MC-202 — Aula 15
Árvores-B

Lehilton Pedrosa

Instituto de Computação – Unicamp

Segundo Semestre de 2015

Roteiro

Introdução

Problema

Trabalhamos com 1.000.000 de registros! Como cada registro pode ser muito grande (uma foto, por exemplo), não podemos guardá-los todos na memória. Toda vez que executamos o programa, temos que executar cerca de 1000 consultas nesse banco de dados.

Introdução

Problema

Trabalhamos com 1.000.000 de registros! Como cada registro pode ser muito grande (uma foto, por exemplo), não podemos guardá-los todos na memória. Toda vez que executamos o programa, temos que executar cerca de 1000 consultas nesse banco de dados.

1. Onde armazenar os dados?

Introdução

Problema

Trabalhamos com 1.000.000 de registros! Como cada registro pode ser muito grande (uma foto, por exemplo), não podemos guardá-los todos na memória. Toda vez que executamos o programa, temos que executar cerca de 1000 consultas nesse banco de dados.

1. Onde armazenar os dados?
2. Qual estrutura de dados?

Introdução

Problema

Trabalhamos com 1.000.000 de registros! Como cada registro pode ser muito grande (uma foto, por exemplo), não podemos guardá-los todos na memória. Toda vez que executamos o programa, temos que executar cerca de 1000 consultas nesse banco de dados.

1. Onde armazenar os dados?
2. Qual estrutura de dados?

Tentativa: vamos usar uma árvore binária para diminuir o tempo de busca.

Entendendo um disco

Foto: wikipédia

Entendendo um disco

Foto: wikipédia

Características de um disco

É um instrumento mecânico: posicionar um cabeçote na posição de leitura

Entendendo um disco

Foto: wikipédia

Características de um disco

É um instrumento mecânico: posicionar um cabeçote na posição de leitura

- **latência:** demora até posicionar o cabeçote ($\approx$ 5ms)
- **página:** a quantidade de dados lida de uma só vez (em geral)

Entendendo um disco

Foto: wikipédia

Características de um disco

É um instrumento mecânico: posicionar um cabeçote na posição de leitura

- **latência:** demora até posicionar o cabeçote ($\approx$ 5ms)
- **página:** a quantidade de dados lida de uma só vez (em geral)
  uma vez posicionado, ler é rápido: +100 MB/s

Verificando nossa solução

Calculando o tempo

Quanto tempo nosso programa vai levar?

Verificando nossa solução

Calculando o tempo

Quanto tempo nosso programa vai levar?

- a árvore tem $1.000.000$ de nós
- a altura é $\log_2(1.000.000) \approx 20$

Verificando nossa solução

Calculando o tempo

Quanto tempo nosso programa vai levar?

- a árvore tem $1.000.000$ de nós
- a altura é $\log_2(1.000.000) \approx 20$

$$\text{TEMPO} = 1000 \text{ buscas} \times 20 \text{ nós/busca} \times 5 \text{ ms/nó} = 100s$$

Verificando nossa solução

Calculando o tempo

Quanto tempo nosso programa vai levar?

- a árvore tem $1.000.000$ de nós
- a altura é $\log_2(1.000.000) \approx 20$

$$\text{TEMPO} = 1000 \text{ buscas} \times 20 \text{ nós/busca} \times 5 \text{ ms/nó} = 100s$$

E se fosse na memória?

- tempo de acesso é **muito menor**: $<$ 20ns
- tempo total: $<$ 1ms

Verificando nossa solução

Calculando o tempo

Quanto tempo nosso programa vai levar?

- a árvore tem $1.000.000$ de nós
- a altura é $\log_2(1.000.000) \approx 20$

$$\text{TEMPO} = 1000 \text{ buscas} \times 20 \text{ nós/busca} \times 5 \text{ ms/nó} = 100s$$

E se fosse na memória?

- tempo de acesso é **muito menor**: < 20ns
- tempo total: < 1ms

Solução

Usar uma árvore k-ária: $\log_k n << \log_2 n$.

Top-Down *vs.* Bottom-Up

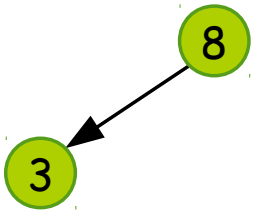
Criando uma árvore:

Top-Down:

Top-Down *vs.* Bottom-Up

Criando uma árvore:

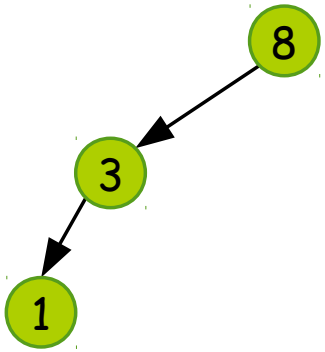
Top-Down:



Top-Down *vs.* Bottom-Up

Criando uma árvore:

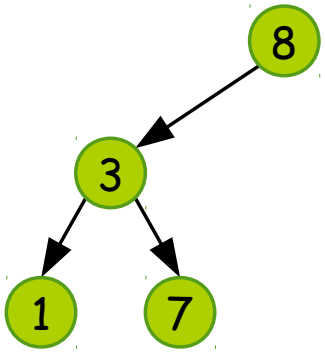
Top-Down:



Top-Down *vs.* Bottom-Up

Criando uma árvore:

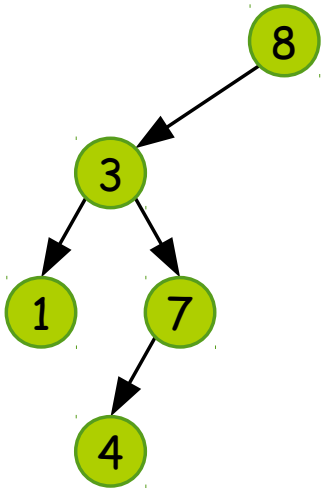
Top-Down:



Top-Down *vs.* Bottom-Up

Criando uma árvore:

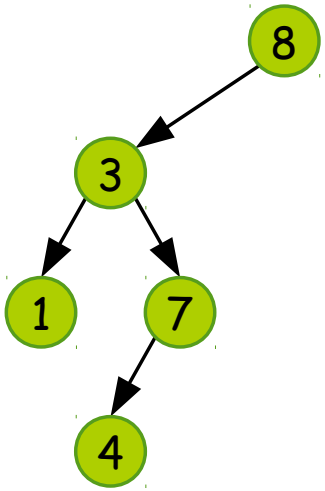
Top-Down:



Top-Down *vs.* Bottom-Up

Criando uma árvore:

Top-Down:

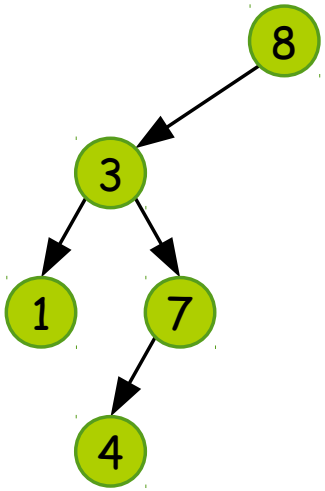


Bottom-Up:

Top-Down *vs.* Bottom-Up

Criando uma árvore:

Top-Down:



Bottom-Up:

Vamos ver nessa aula!

Árvore-B

Árvore-B

- introduzida por Rudolf Bayer e Ed McCreight (1971)
- ideia: toda folha tem a mesma altura (usamos inserção bottom-up)

Árvore-B

Árvore-B

- introduzida por Rudolf Bayer e Ed McCreight (1971)
- ideia: toda folha tem a mesma altura (usamos inserção bottom-up)

Definição

Uma **Árvore-B** de ordem $b \geq 2$ é uma árvore que:

1. todas as folhas tem o mesmo nível
2. se um nó tem r registros, então:
   - r vale no máximo b
   - com exceção da raiz, r vale no mínimo $\lfloor b/2 \rfloor$
   - o nó tem $r + 1$ subárvores
3. a árvore satisfaz a propriedade de busca

Árvore-B

Árvore-B

- introduzida por Rudolf Bayer e Ed McCreight (1971)
- ideia: toda folha tem a mesma altura (usamos inserção bottom-up)

Definição

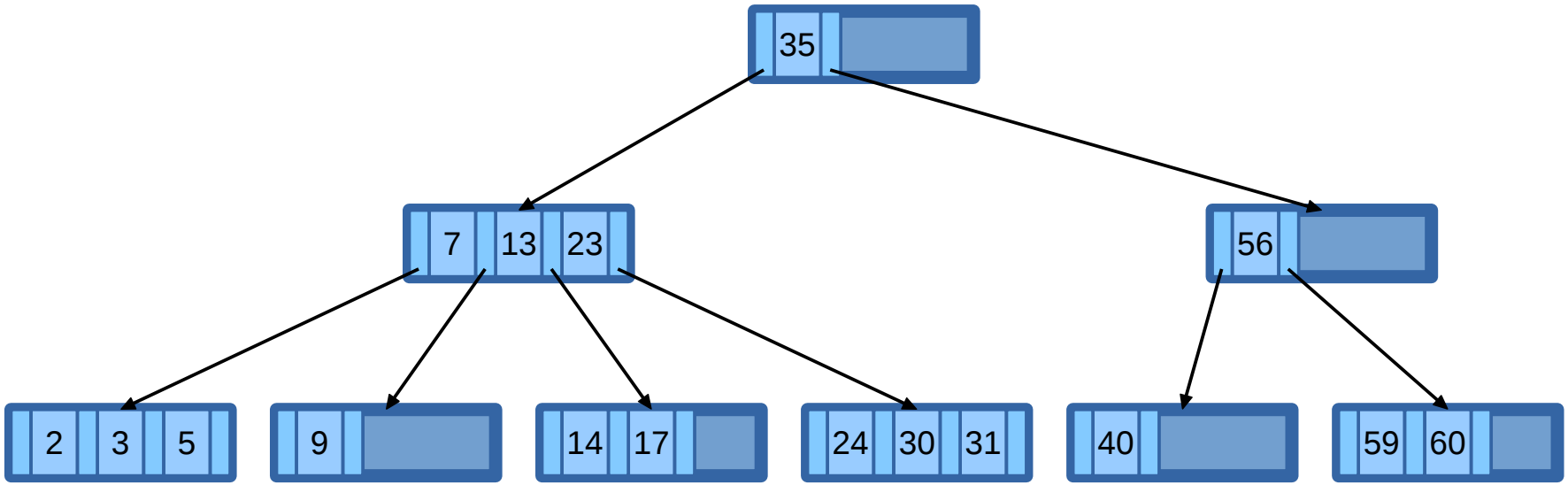
Uma **Árvore-B** de ordem $b \geq 2$ é uma árvore que:

1. todas as folhas tem o mesmo nível
2. se um nó tem r registros, então:
   - r vale no máximo b
   - com exceção da raiz, r vale no mínimo $\lfloor b/2 \rfloor$
   - o nó tem $r + 1$ subárvores
3. a árvore satisfaz a propriedade de busca

