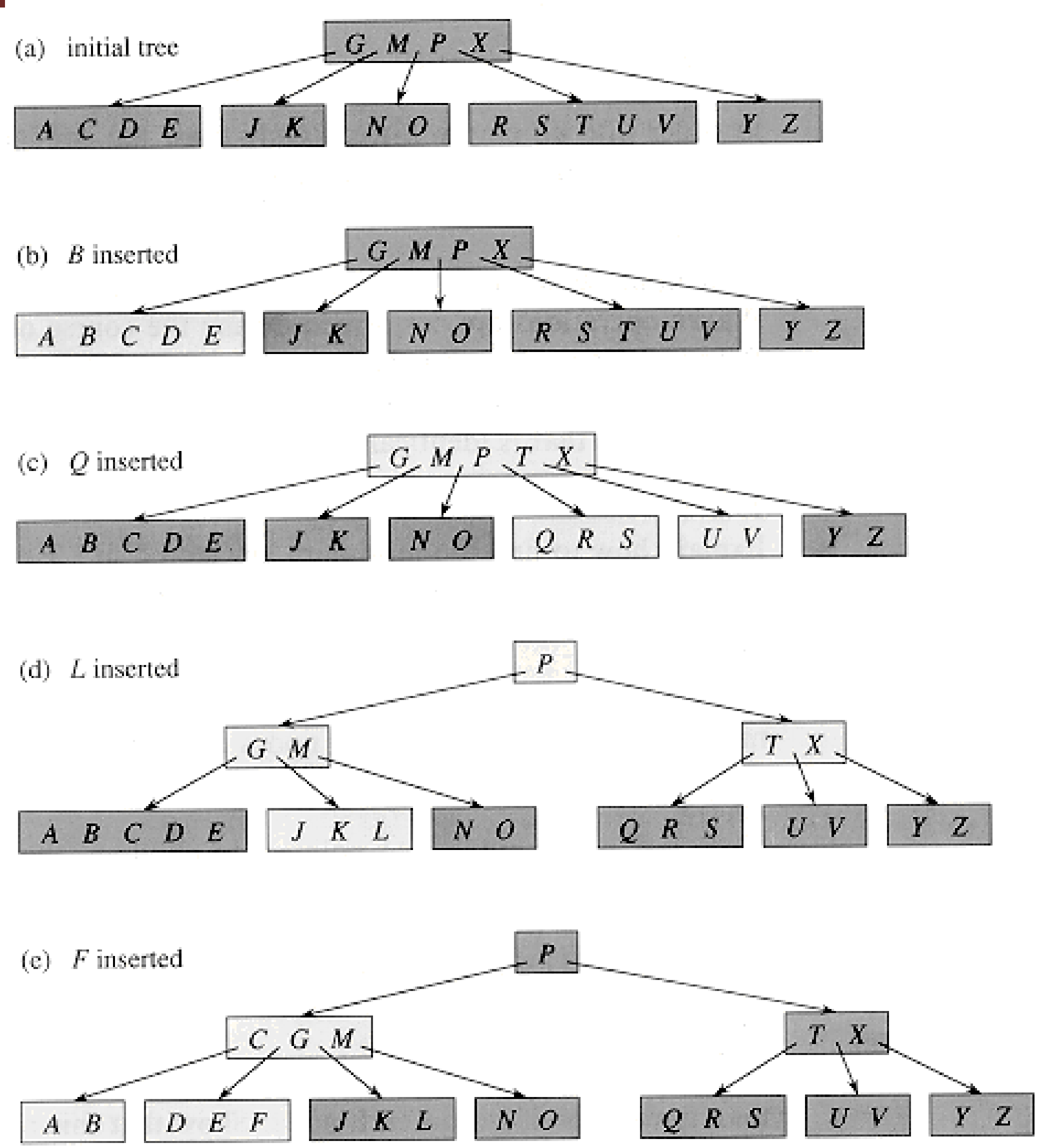
Grava tudo em disco

Inserção na B-Tree

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if folha[x]
    then while i ≥ 1 e k < chave_i[x]
            do chave_{i+1}[x] ← chave_i[x]
               i ← i − 1
         chave_{i+1}[x] ← k
         n[x] ← n[x] + 1
         DISK-WRITE(x)
    else while i ≥ 1 e k < chave_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 > chave_i[x]
                    then i ← i + 1
         B-TREE-INSERT-NONFULL(c_i[x], k)
```

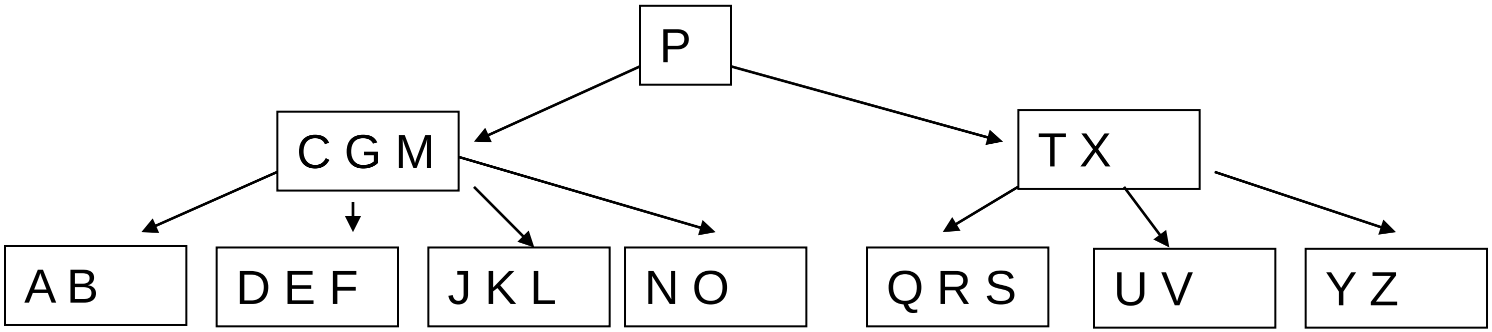
Caso em que x é um nó de folha
Linha: empurra as chaves maiores que k para à direita

Inserção: exemplo



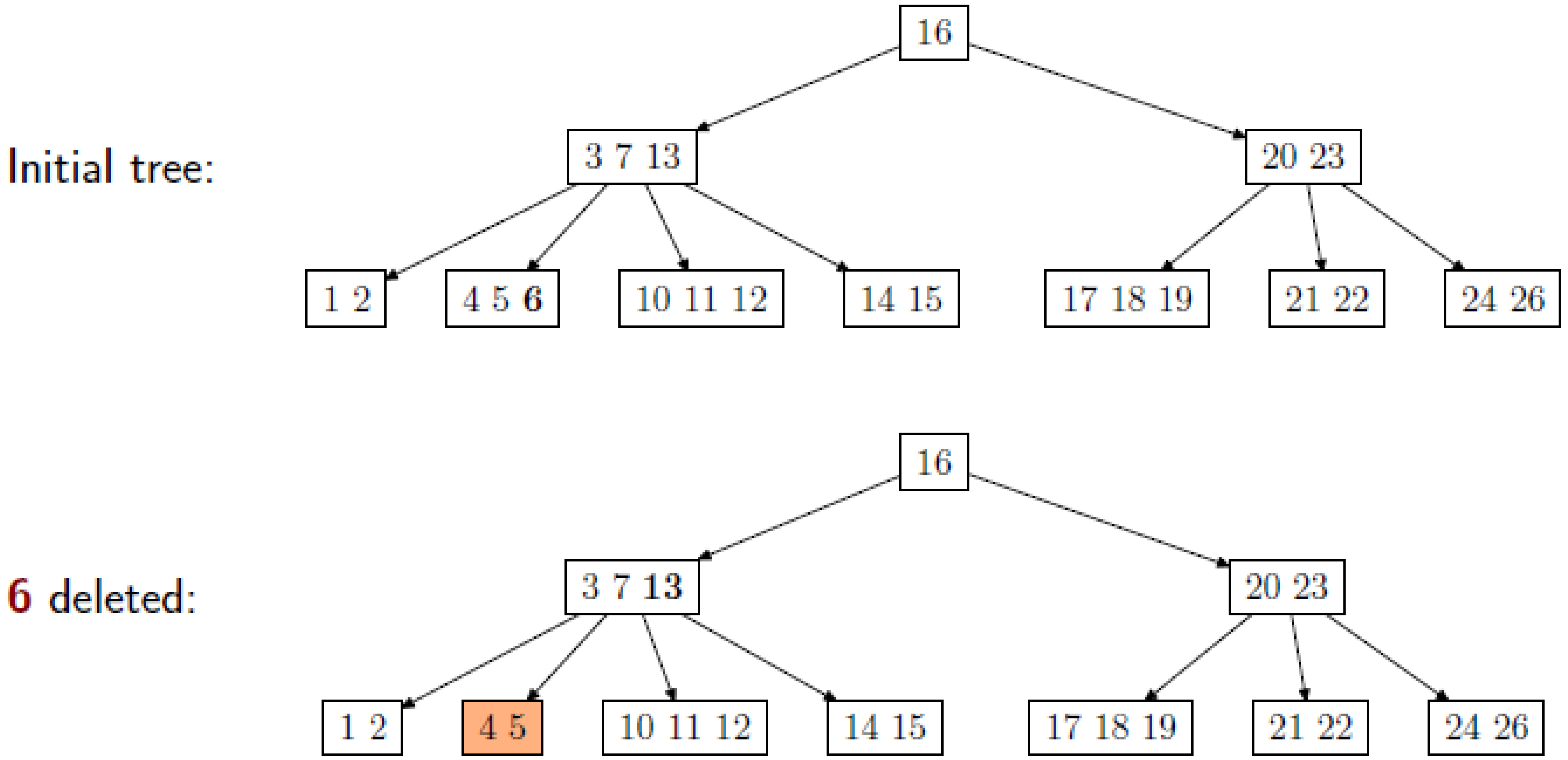
Remoção em B-tree

- **Processo um pouco mais complicado que inserção**
- **Exemplos: árvore inicial**



Remoção em B-Tree

1. **Se a chave k está no nó x e x é uma folha, elimine a chave k de x**

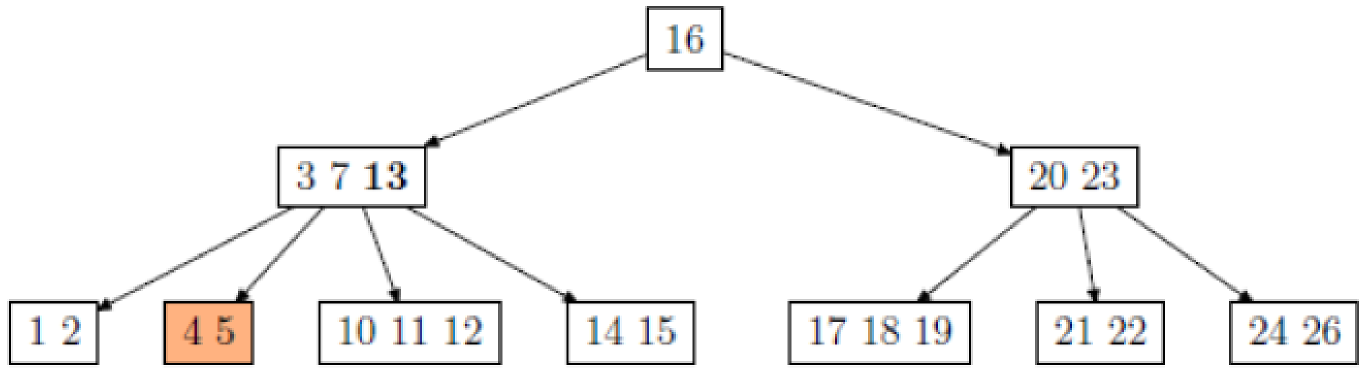


Remoção em B-Tree

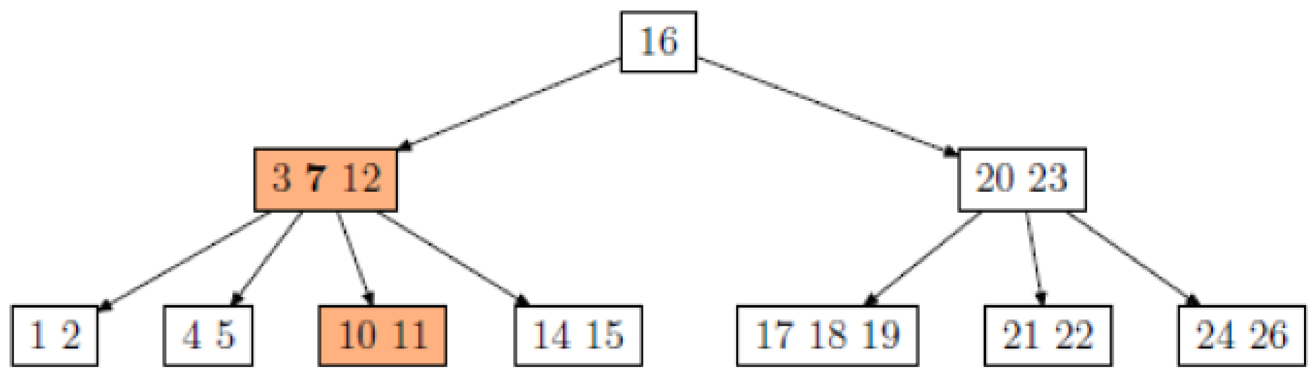
2. **Se a chave k está no nó x e x é um nó interno então:**