Característica

- **Altura máxima:** $\log_{\lfloor b/2 \rfloor}(n+1)/2$

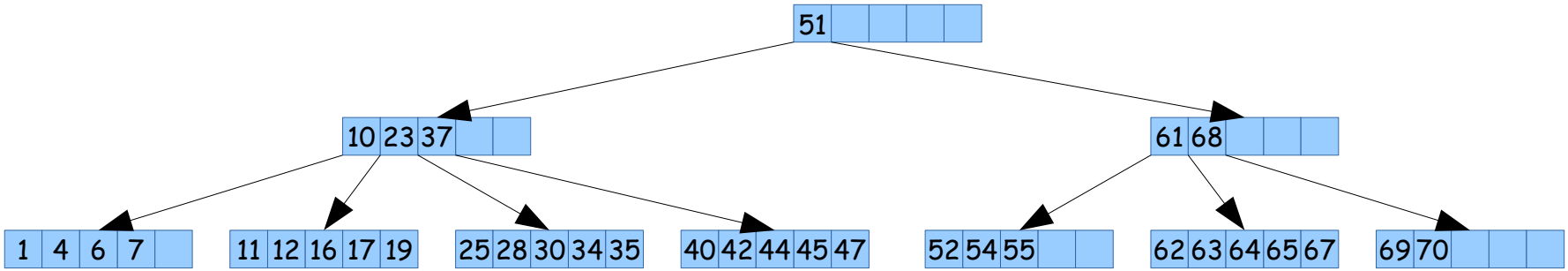
Exemplo



Ordem 3

- cada nó tem pelo menos 1 registro
- cada nó tem no máximo 3 registros

Outro exemplo



Ordem 5

- cada nó (com exceção da raiz) tem pelo menos 2 registros
- cada nó tem no máximo 5 registros

Especificação do nó

Nó de Árvore-B

```
#define ORDEM 255
```

Especificação do nó

Nó de Árvore-B

```
#define ORDEM 255

typedef struct NoArvB {
    int tam;
```

Especificação do nó

Nó de Árvore-B

```
#define ORDEM 255

typedef struct NoArvB {
    int tam;
    int chaves[ORDEM];
    struct NoArvB *filhos[ORDEM+1];
} *NoArvB;
```

Especificação do nó

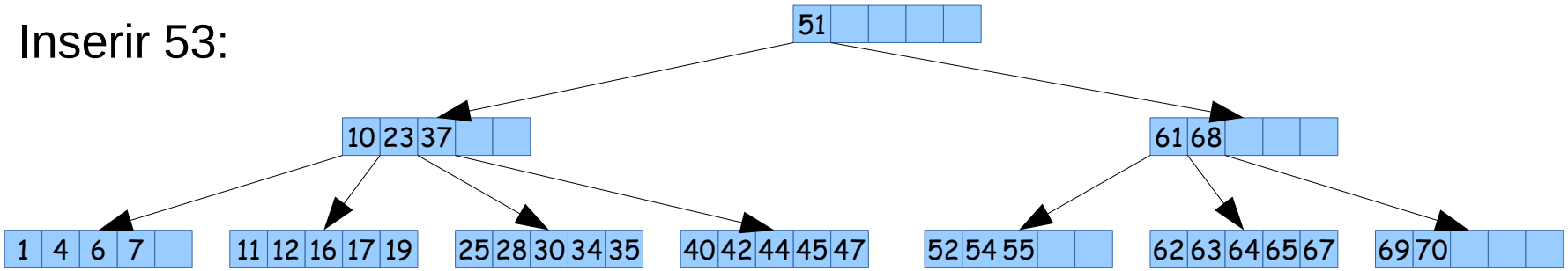
Nó de Árvore-B

```
#define ORDEM 255

typedef struct NoArvB {
    int tam;
    int chaves[ORDEM];
    struct NoArvB *filhos[ORDEM+1];
} *NoArvB;
```

Normalmente a ordem é ajustada para o máximo de elementos que cabem em um página de disco!

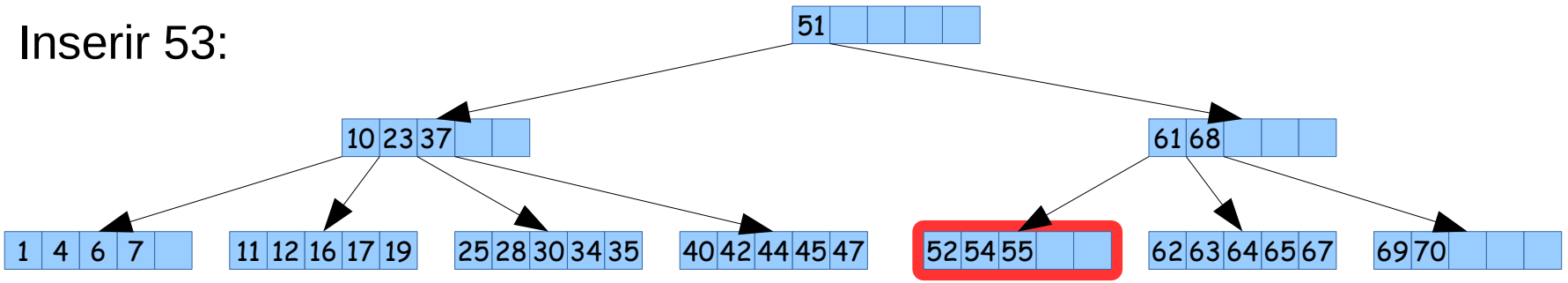
Inserindo em nó com espaço

Inserir 53:



Procedimento

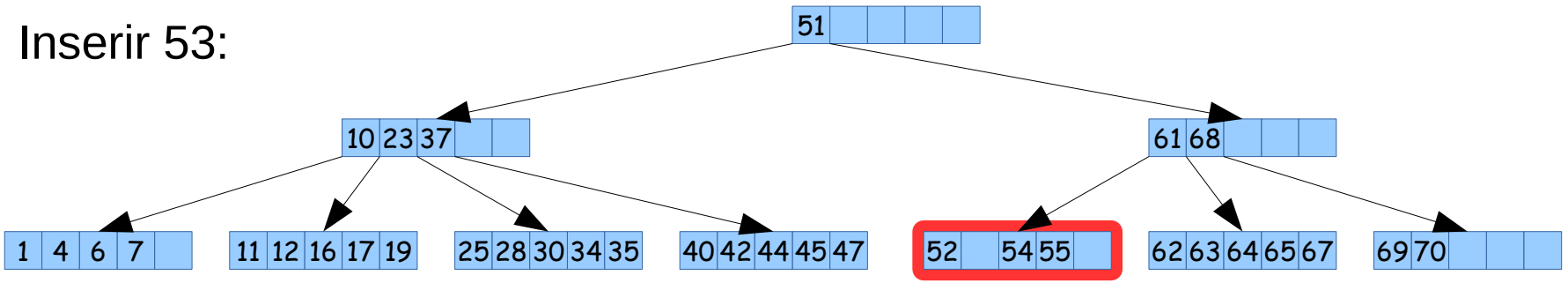
Inserindo em nó com espaço



Procedimento

1. procuramos folha

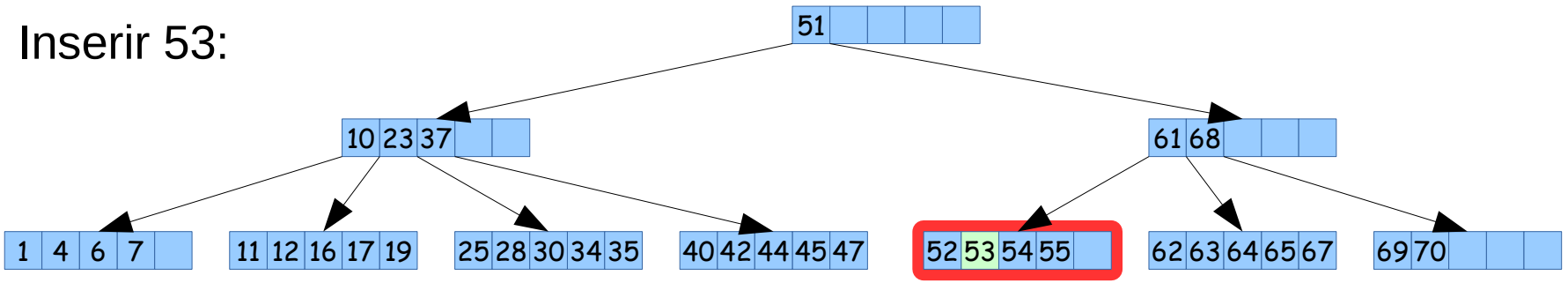
Inserindo em nó com espaço



Procedimento

1. procuramos folha
2. realocamos maiores

Inserindo em nó com espaço

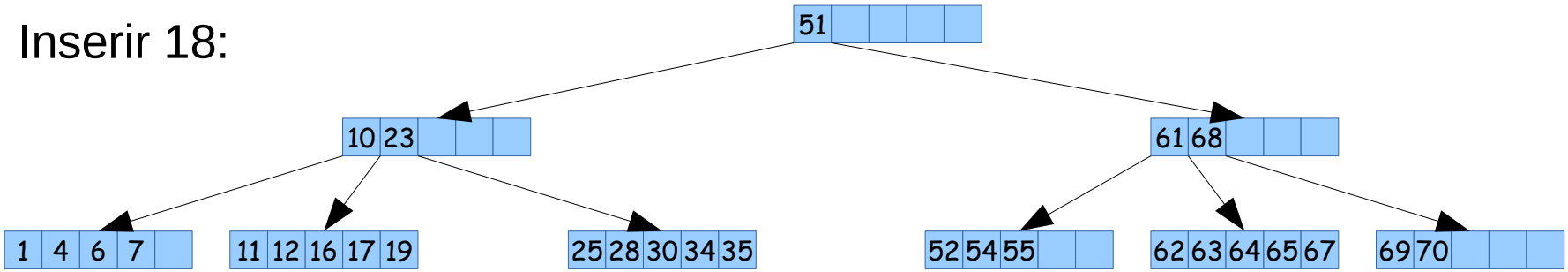


Procedimento

1. procuramos folha
2. realocamos maiores
3. inserimos elemento

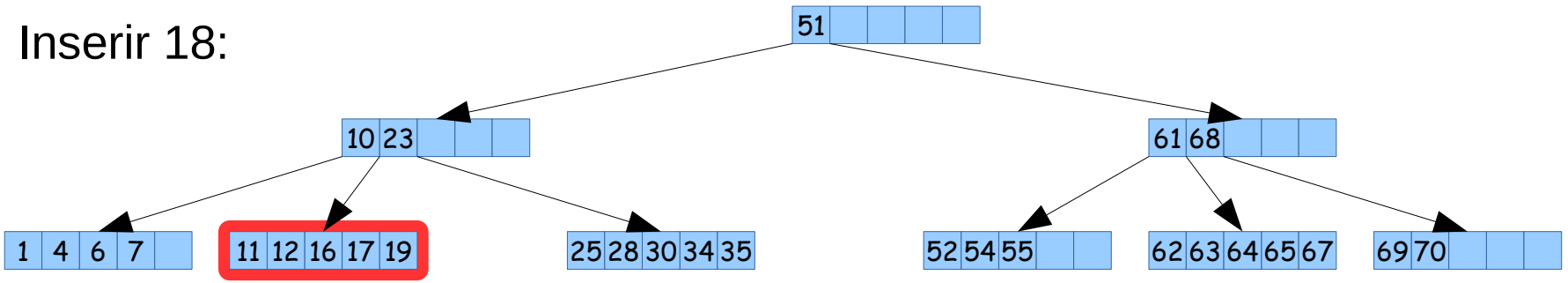
Inserindo em nó cheio

Inserir 18:



Procedimento

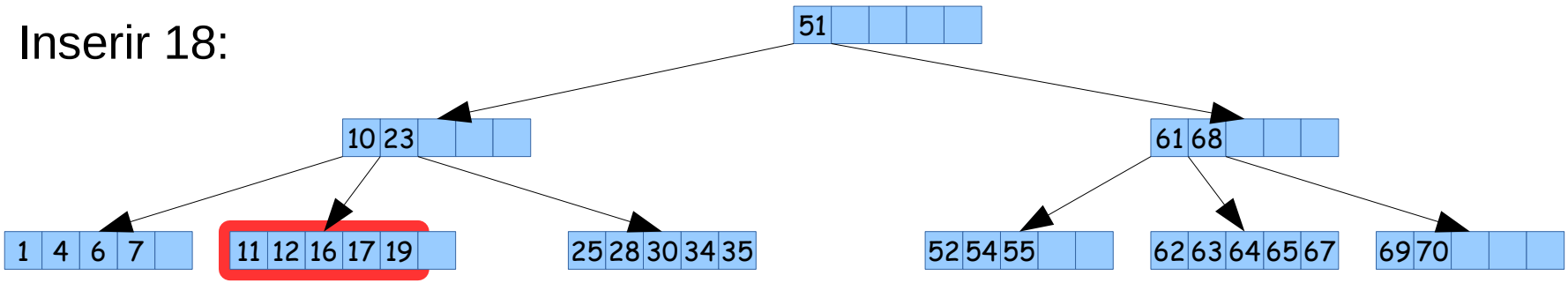
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos

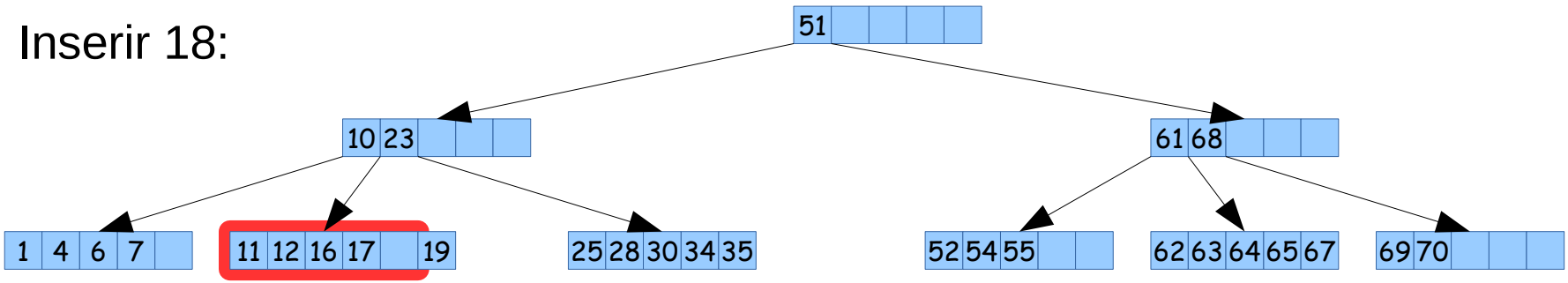
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)

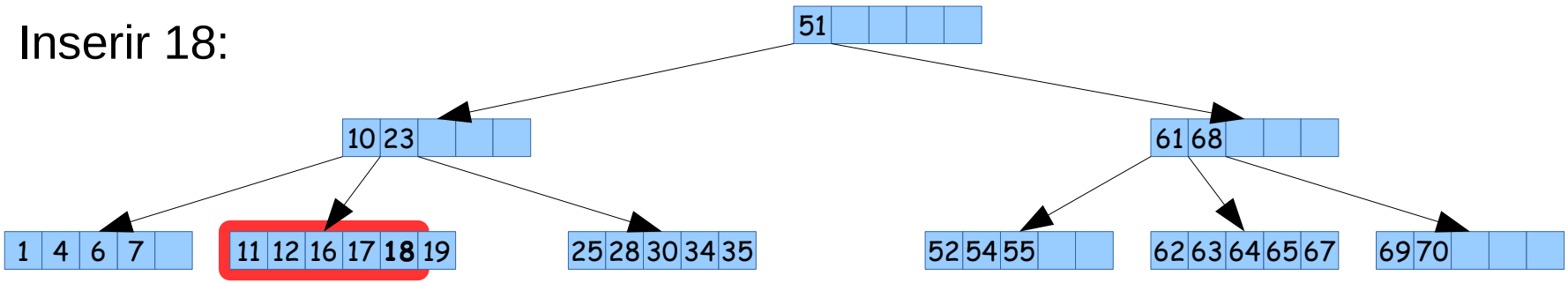
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)

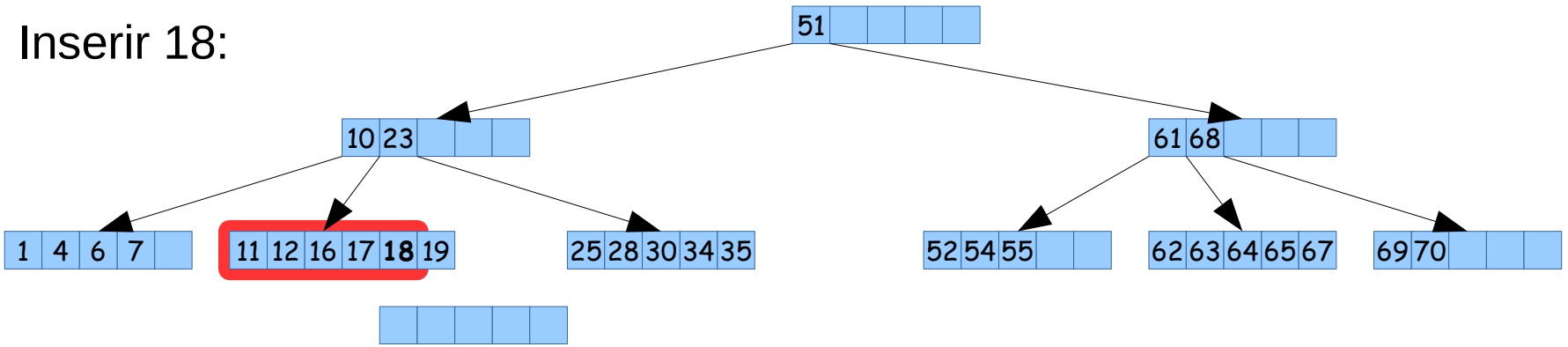
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)

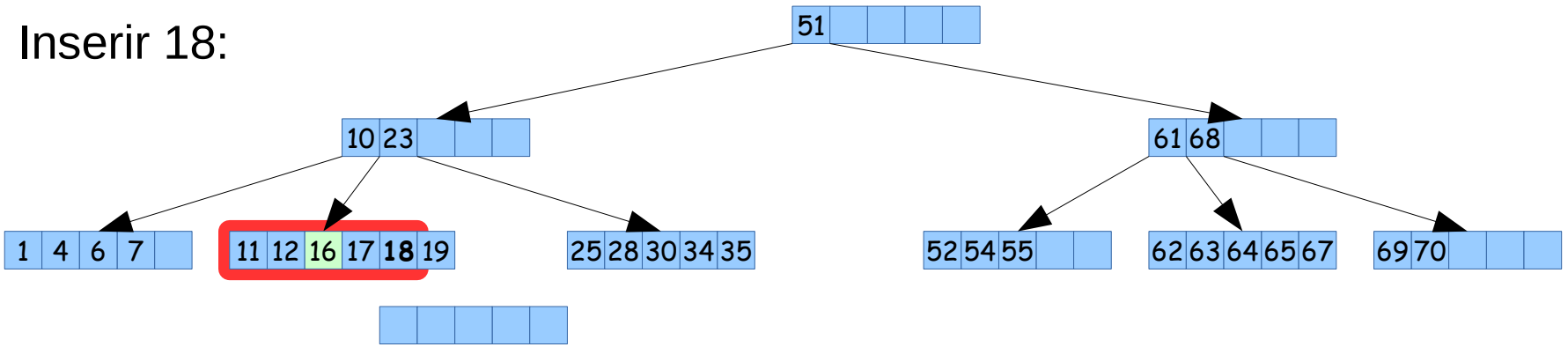
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó

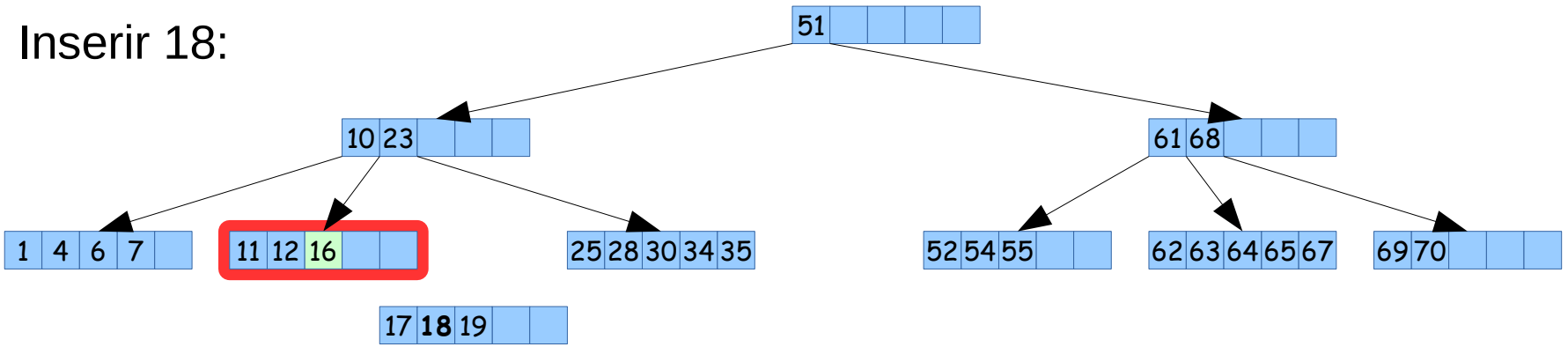
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó

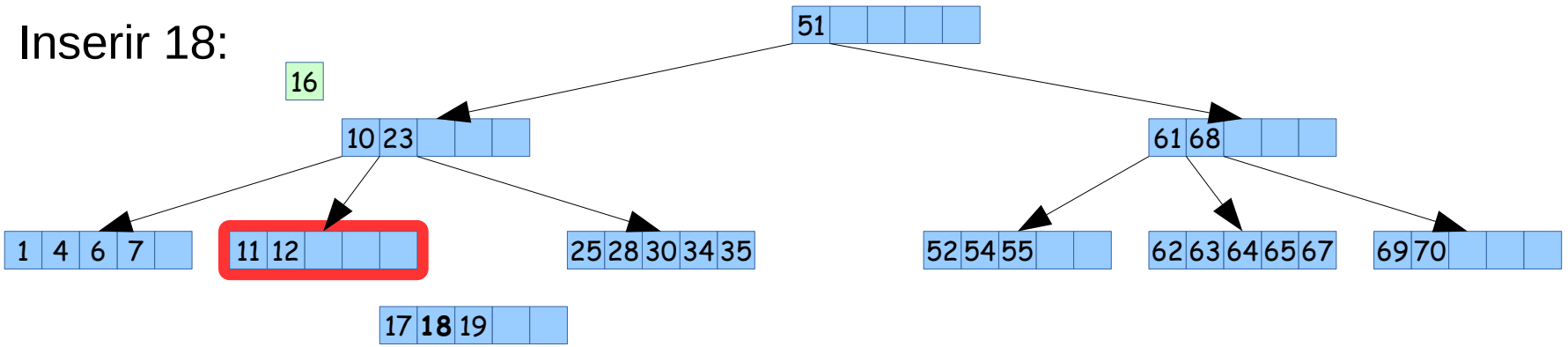
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos

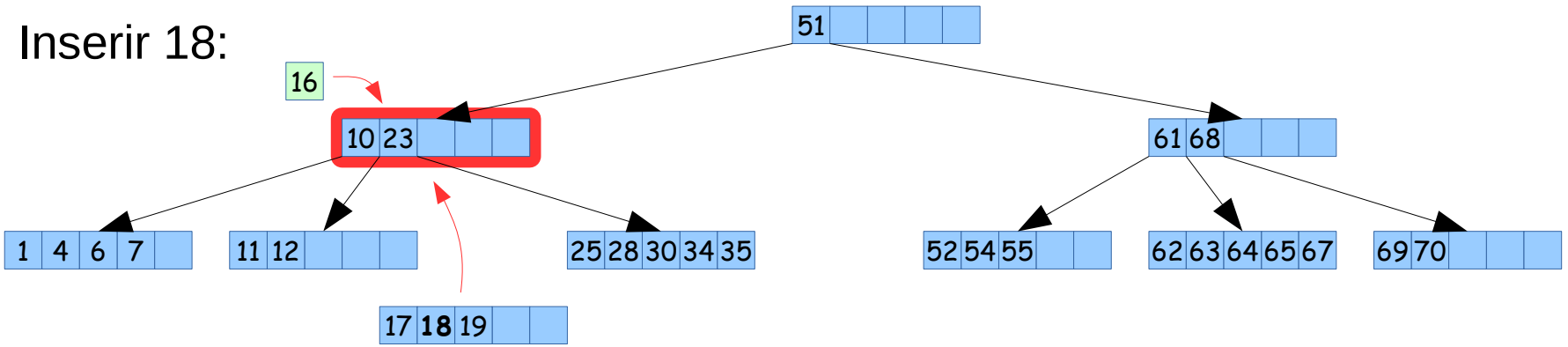
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos
3. removemos elemento do meio

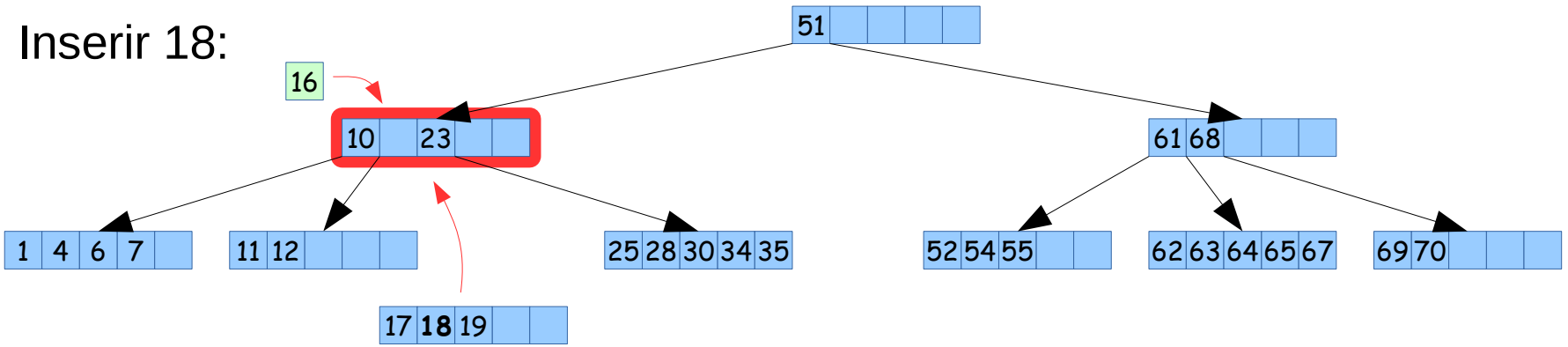
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos
3. removemos elemento do meio e propagamos inserção

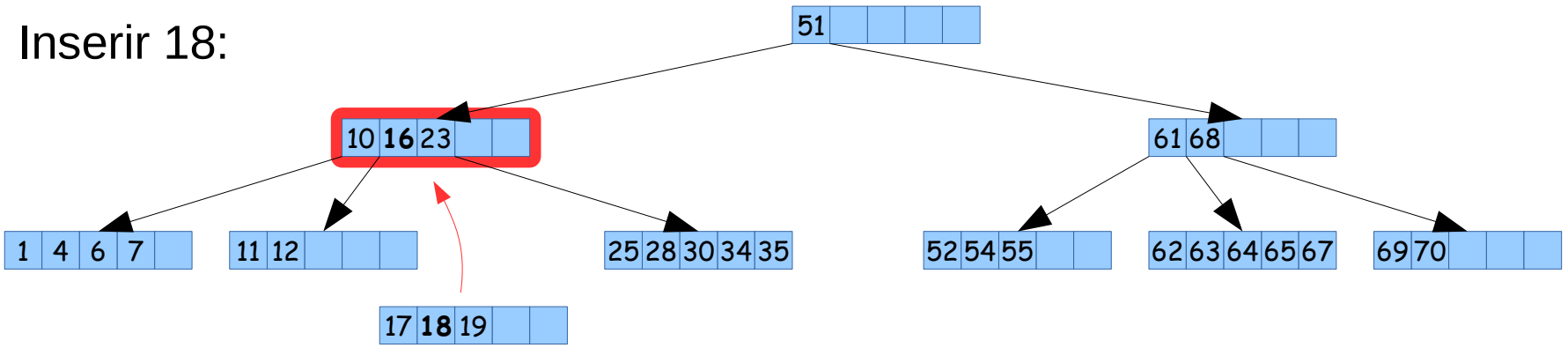
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos
3. removemos elemento do meio e propagamos inserção

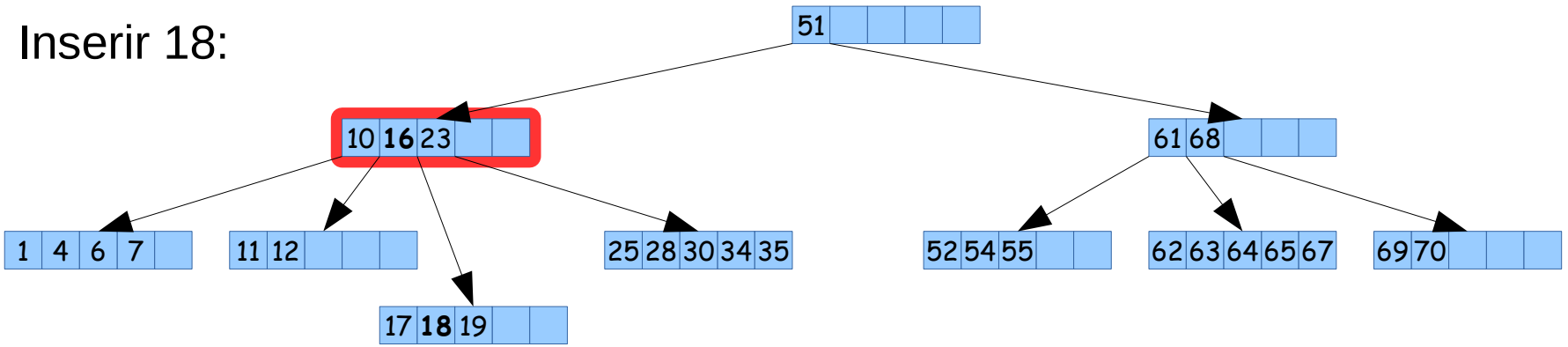
Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos
3. removemos elemento do meio e propagamos inserção

Inserindo em nó cheio



Procedimento

1. procuramos folha e inserimos (com *overflow*)
2. dividimos o nó em 2: criamos novo nó e copiamos
3. removemos elemento do meio e propagamos inserção

Inserção: visão geral

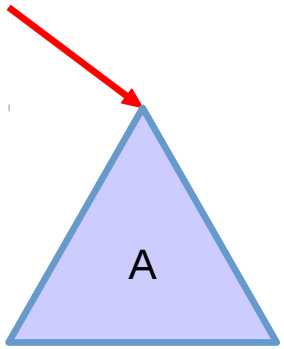
Ideia

Inserimos recursivamente e propagamos a inserção até a raiz

Inserção: visão geral

Ideia

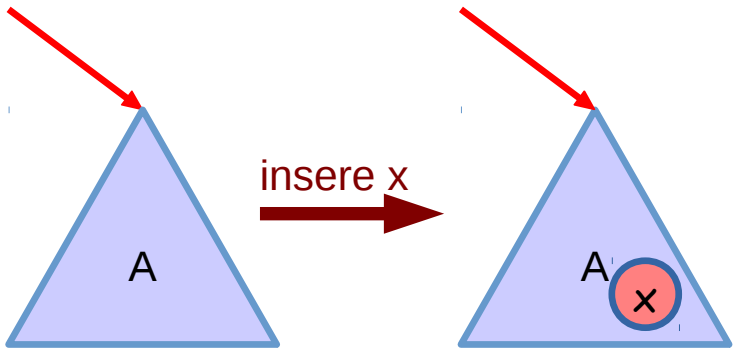
Inserimos recursivamente e propagamos a inserção até a raiz



Inserção: visão geral

Ideia

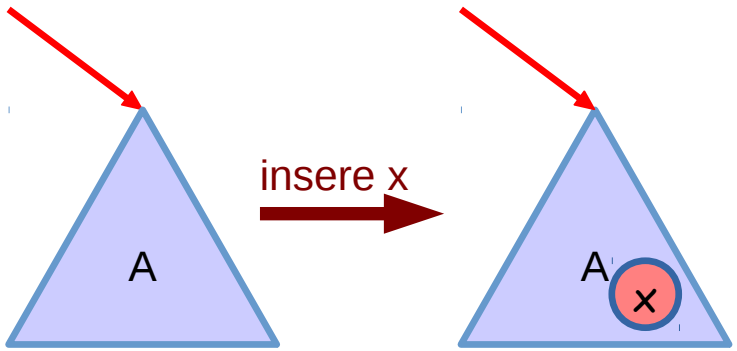
Inserimos recursivamente e propagamos a inserção até a raiz



Inserção: visão geral

Ideia

Inserimos recursivamente e propagamos a inserção até a raiz

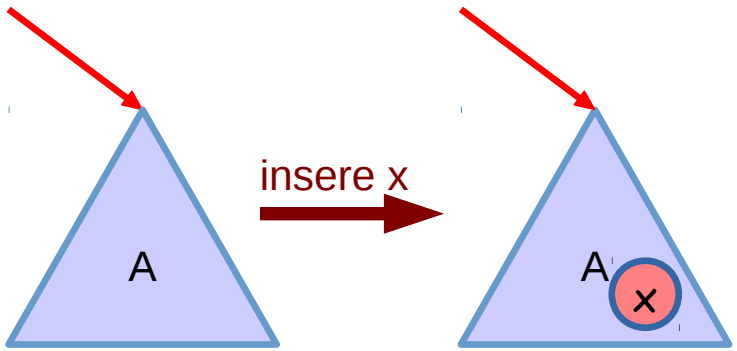


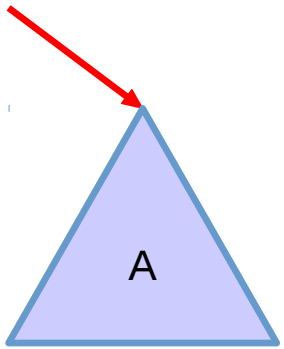
Não dividiu!