   a. Se o filho y que procede k no nó x tem pelo menos t chaves, então encontre o predecessor k’ de k na subarvore com raiz em y. Elimine k’, e substitua k por k’ em x
   b. Simetricamente, se o filho z que segue k no nó x tem pelo menos t chaves, então encontre o sucessor k’ de k na subarvore com raiz em z. Elimine k’, e substitua k por k’ em x

Remoção em B-Tree

Initial tree:



13 deleted:



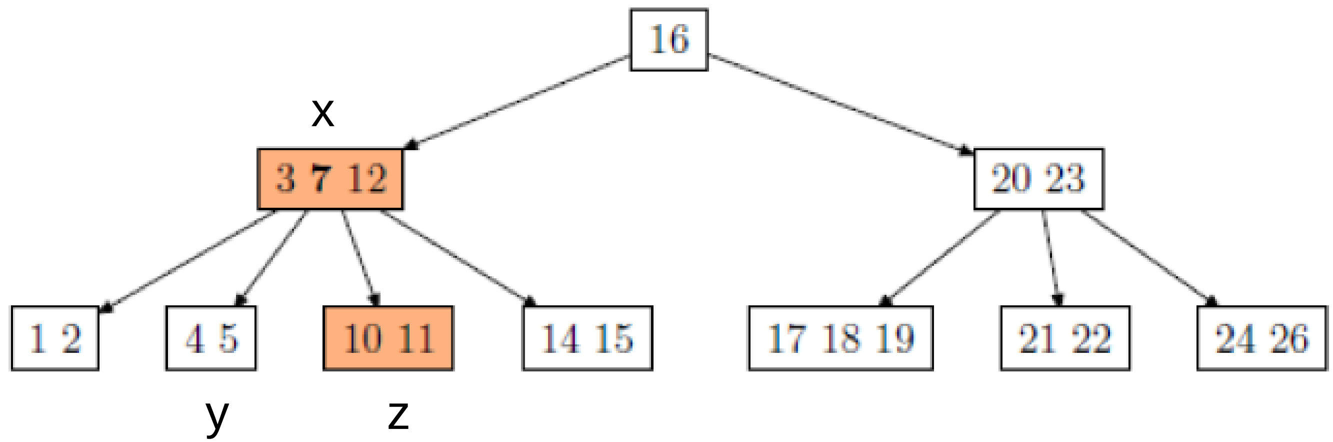
- Caso 2a é ilustrado. O predecessor de 13, que se situa no filho precedente de *x*, é movido para cima e ocupa a posição de 13. O filho predecessor tinha uma chave sobrando nesse caso.

Remoção em B-Tree

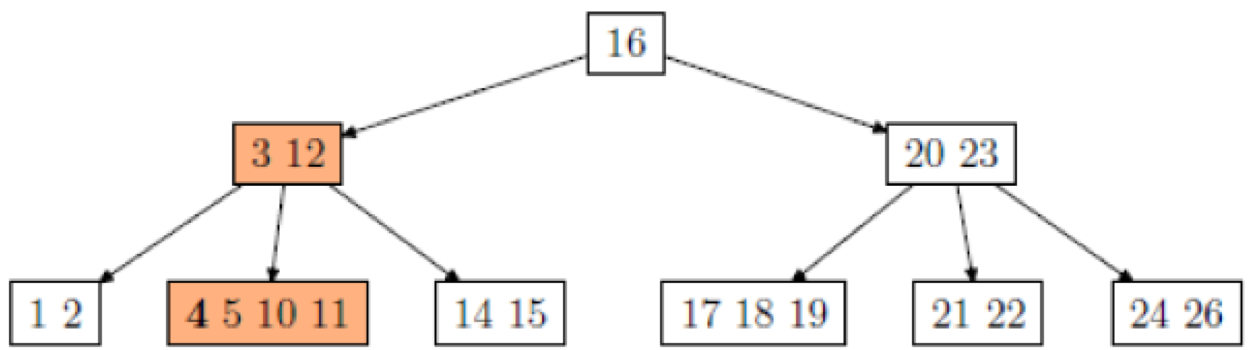
c. Caso contrário, se tanto y quanto z tem apenas t-1 chaves, faça a intercalação de k e todos os itens z em y, de modo que x perca tanto k quanto o ponteiro para z, e y contenha agora 2t -1 chaves. Em seguida, libere z e retire k

Remoção em B-Tree

Initial tree:



7 deleted:



- 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 em B-tree

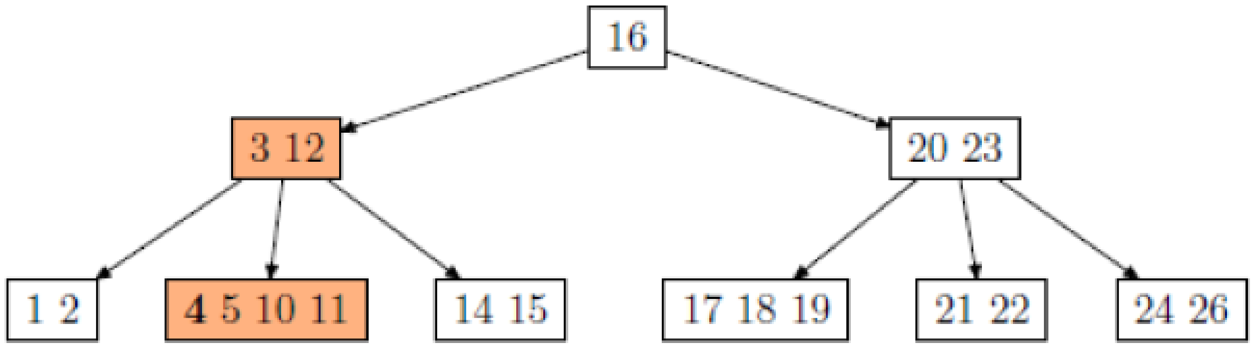