Inserção: visão geral

Ideia

Inserimos recursivamente e propagamos a inserção até a raiz



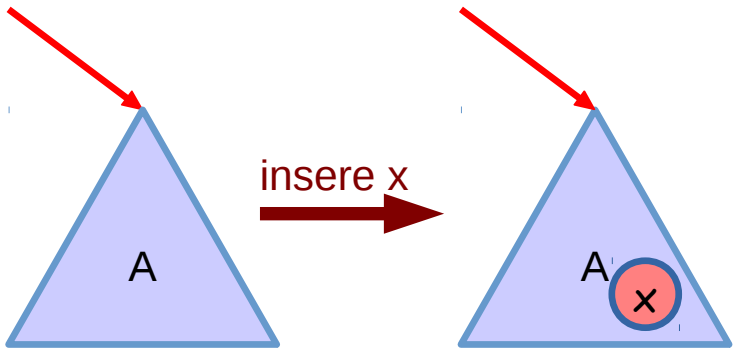


Não dividiu!

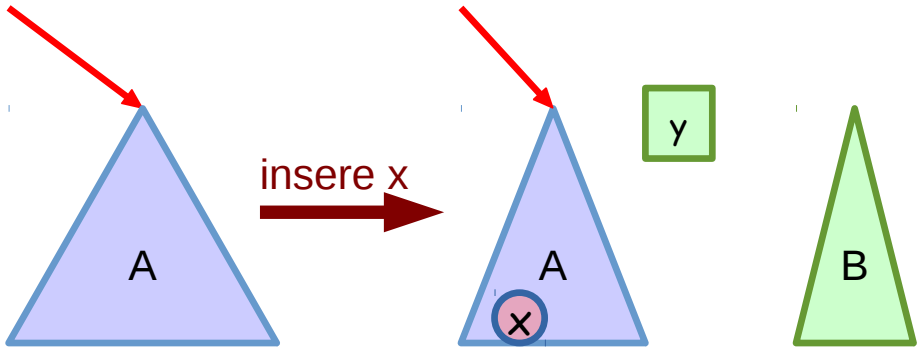
Inserção: visão geral

Ideia

Inserimos recursivamente e propagamos a inserção até a raiz



Não dividiu!

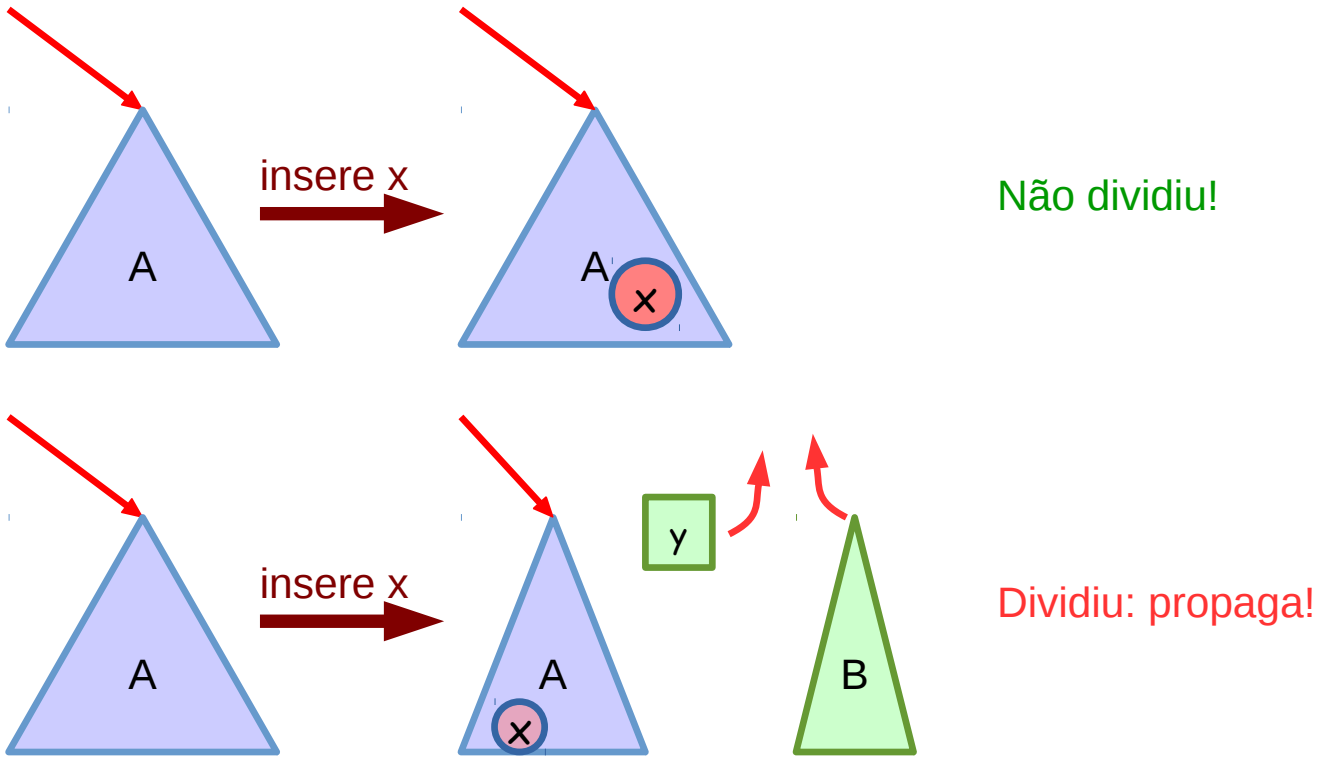


Dividiu:

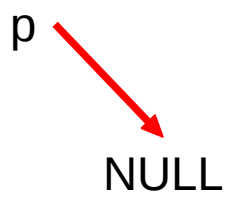
Inserção: visão geral

Ideia

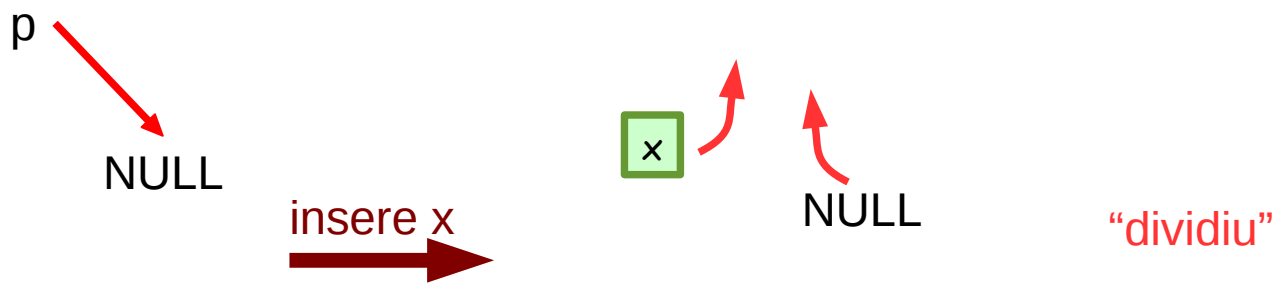
Inserimos recursivamente e propagamos a inserção até a raiz