3. Se a chave k não estiver presente no nó interno x, determine a raiz ci[x] da subarvore apropriada que deve conter k, se k estiver absolutamente na árvore. Se ci[x] tiver somente t-1 chaves, execute o passo 3a ou 3b conforme necessário para garantir que desceremos até um nó contendo pelo menos t chaves

B-tree Remoção

a. Se ci[x] tiver somente t-1 chaves, mas tiver um irmão com t chaves, forneça a ci[x] uma chave extra, movendo uma chave de x para baixo até ci[x], movendo uma chave do irmão esquerdo ou direito imediato de ci[x] para dentro de x, e movendo o ponteiro do filho apropriado do irmão para ci[x]
b. Se ci[x] e todos os irmãos de ci[x] tem t-1 chaves, faça a intercalação de ci[x] com um único irmão, o que envolve mover a chave de x para baixo até o novo nó intercalado, a fim de se tornar a chave mediana para este nó

B-tree Remoção Caso 3b

Initial tree: Key 4 to be deleted



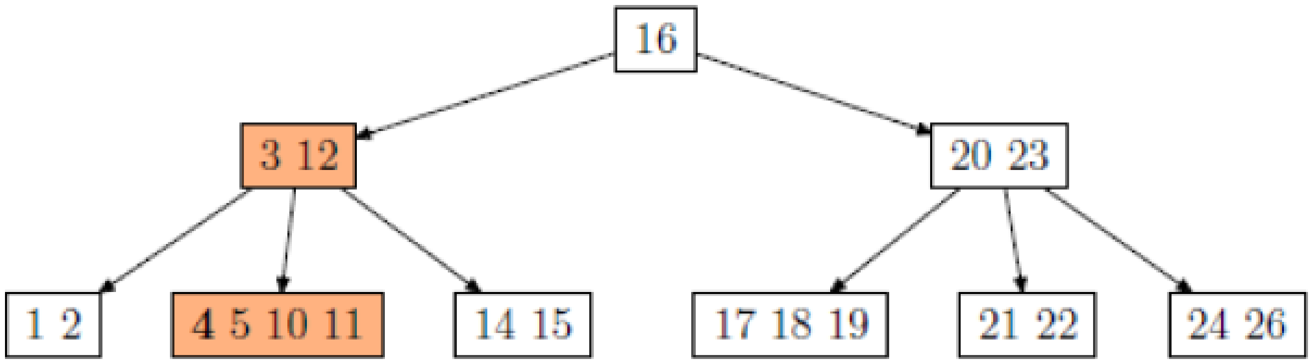
• A recursão aqui não pode descer do nó 3, 12 porque ele tem *t-1* chaves. No caso das duas folhas da esquerda e da direita terem mais que *t-1* chaves, 3, 12 poderia obter uma e o 3 ser movido para baixo

• 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

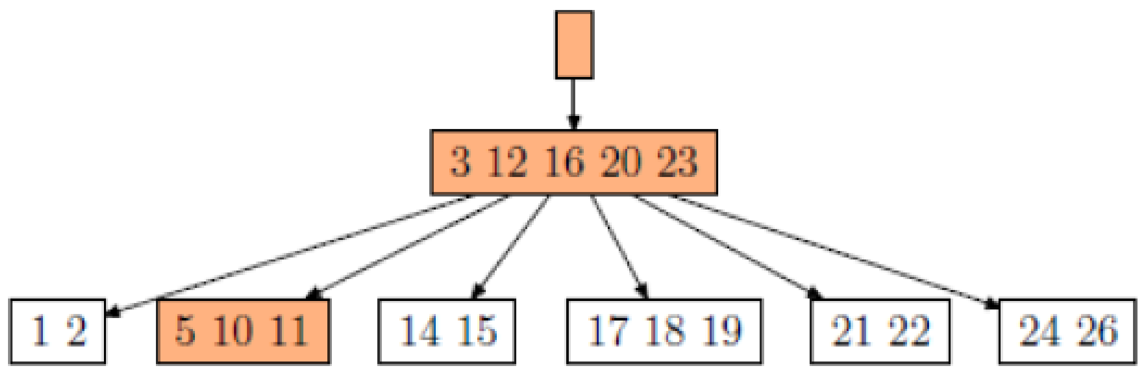
• Portanto, a raiz deve ser empurrada para baixo e intercalada com seus dois filhos, assim o 4 pode ser excluído com segurança

B-Tree Remoção Caso 3b

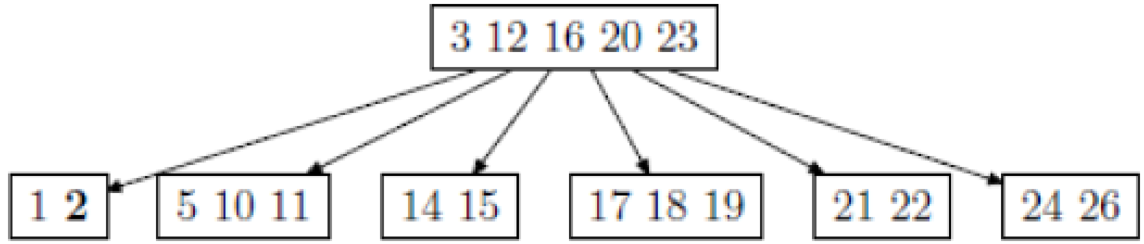
Initial tree:



4 deleted:

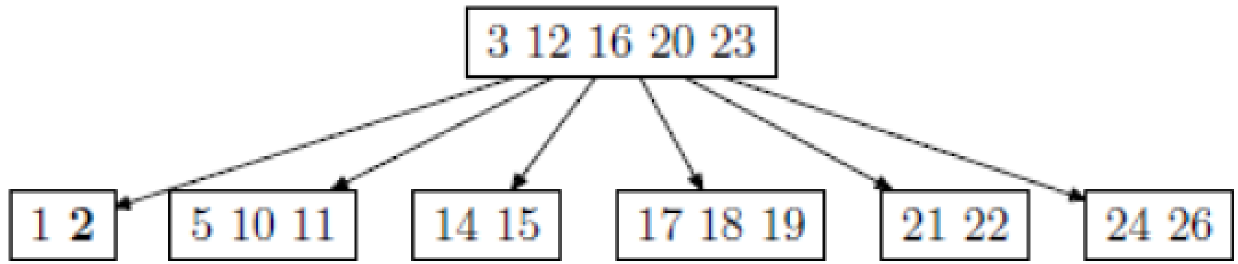


Outcome:



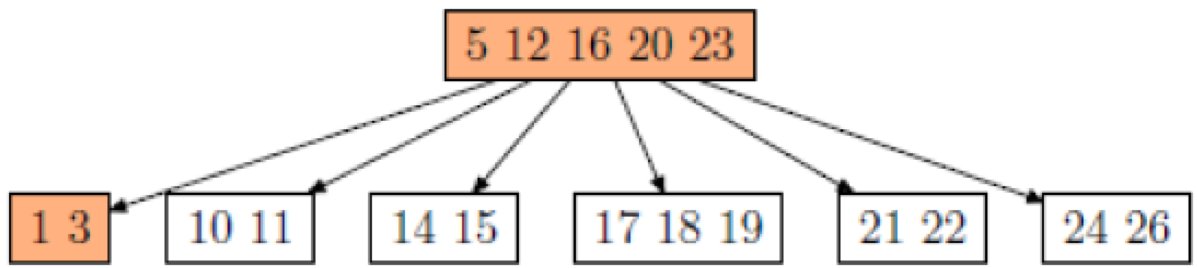
B-Tree Remoção Caso 3a

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)

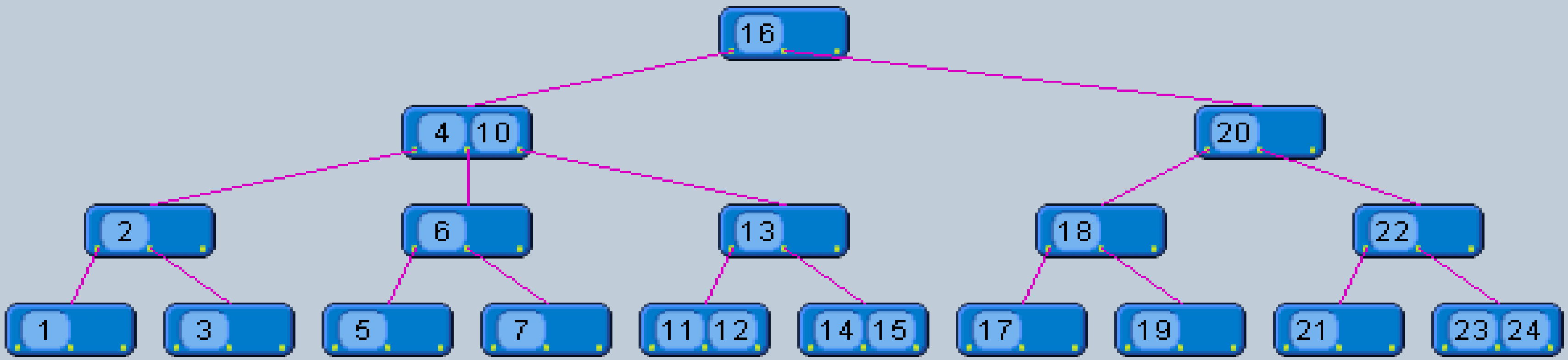
else (leaf node)
  $y \leftarrow$ PRECEDING-CHILD(x)
  $z \leftarrow$ SUCCESSOR-CHILD(x)
  $w \leftarrow \text{root}(x)$
  $v \leftarrow RootKey(x)$
    if $n[x] > t - 1$ then REMOVE-KEY(k, x) // Caso 1
    else if $n[y] > t - 1$ then // Caso 3a
      $k' \leftarrow$ FIND-PREDECESSOR-KEY(w, v)
     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

Remoção de Elementos – Pseudocódigo (3)

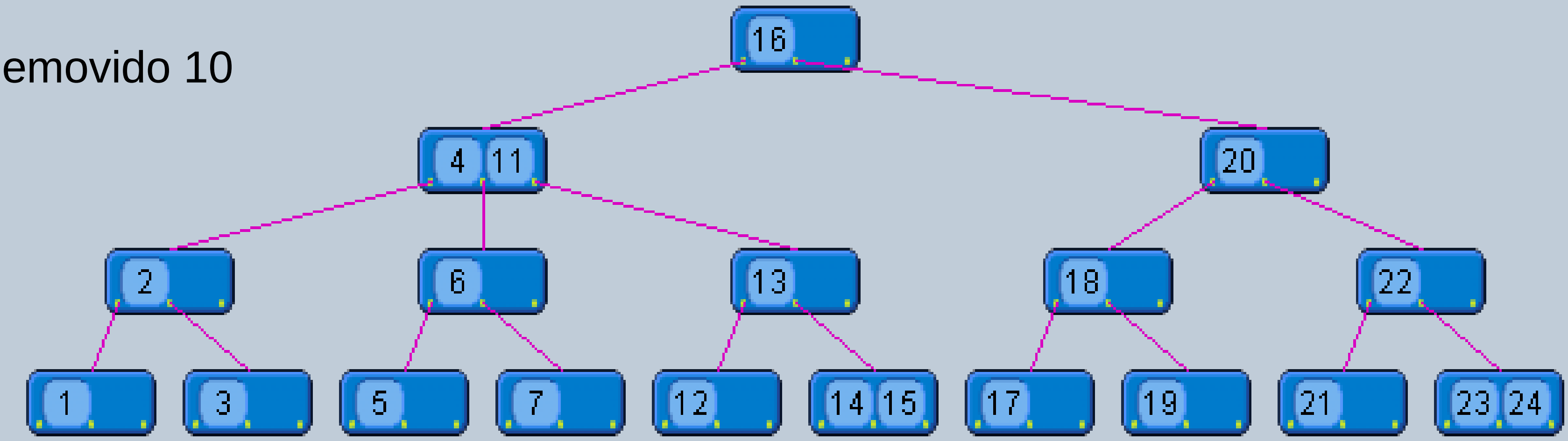
```
else                                    // Caso 3b
    s ← FIND-SIBLING(w)
    w′ ← root(w)
        if n[w′] = t − 1 then
            MERGE-NODES(w′, w)
            MERGE-NODES(w, s)
            B-TREE-DELETE-KEY(k, x)
        else
            MOVE-KEY(v, w, x)