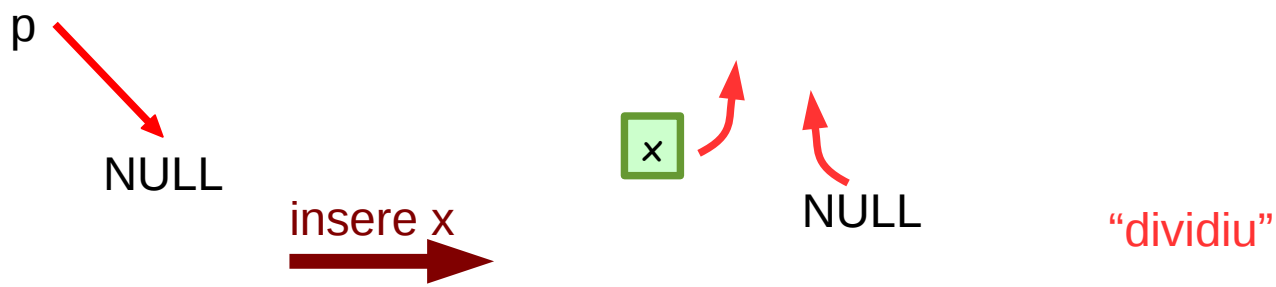
Caso 1: subárvore vazia



Caso 1: subárvore vazia

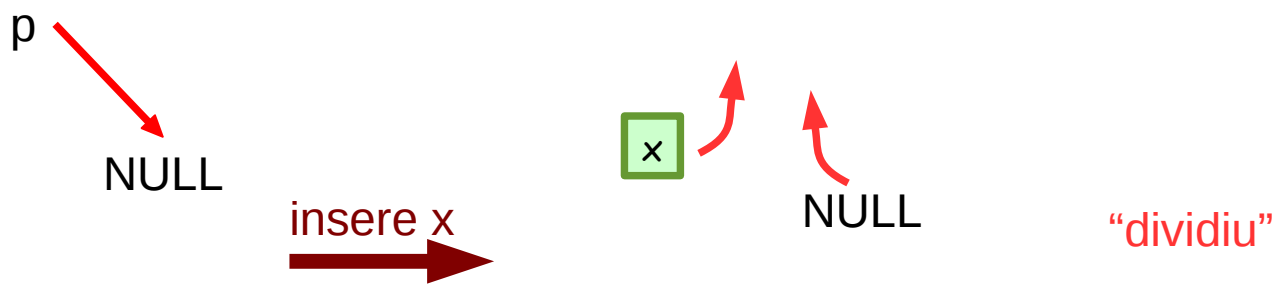


Caso 1: subárvore vazia



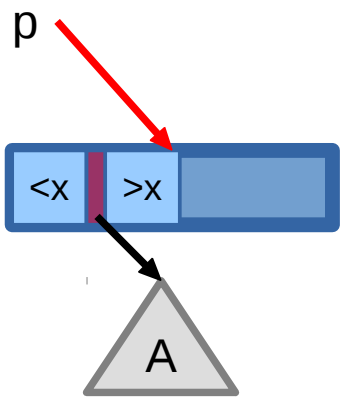
- **dividiu:** sim

Caso 1: subárvore vazia

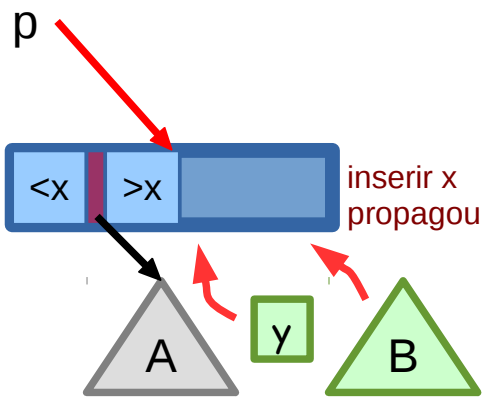


- **dividiu:** sim
- **procedimento:** a árvore propagada é vazia

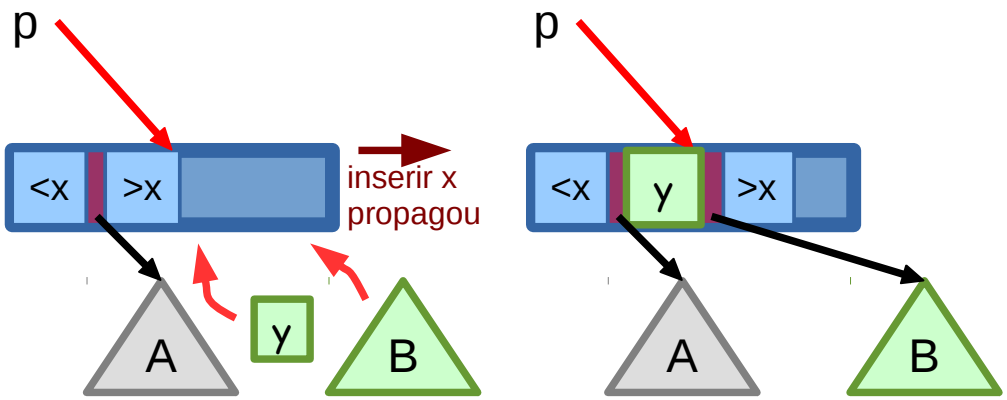
Caso 2: nó tem espaço livre



Caso 2: nó tem espaço livre

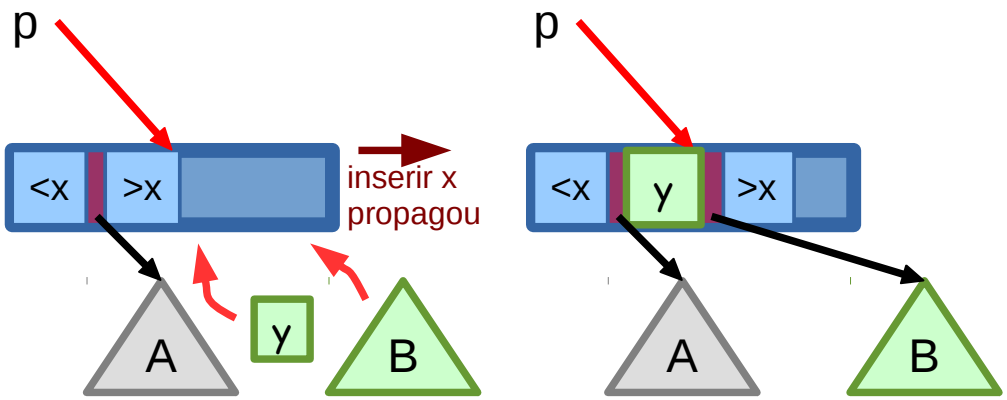


Caso 2: nó tem espaço livre



não dividiu

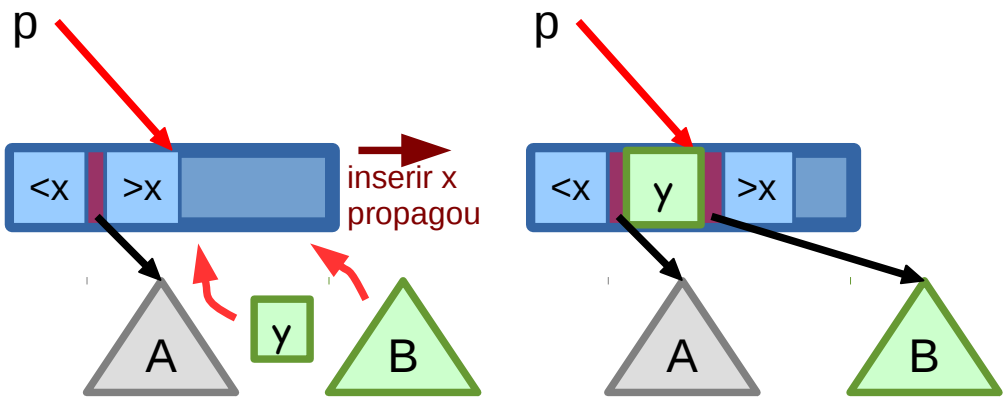
Caso 2: nó tem espaço livre



não dividiu

- **dividiu:** nao

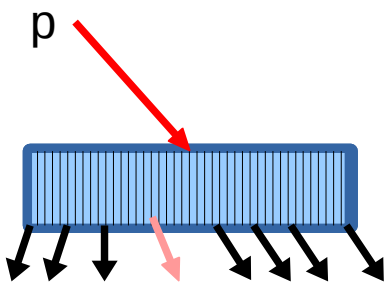
Caso 2: nó tem espaço livre



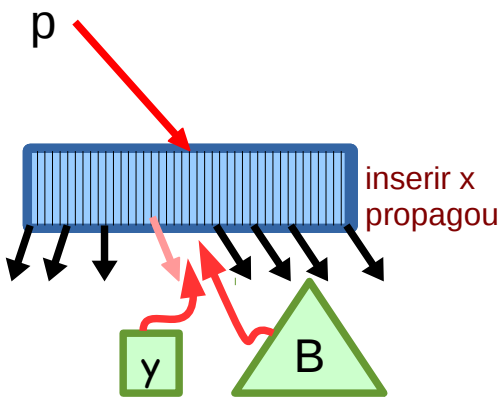
não dividiu

- **dividiu:** nao
- **procedimento:** desloca chaves/ponteiros e insere chave/ponteiro propagado

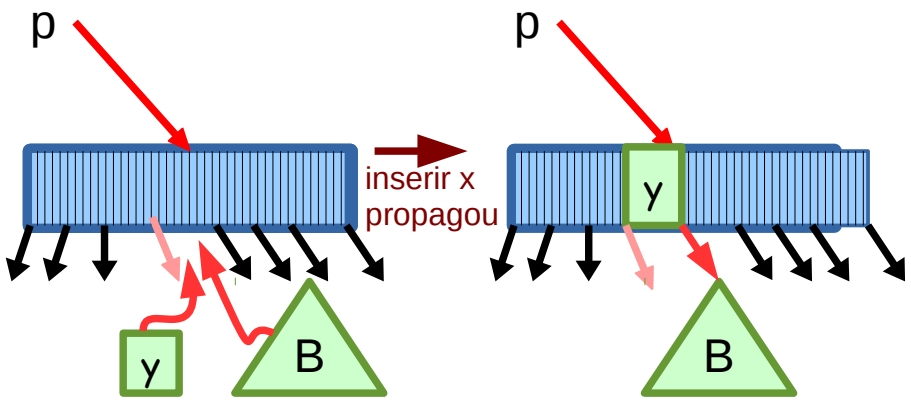
Caso 3: nó cheio



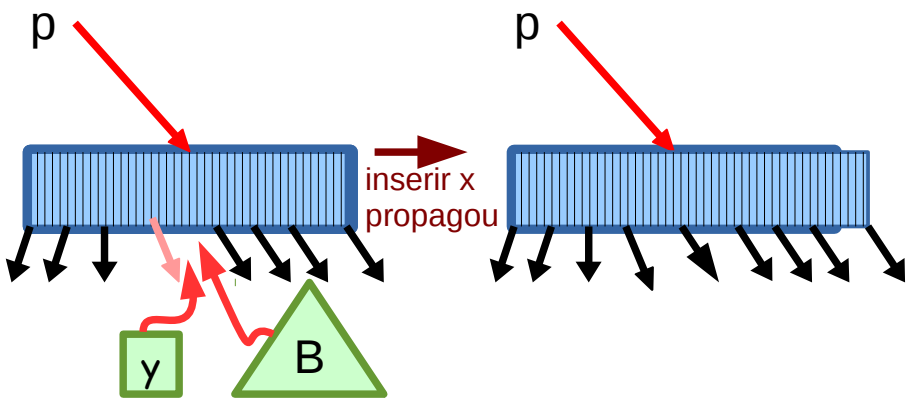
Caso 3: nó cheio



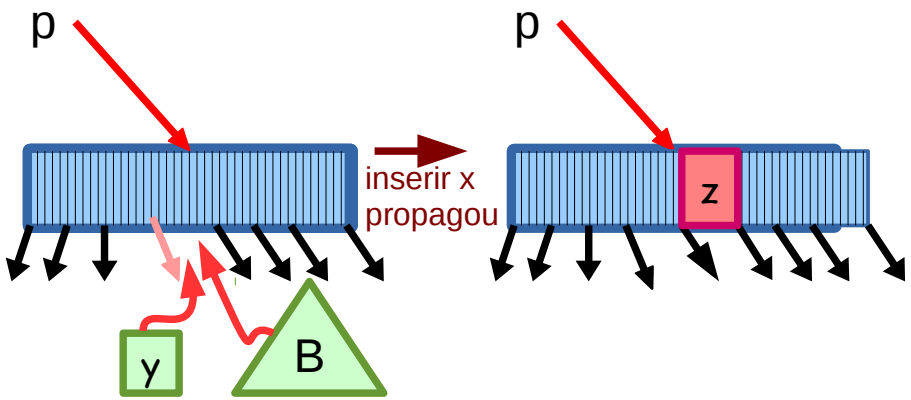
Caso 3: nó cheio



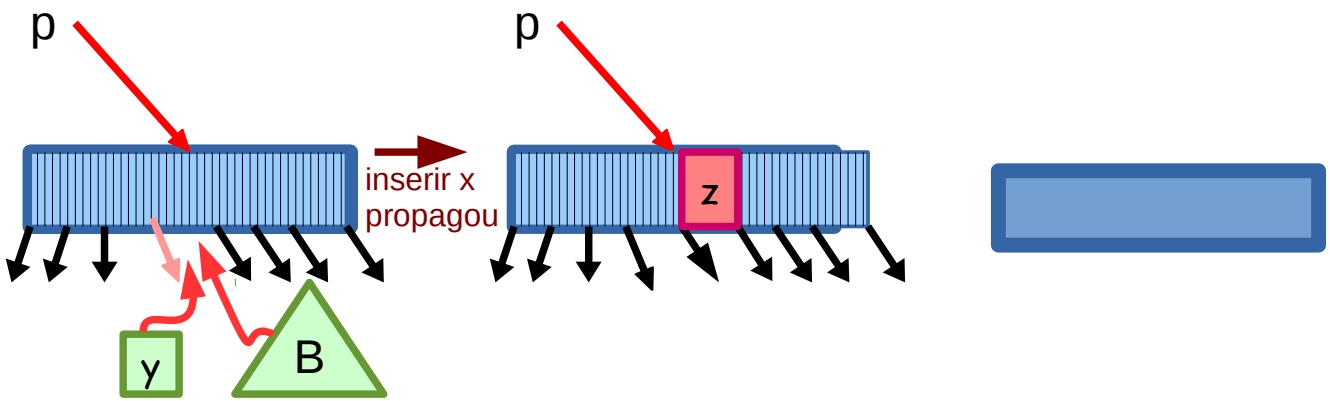
Caso 3: nó cheio



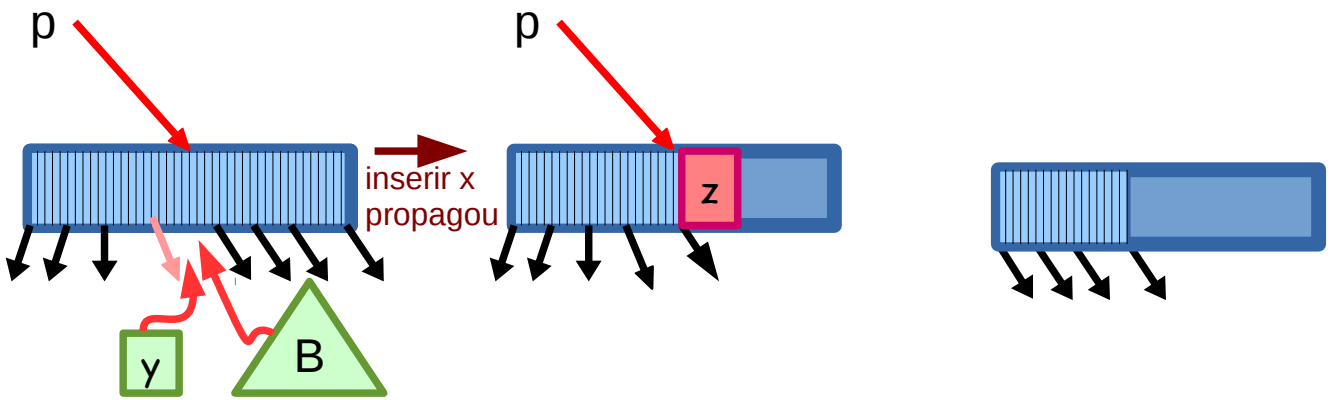
Caso 3: nó cheio



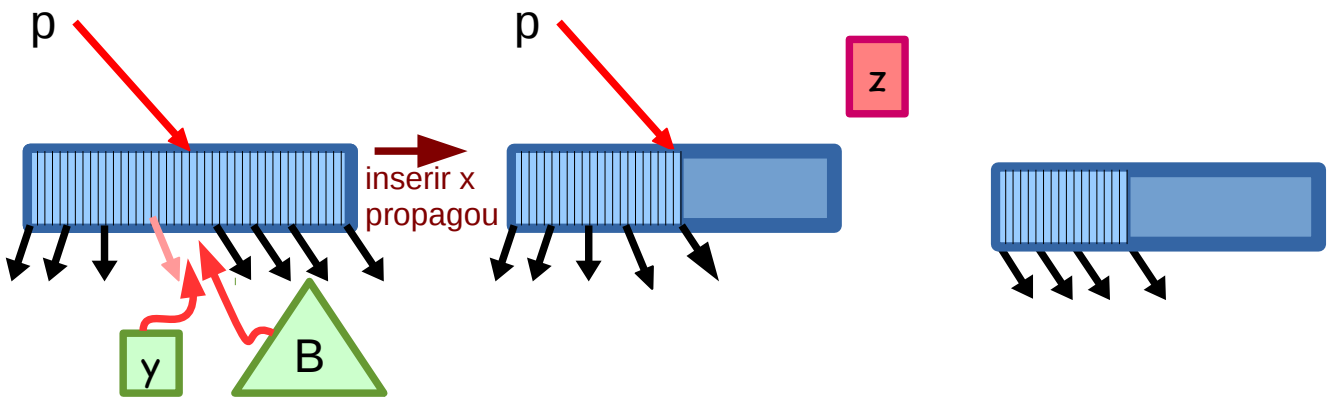
Caso 3: nó cheio



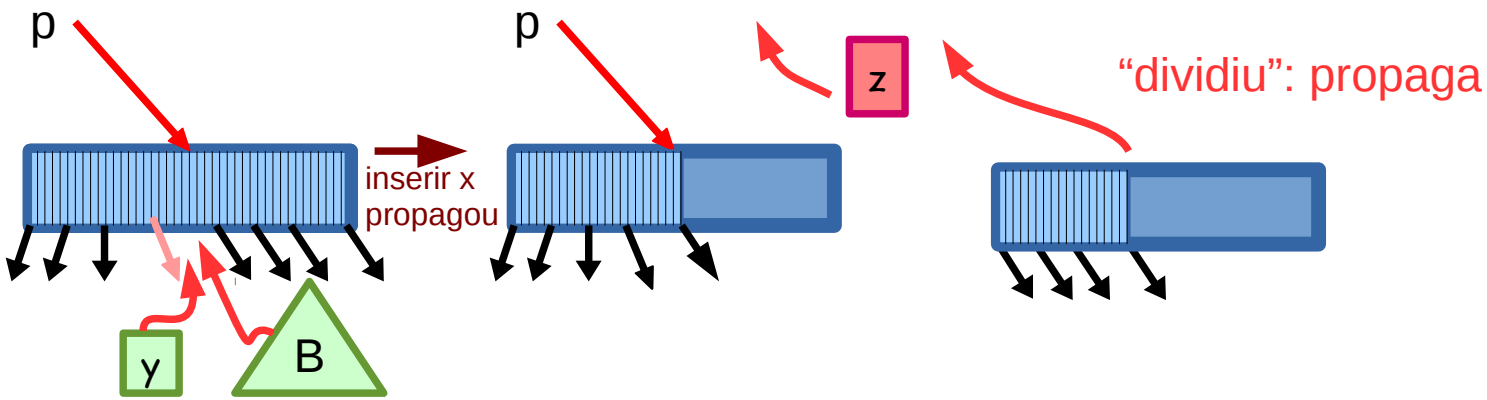
Caso 3: nó cheio



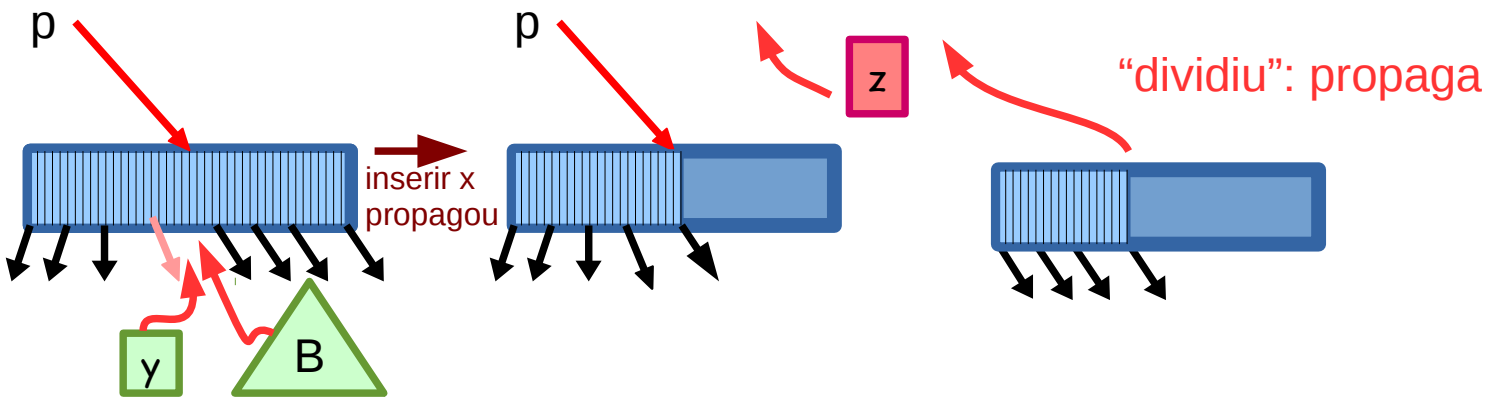
Caso 3: nó cheio



Caso 3: nó cheio

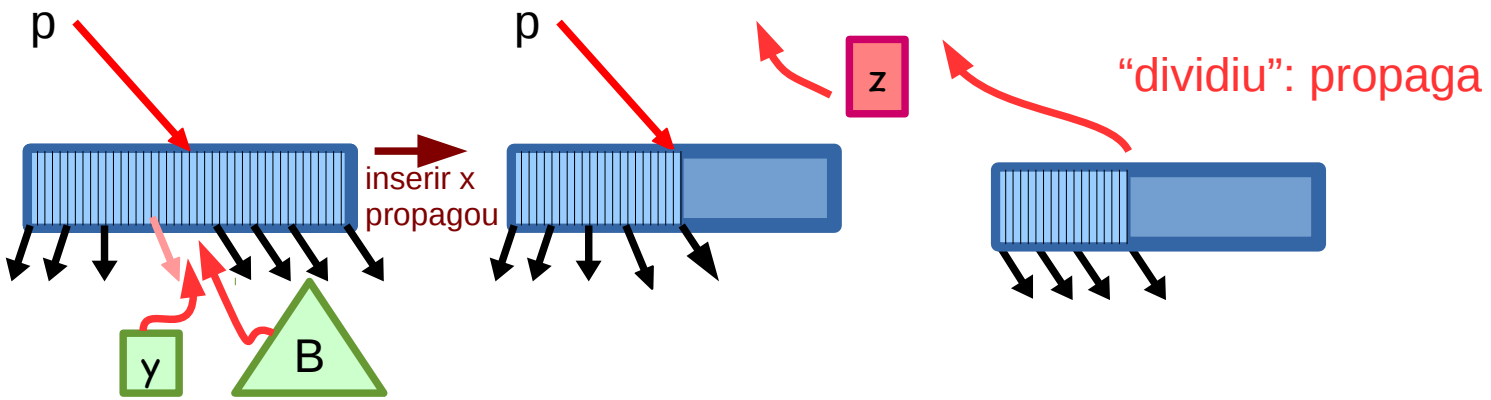


Caso 3: nó cheio



- **dividiu:** sim

Caso 3: nó cheio



- **dividiu:** sim
- **procedimento:** insere elemento propagado, aloca novo nó e divide elementos

Implementando inserção: recursão

```
// retorna 1 se dividiu
int inserir_recusivo(NoArvB *p, int x, int *y, NoArv **B) {
    if (p == NULL) { // caso 1: árvore vazia
        *y = x;
        *B = NULL;
        return 1;
    }
```

Implementando inserção: recursão

```
// retorna 1 se dividiu
int inserir_recusivo(NoArvB *p, int x, int *y, NoArv **B) {
    if (p == NULL) { // caso 1: árvore vazia
        *y = x;
        *B = NULL;
        return 1;
    } else {
        int pos = procura_posicao_insercao(p, x);
```

Implementando inserção: recursão

```
// retorna 1 se dividiu
int inserir_recusivo(NoArvB *p, int x, int *y, NoArv **B) {
    if (p == NULL) { // caso 1: árvore vazia
        *y = x;
        *B = NULL;
        return 1;
    } else {
        int pos = procura_posicao_insercao(p, x);
        int dividiu = inserir_recusivo(p->filhos[pos], y, B);
```

Implementando inserção: recursão

```
// retorna 1 se dividiu
int inserir_recusivo(NoArvB *p, int x, int *y, NoArv **B) {
    if (p == NULL) { // caso 1: árvore vazia
        *y = x;
        *B = NULL;
        return 1;
    } else {
        int pos = procura_posicao_insercao(p, x);
        int dividiu = inserir_recusivo(p->filhos[pos], y, B);
        if (dividiu) {
            if (p->tam < ORDEM ) { // caso 2: nó tem espaço
                deslocar(p, pos);
                p->chaves[pos] = *y;  p->filhos[pos] = *B;
                return 0;
            }
```

Implementando inserção: recursão

```
// retorna 1 se dividiu
int inserir_recusivo(NoArvB *p, int x, int *y, NoArv **B) {
    if (p == NULL) { // caso 1: árvore vazia
        *y = x;
        *B = NULL;
        return 1;
    } else {
        int pos = procura_posicao_insercao(p, x);
        int dividiu = inserir_recusivo(p->filhos[pos], y, B);
        if (dividiu) {
            if (p->tam < ORDEM ) { // caso 2: nó tem espaço
                deslocar(p, pos);
                p->chaves[pos] = *y;  p->filhos[pos] = *B;
                return 0;
            } else { // caso 3: nó cheio
                dividir(p, pos, y, B);
                return 1;
            }
        }
    }
}
```