            B-TREE-DELETE-KEY(k, x)
```

- 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.

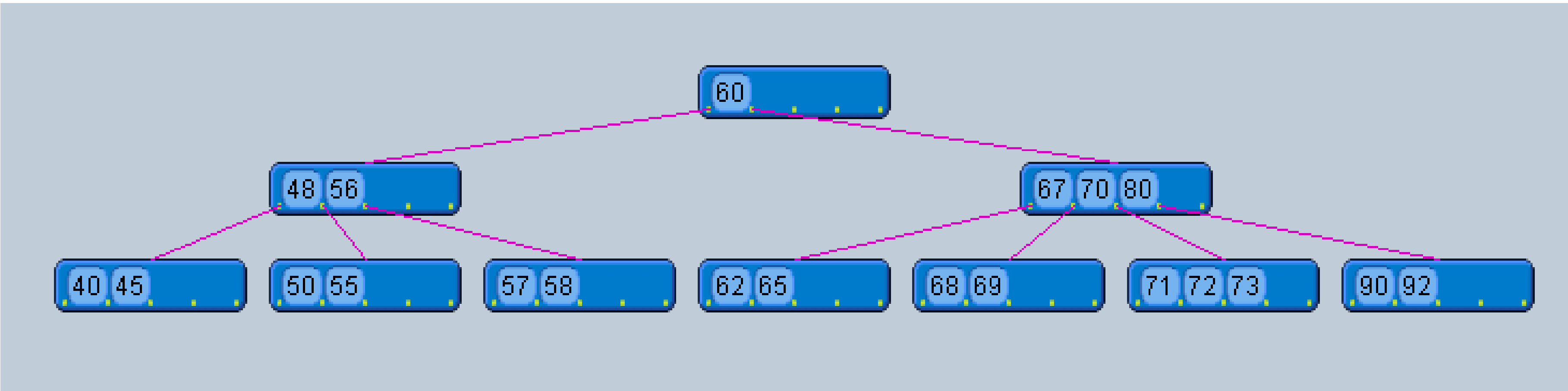
Remover o valor 10

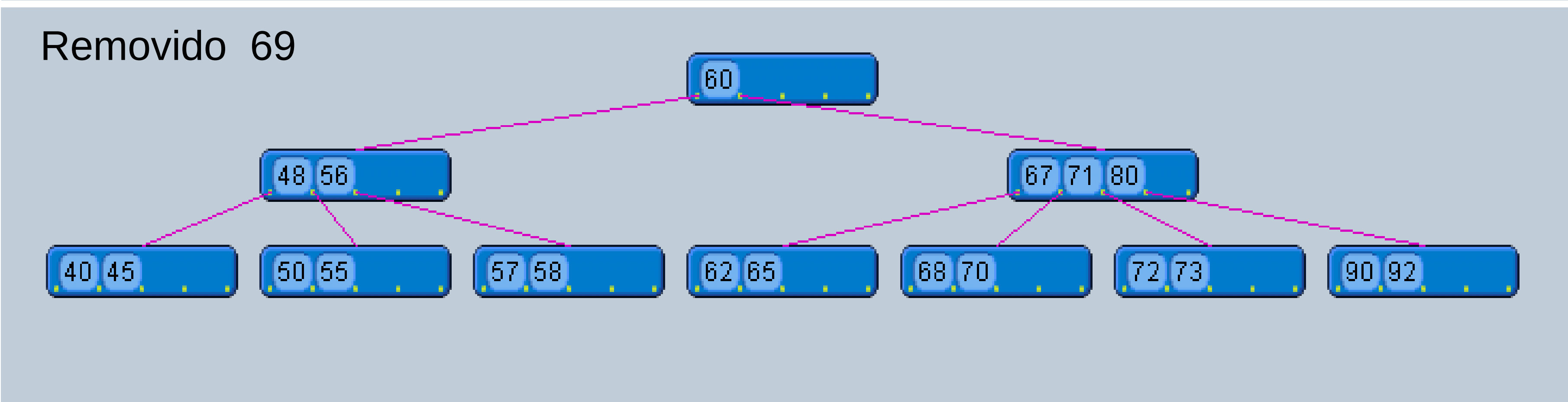


Removido 10

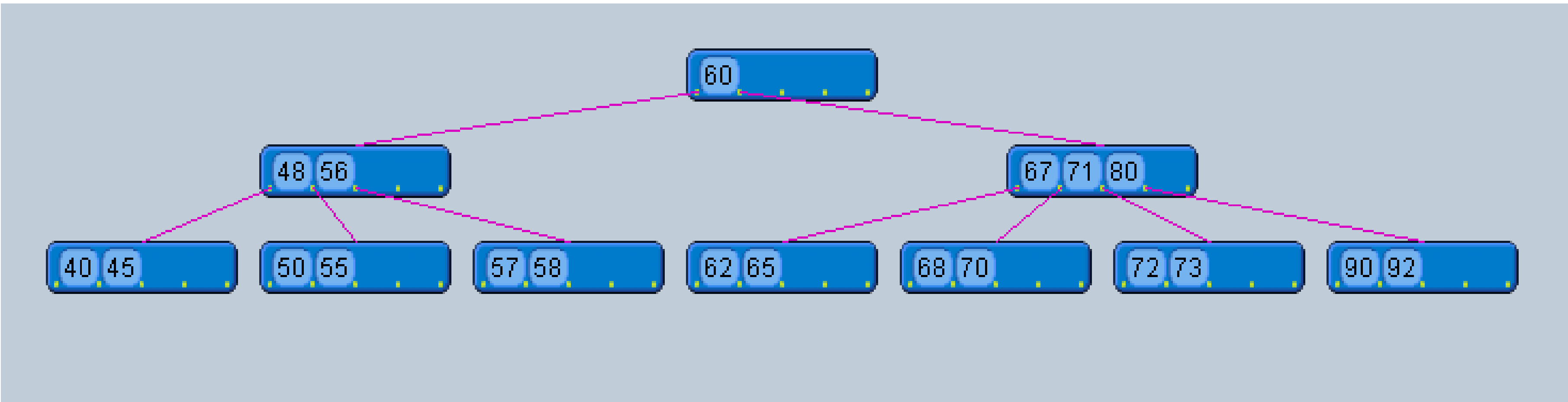


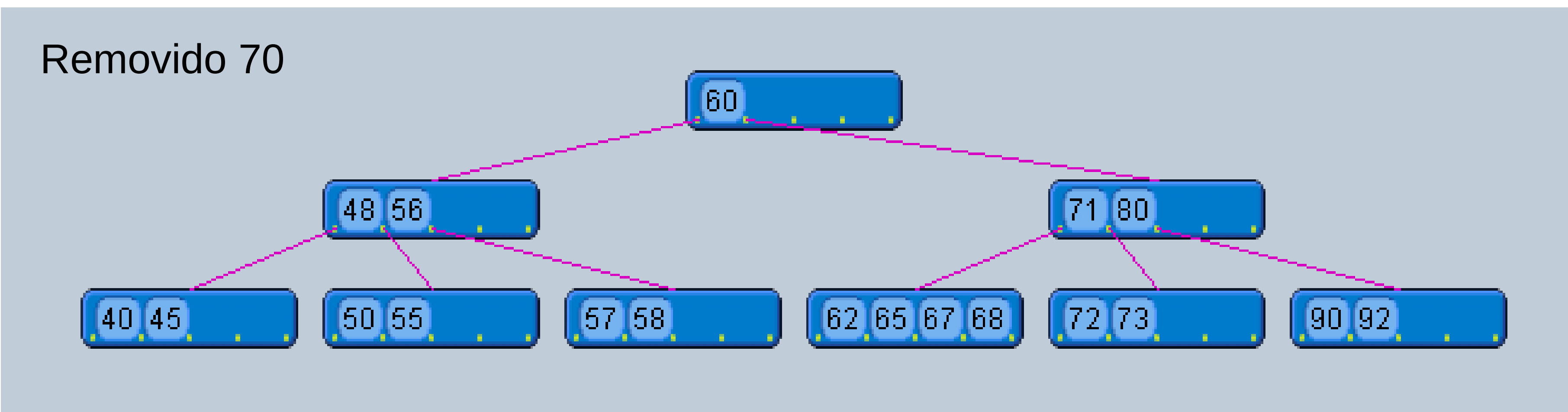
Remover o valor 69



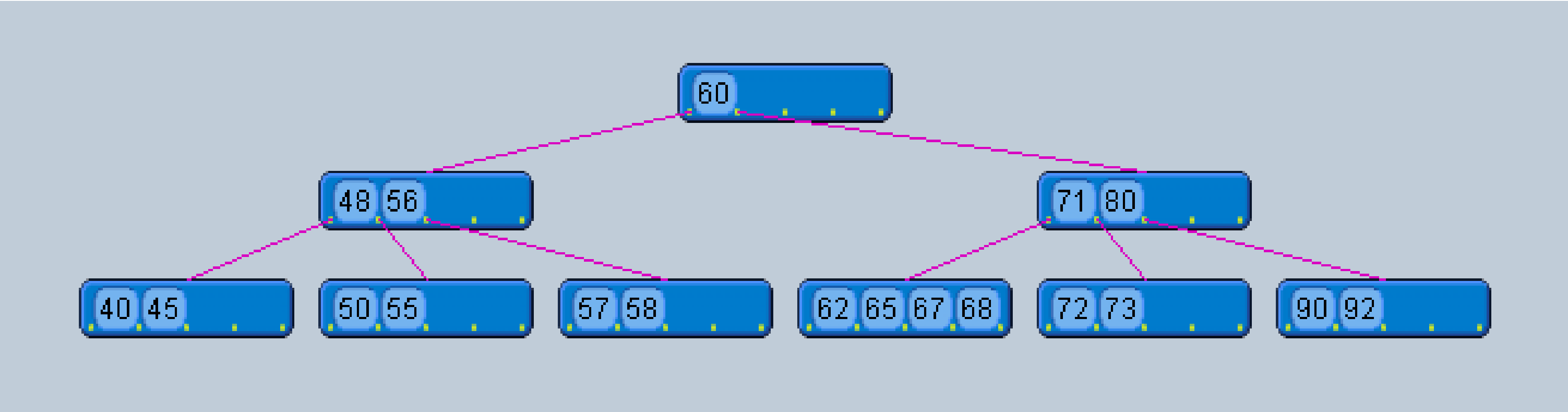


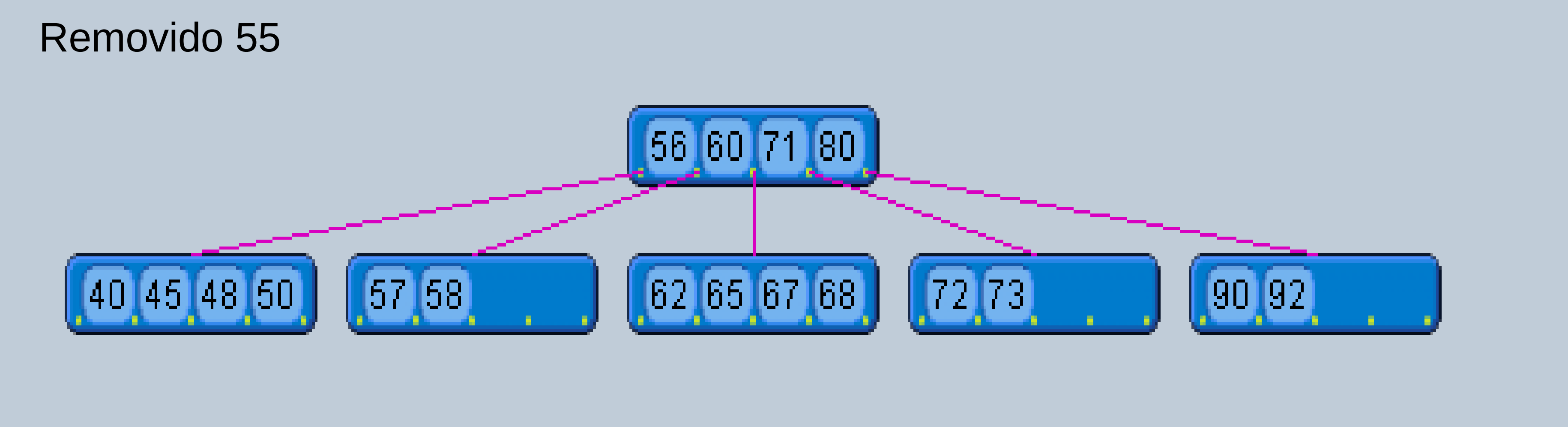
Remover o valor 70





Remover o valor 55





Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein, et al., *Introduction to Algorithms*, 2nd Edition, MIT Press, 2001.
- Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. Disponível em: http://www.brpreiss.com/books/opus4/html/page340.html#SECTION0011700000000000000000
- http://www.bluerwhite.org/btree/
- animação em Java: http://slady.net/java/bt/

Governo Federal

Ministério da Educação

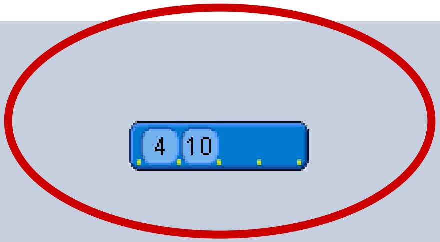
deINF departamentodeInformática

Universidade Federal do Maranhão

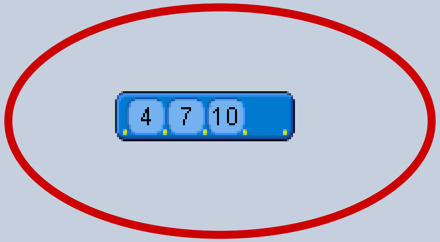
A Universidade que Cresce com Inovação e Inclusão Social

Cenários

Inserção numa B-tree

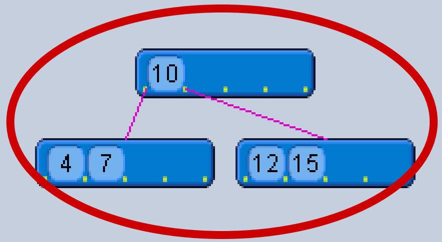


Estado inicial

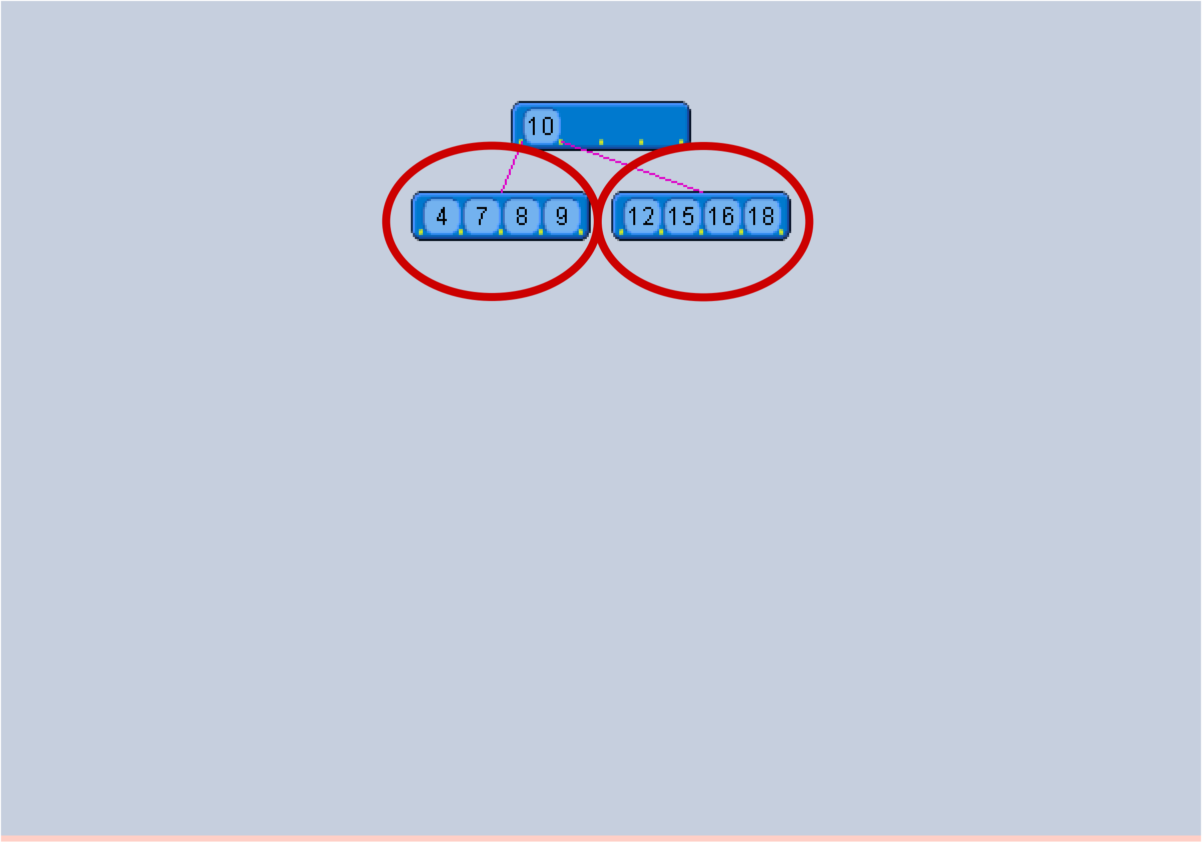


Inserindo o valor 10

Inserindo o valor 12 e 15

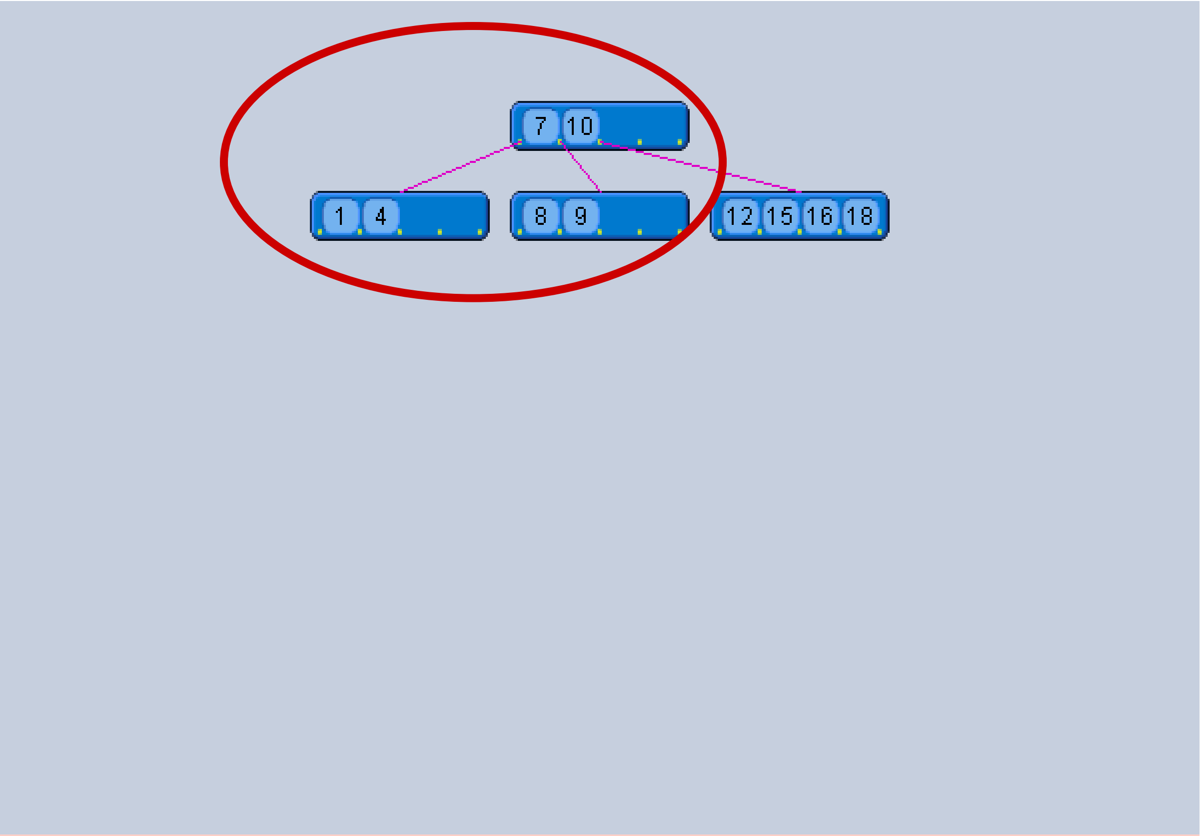


Inserindo o valor 12 e 15

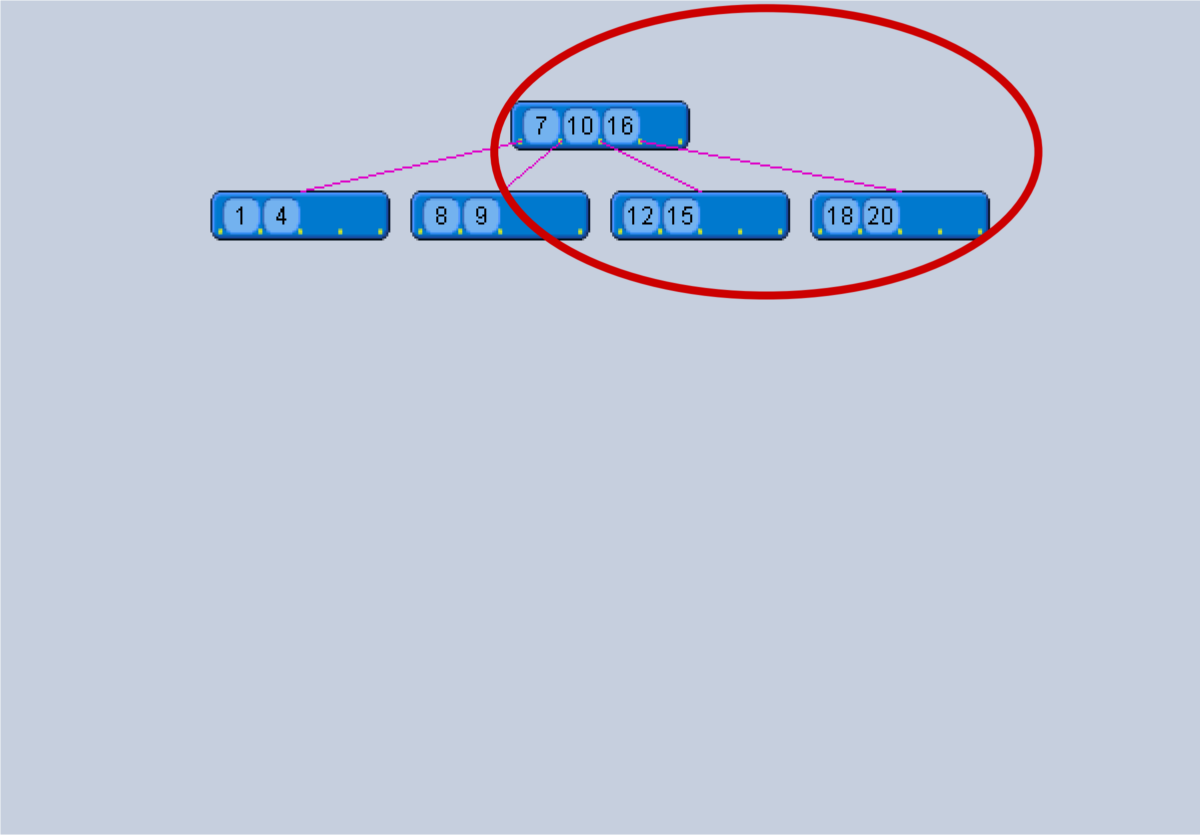


Inserindo o valor 8,9,16 e 18

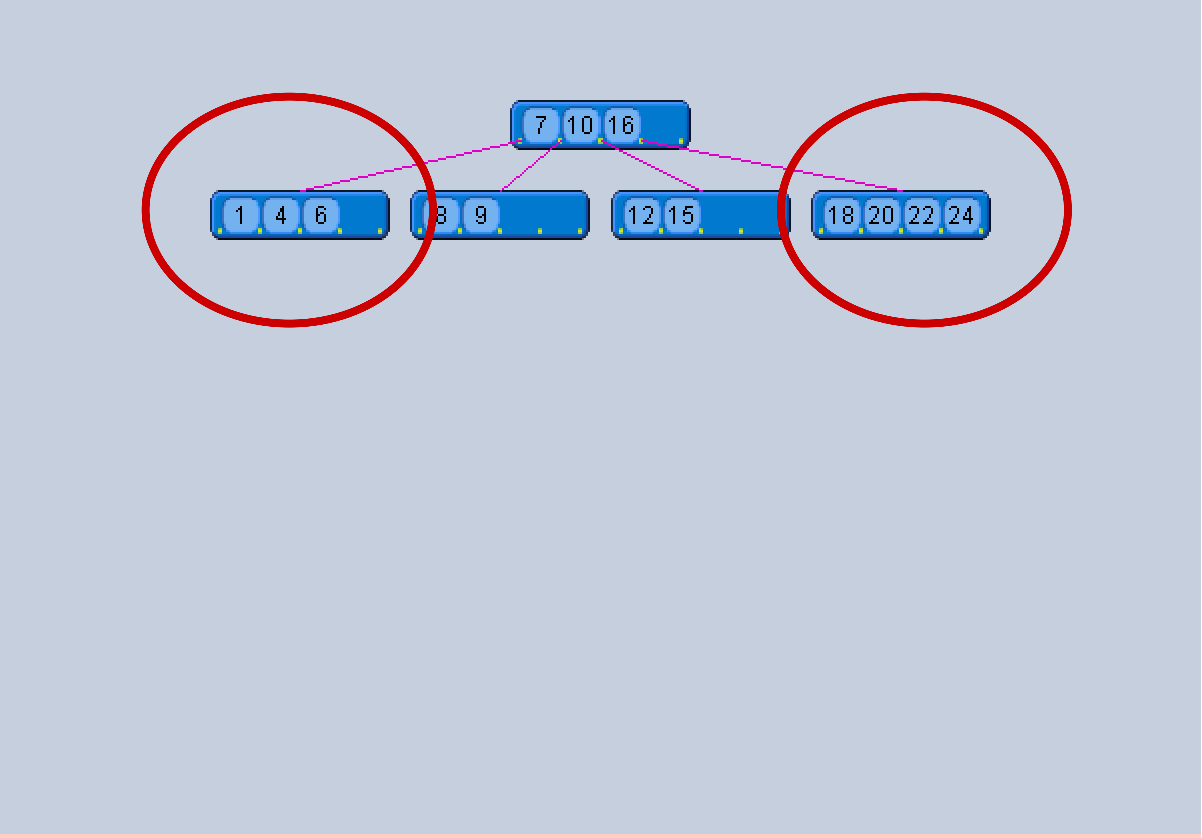
Inserindo o valor 1



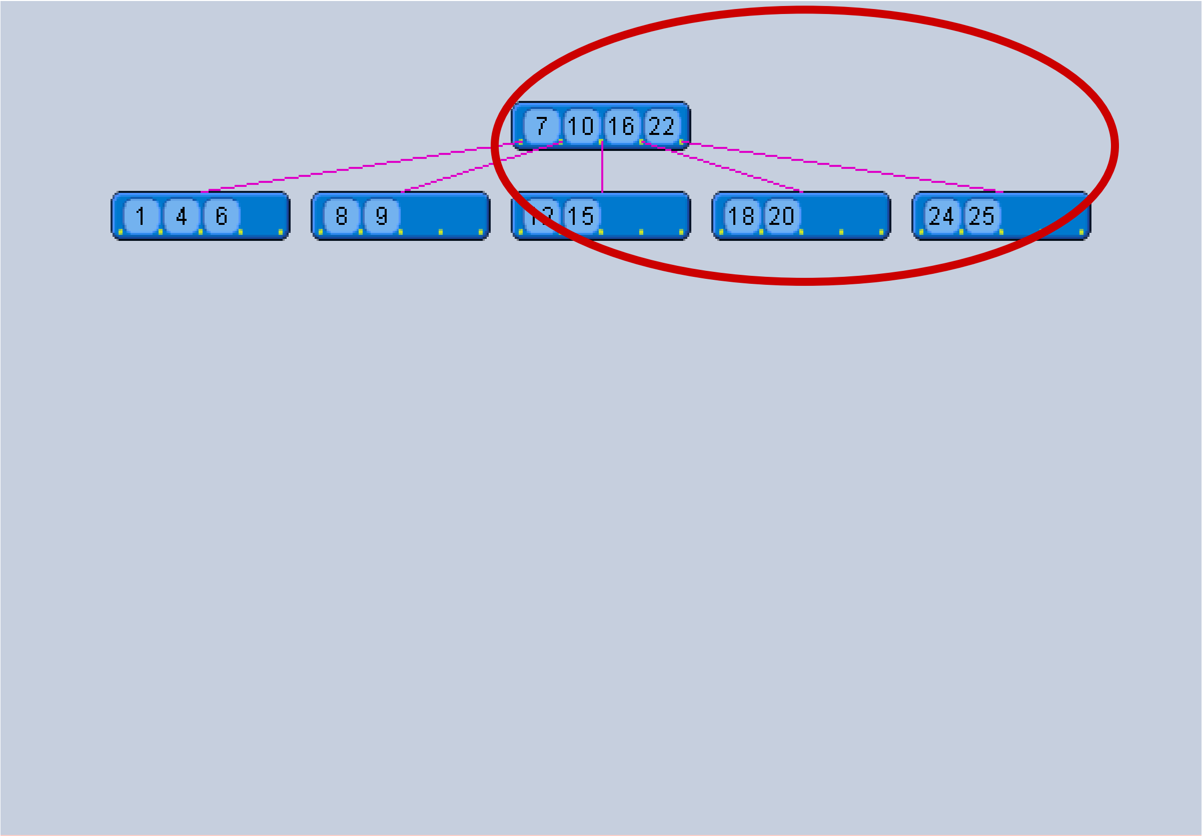
Inserindo o valor 1



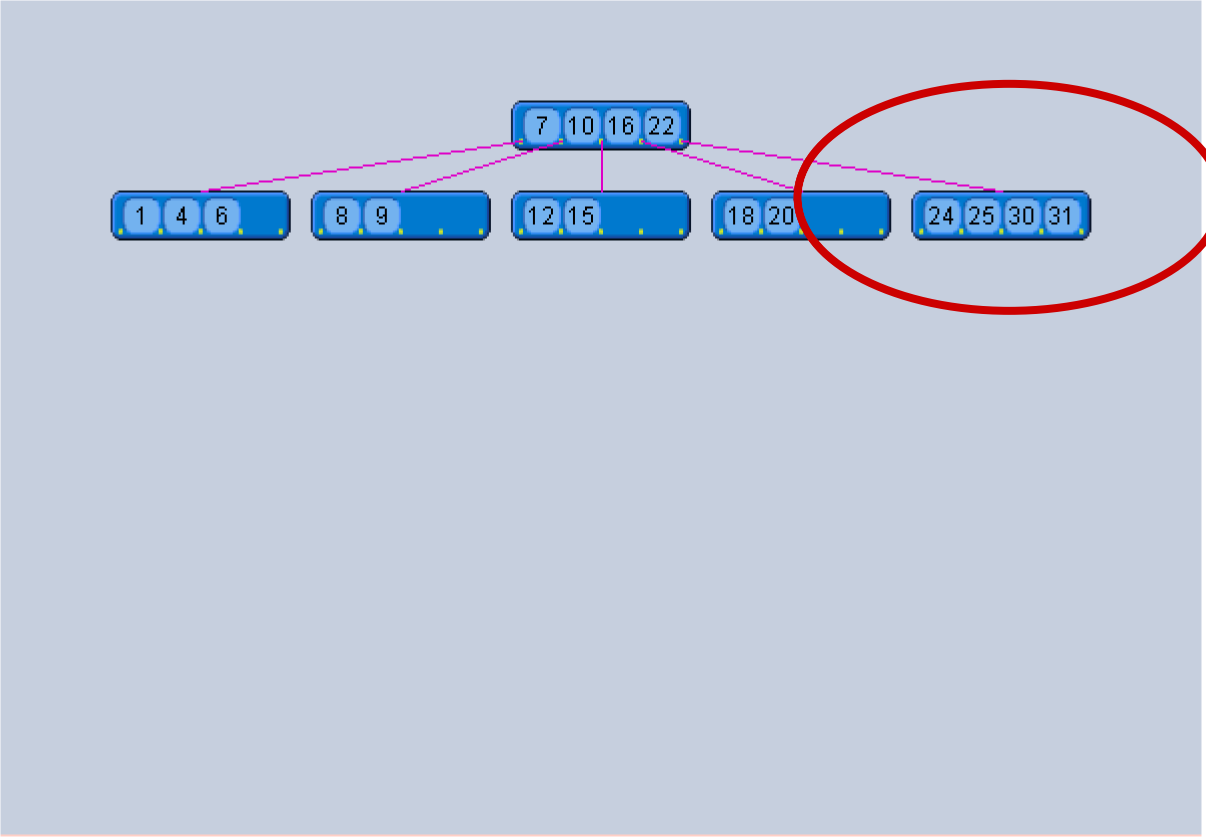
Inserindo o valor 20



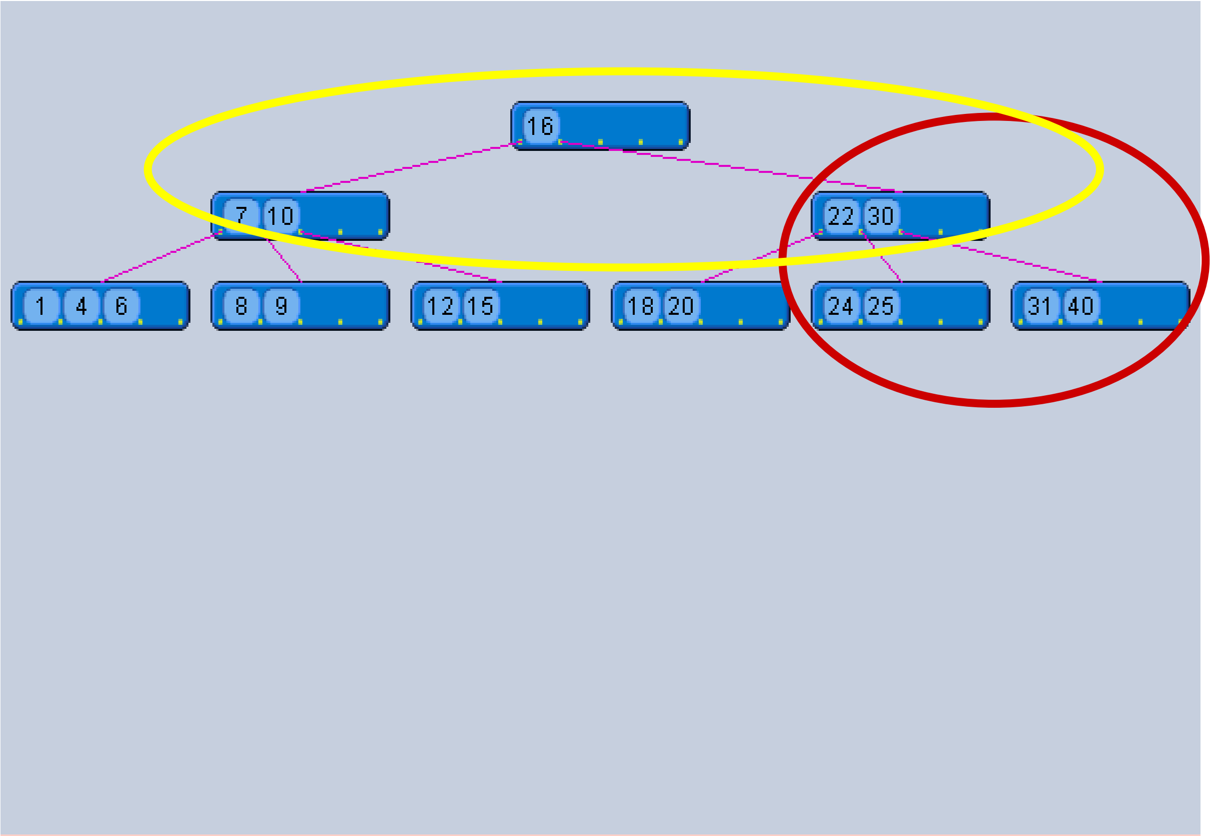
Inserindo o valor 6,22 e 24



Inserindo o valor 25



Inserindo o valor 30,31



Inserindo o valor 40

Governo Federal

Ministério da Educação

deINF departamentodeInformática

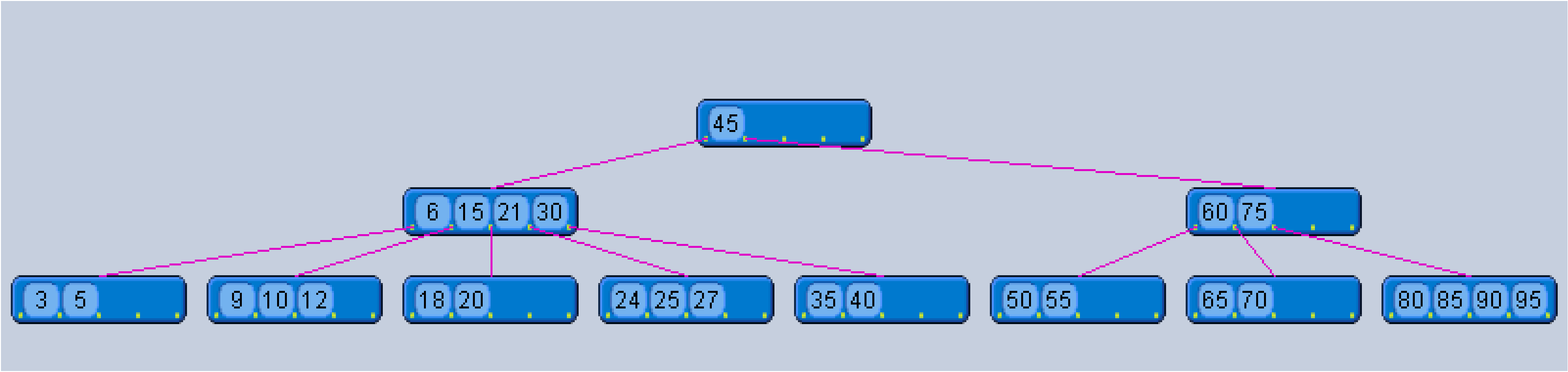
Universidade Federal do Maranhão

A Universidade que Cresce com Inovação e Inclusão Social

Cenários

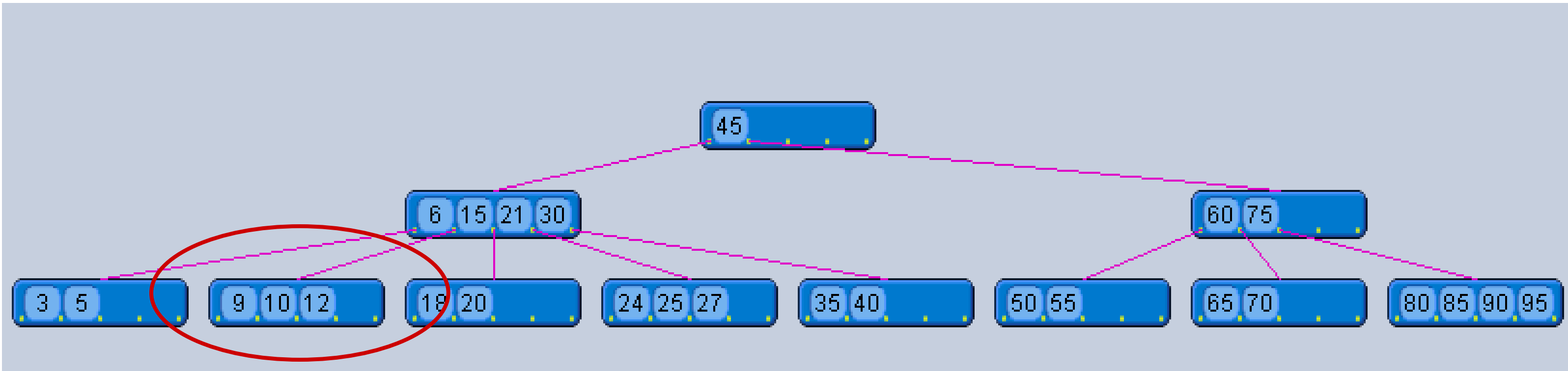
Remoção numa B-tree

Partindo da árvore B

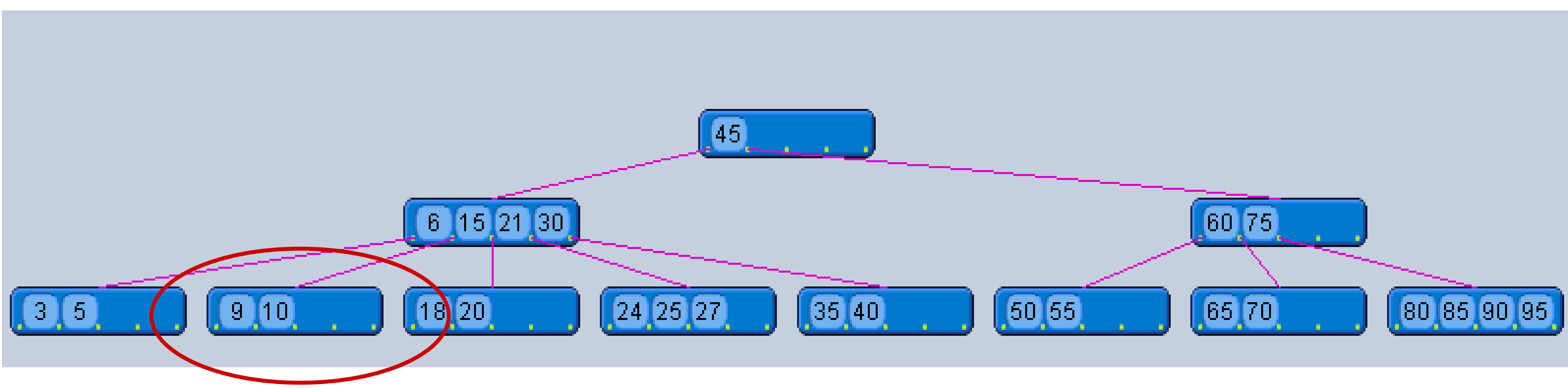


Regras

1. **Caso seja retirado de uma folha X:**
    a. se depois da retirada a folha tiver pelo menos t elementos, então tudo bem

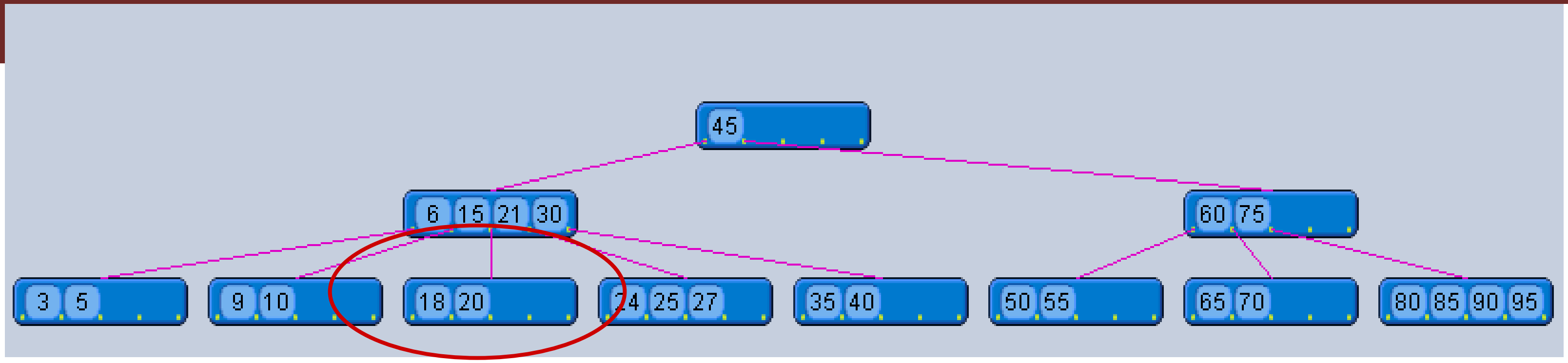


Removendo o valor
12.

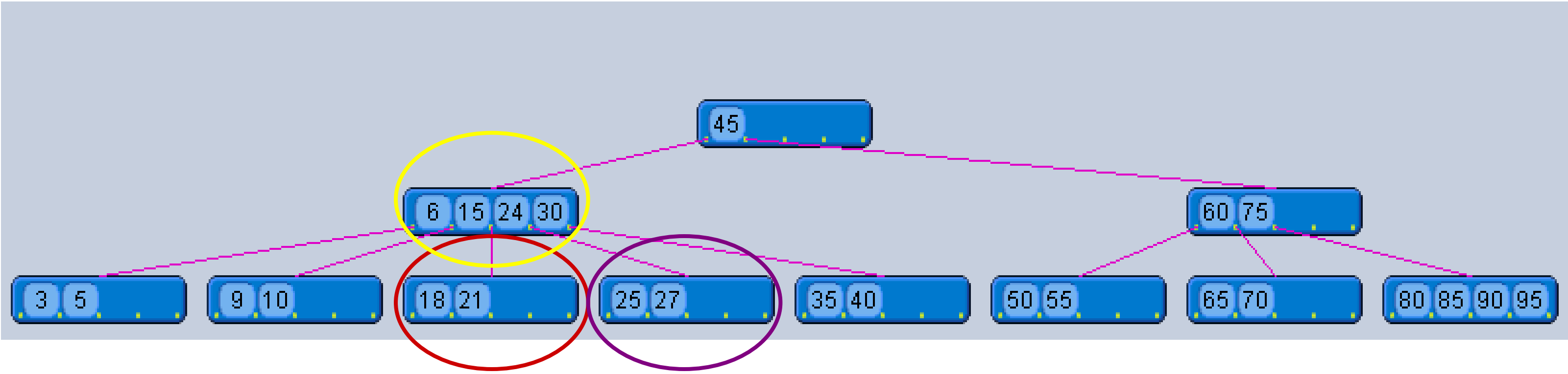


Regras

b. Se depois da retirada a folha tiver menos que t elementos, faça:
   - Procure se o irmão a direita ou a esquerda tenha mais que t elementos, se um deles tiver
     - a chave k do pai que separa os irmãos é incluída em X, e o irmão Y cede um elemento para o pai substituindo a chave emprestada

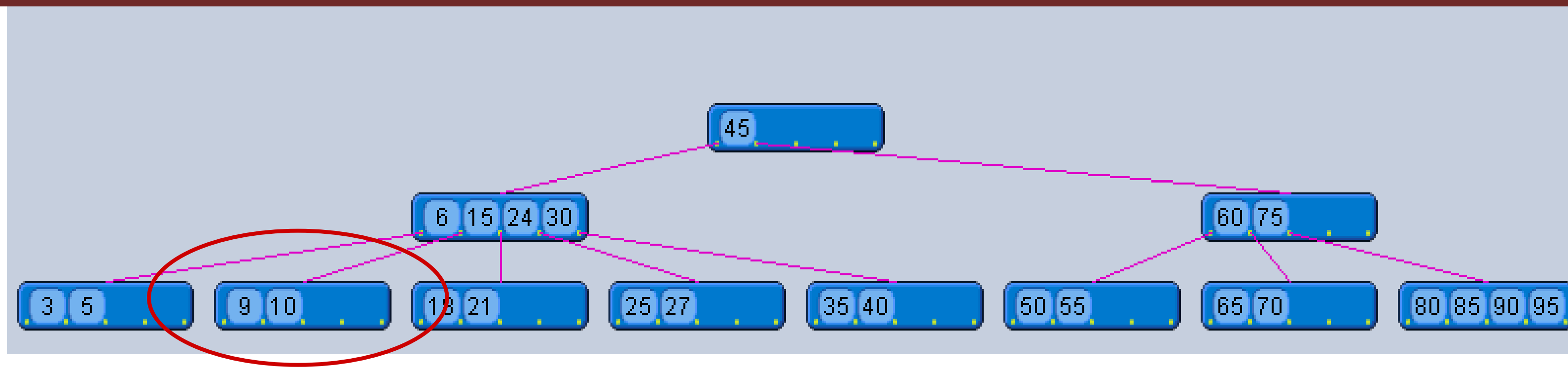


Removendo o valor
20.

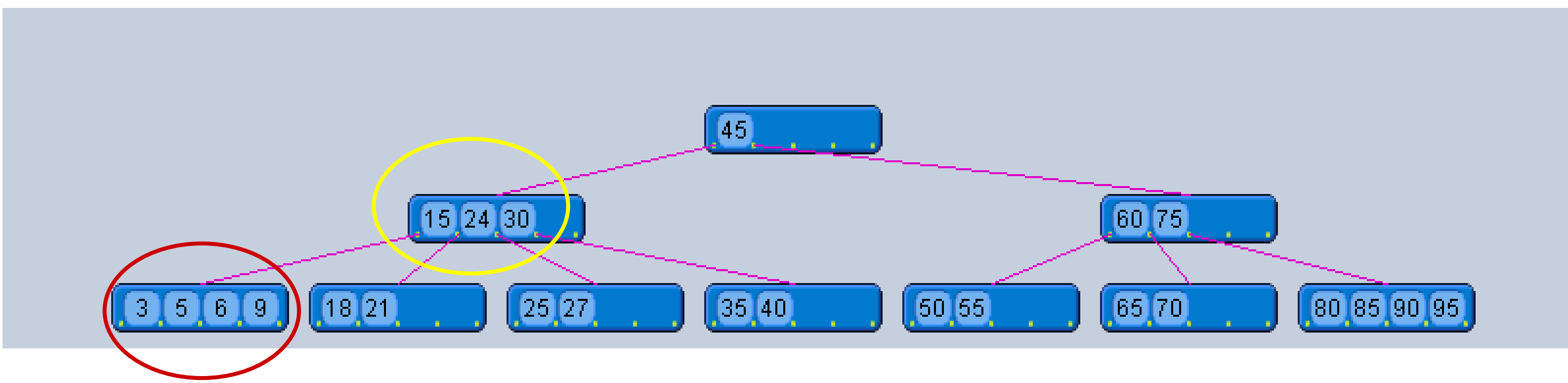


Regras

c. Se os dois irmãos de X tiverem exatamente t elementos:
   - Nenhum elemento poderá ser deslocado
   - O nó X e um de seus irmãos são concatenados
   - A chave separadora do pai também estará no nó conectado

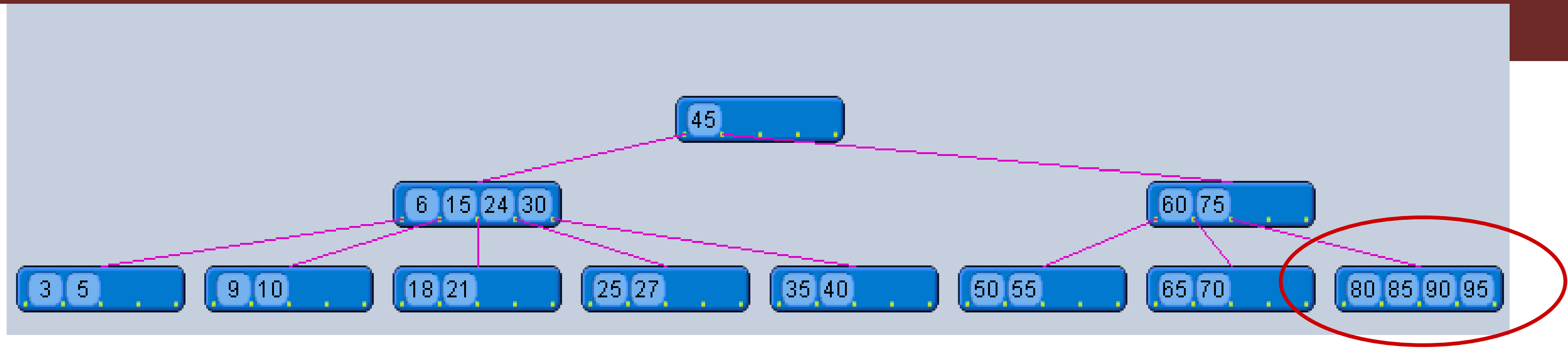


Removendo o valor
10.



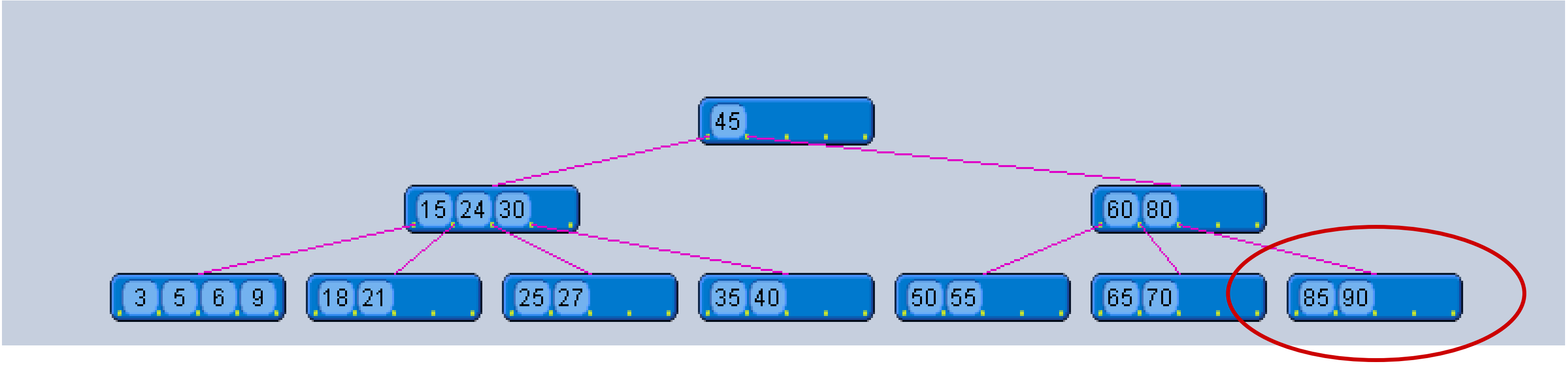
Regras

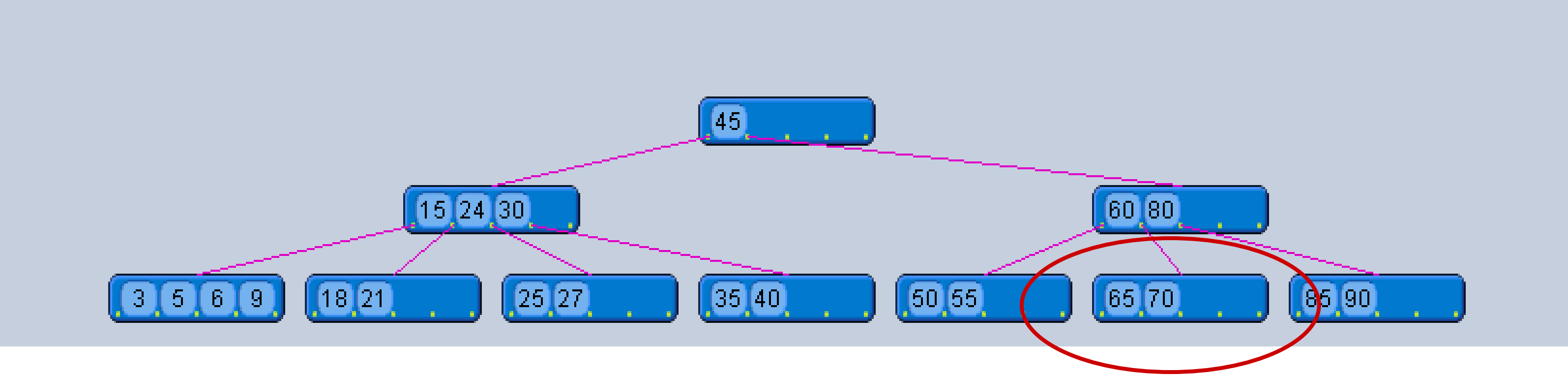
d. Se o pai tiver apenas t elementos, devemos considerar os irmãos do pai como no caso anterior e proceder recursivamente
   - Se a raiz da árvore contiver apenas 1 elemento a árvore sofrerá uma redução de altura



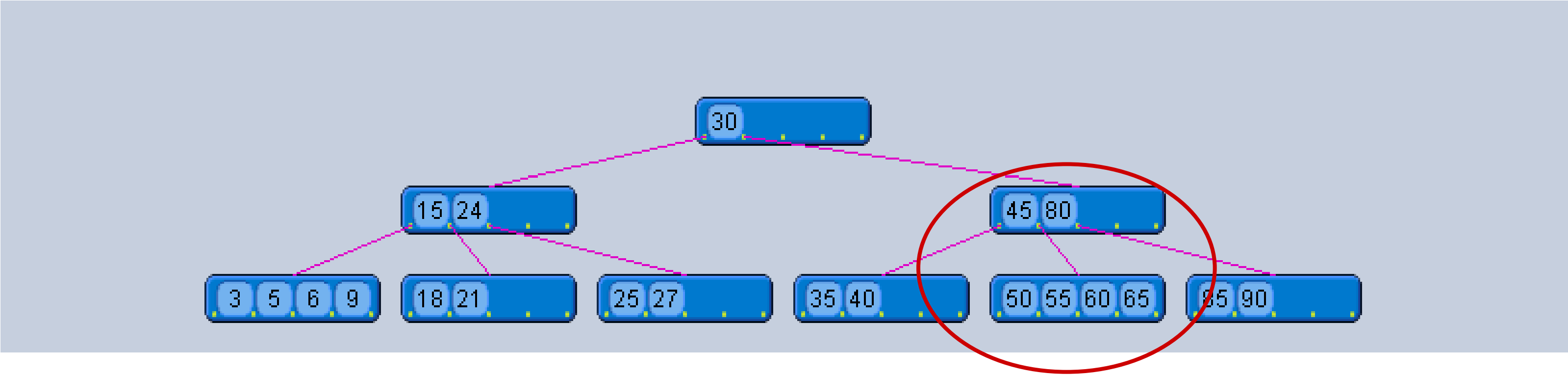
Removendo o valor

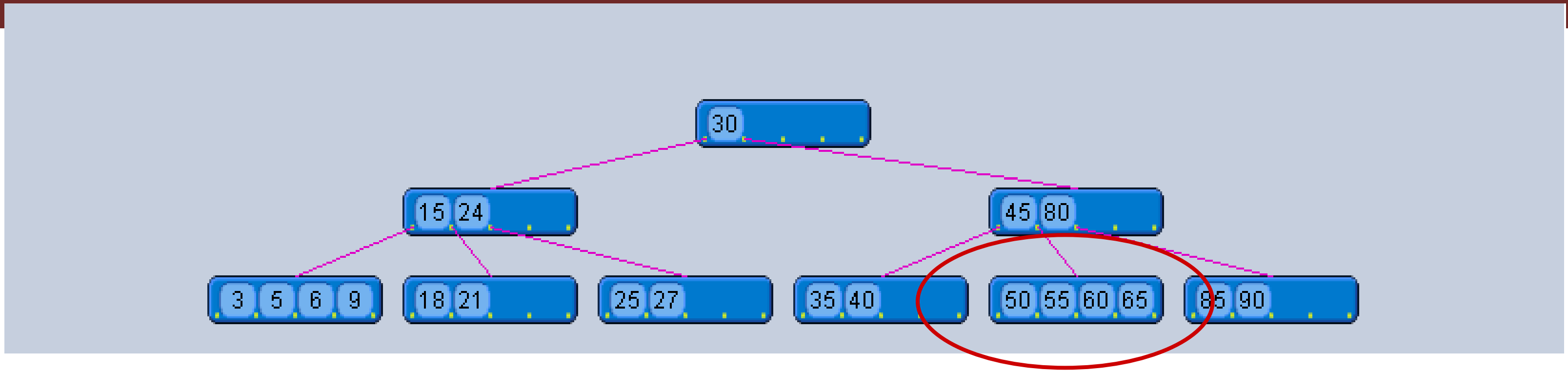
75 e 95



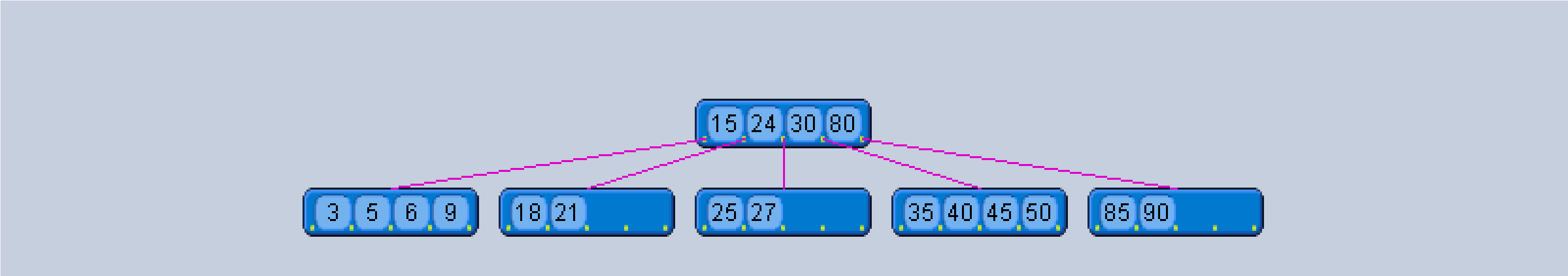


Removendo o valor
70



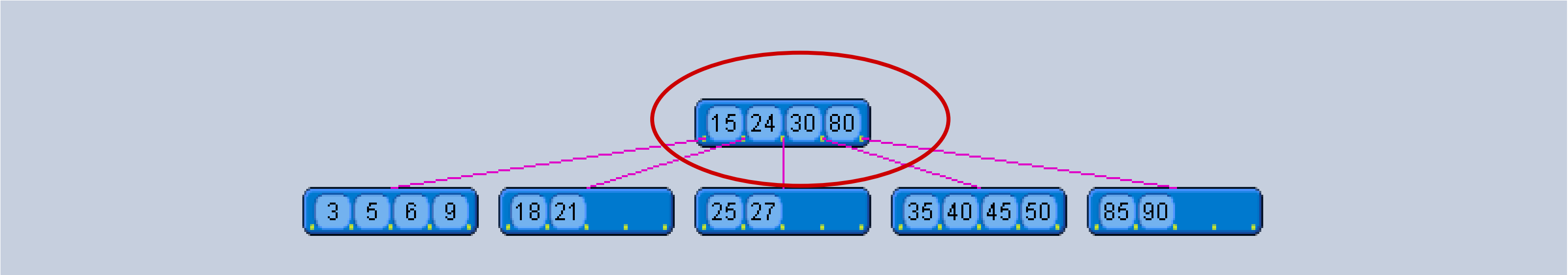


Removendo o valor
55,60 e 65

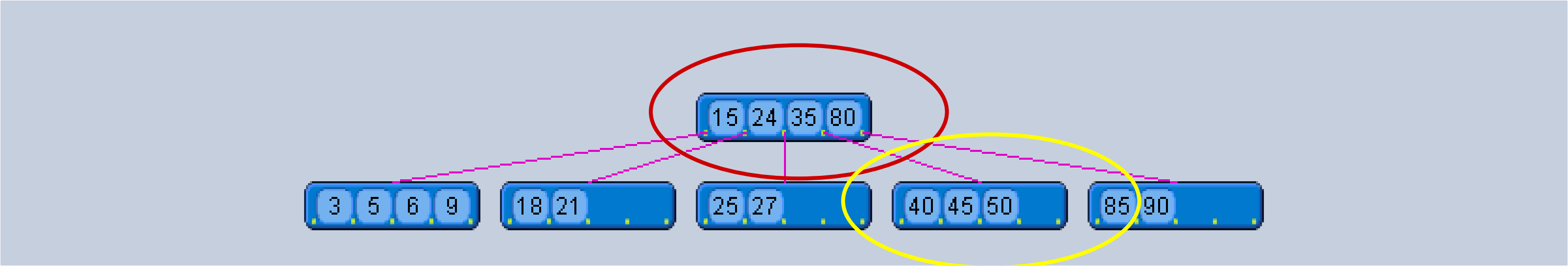


Regras

- **O item será retirado de um nó interno Z**
  - Será encontrado o seu sucessor (que está em uma folha) e será substituído pelo mesmo



Removendo o valor
30



Exercício

1) Desenhe uma árvore B de ordem 3 que contenha as seguintes chaves: 1, 3, 6, 8, 14, 32, 36, 38, 39, 41, 43.

2) Desenhe uma árvore B de ordem 2 que contenha as seguintes chaves: 2, 5, 7, 10, 13, 16, 18, 21.