Implementando inserção: chamada principal

Inserir na raiz

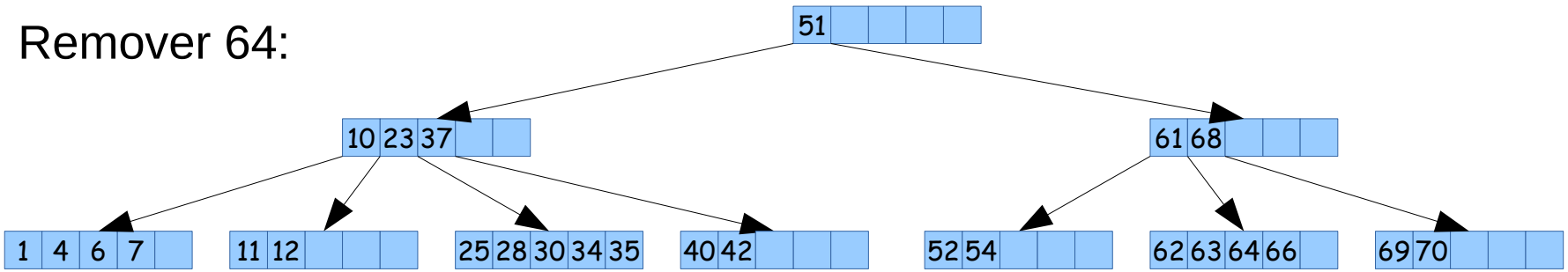
```
void inserir_arvb(NoArvB **raiz, int x) {
    int dividiu;
    int y;
    NoArvB *B, *novo;

    int dividiu = inserir_recusivo(*raiz, x, &y, &B);
```

Implementando inserção: chamada principal

Inserir na raiz

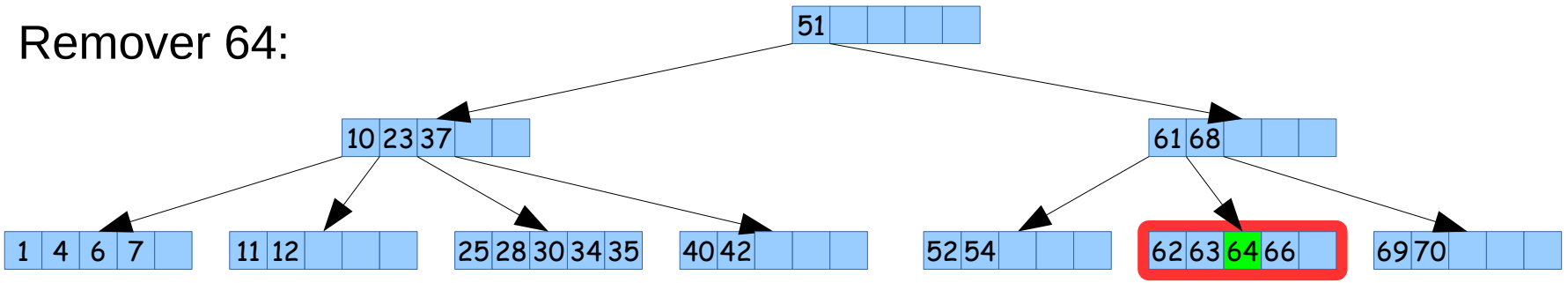
```
void inserir_arvb(NoArvB **raiz, int x) {
    int dividiu;
    int y;
    NoArvB *B, *novo;

    int dividiu = inserir_recusivo(*raiz, x, &y, &B);
    if (dividiu) {
        // árvore aumenta de altura
        novo = mallox(sizeof(NoArvB));
        novo->tam = 1;
        novo->filhos[0] = *raiz;
        novo->chaves[0] = y;
        novo->filhos[1] = B;

        *raiz = novo;
    }
}
```

Removendo em nó cheio



Procedimento

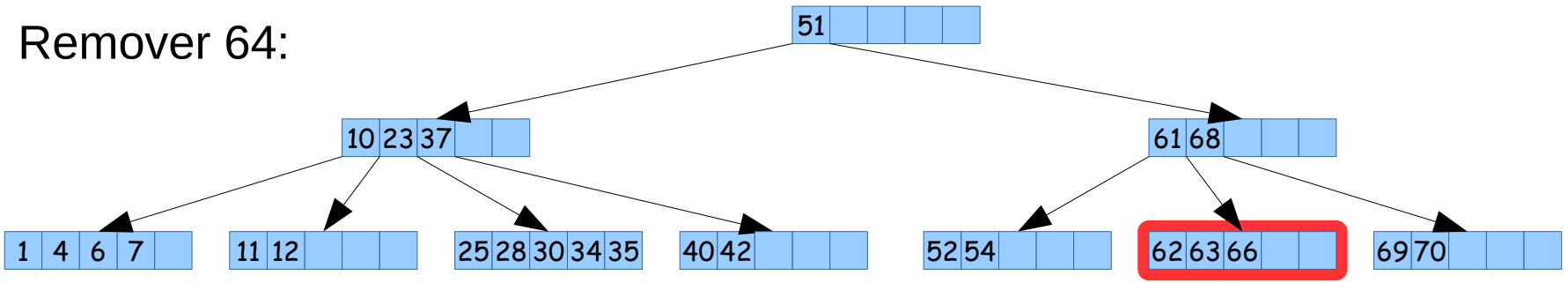
Removendo em nó cheio



Procedimento

1. procuramos folha

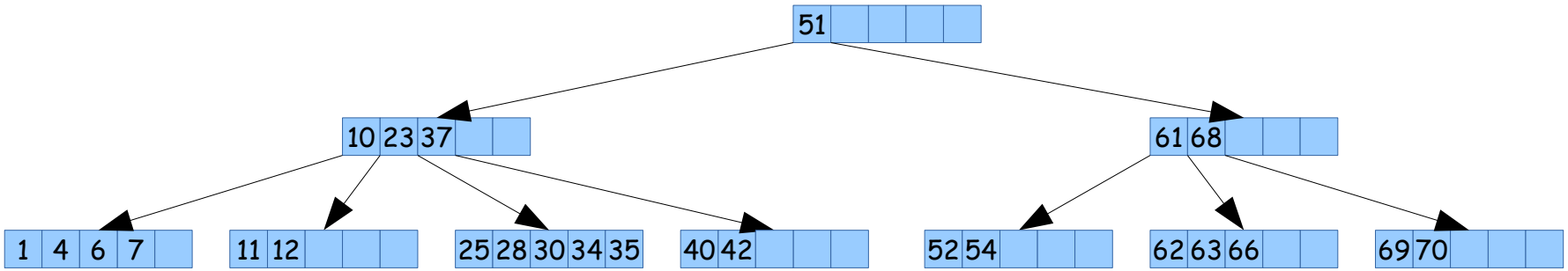
Removendo em nó cheio



Procedimento

1. procuramos folha
2. e removemos

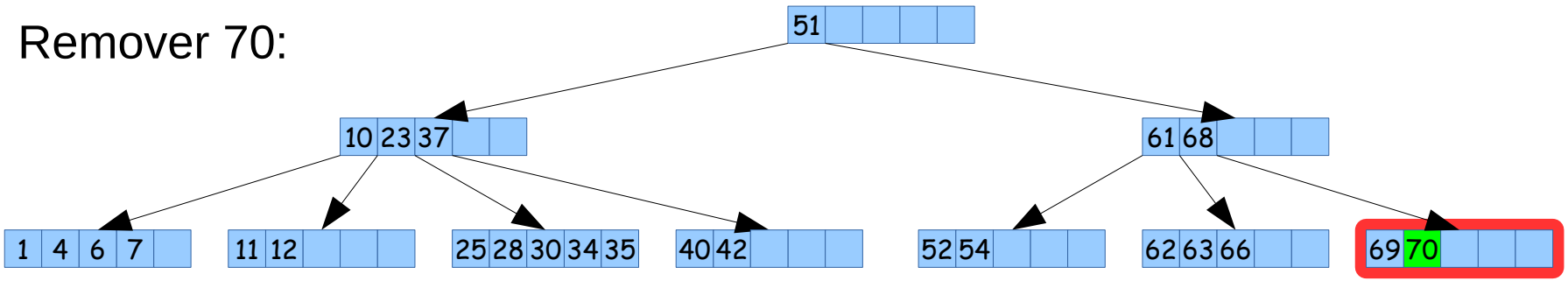
Removendo em nó cheio



Procedimento

1. procuramos folha
2. e removemos

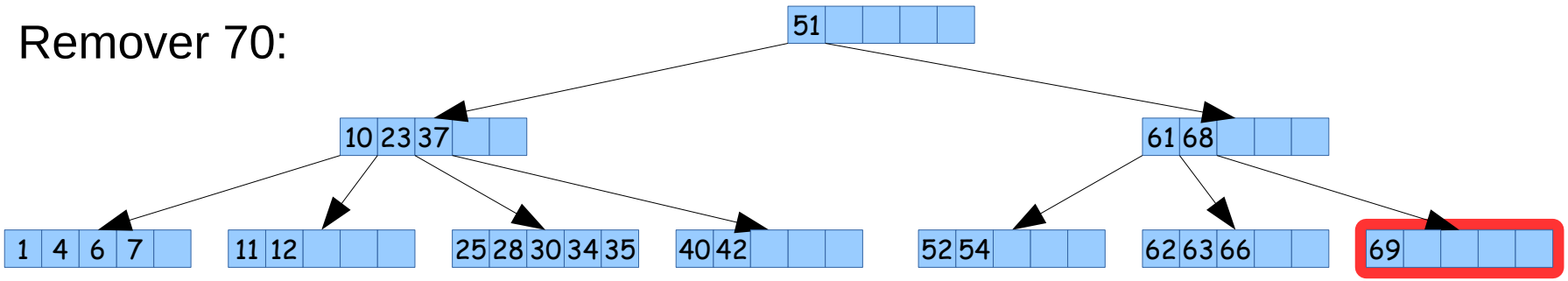
Removendo nó vazio com irmão cheio



Procedimento

1. procuramos folha

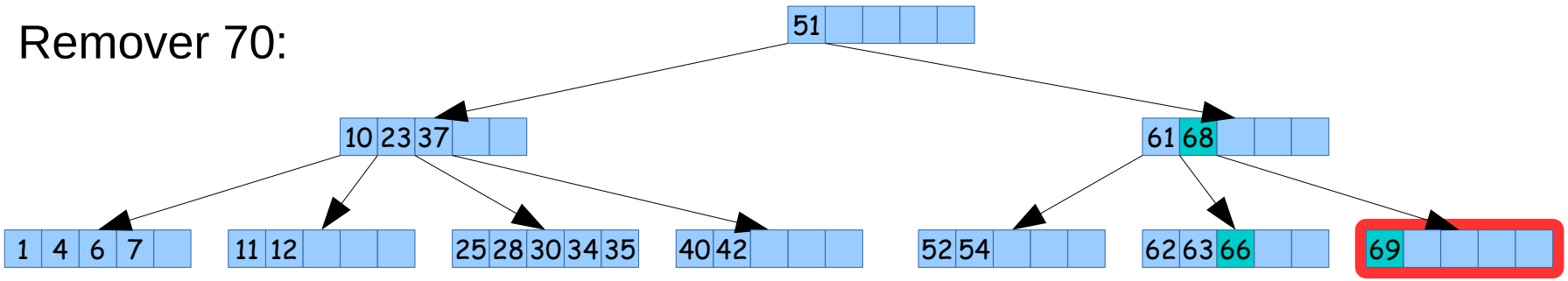
Removendo nó vazio com irmão cheio



Procedimento

1. procuramos folha e removemos

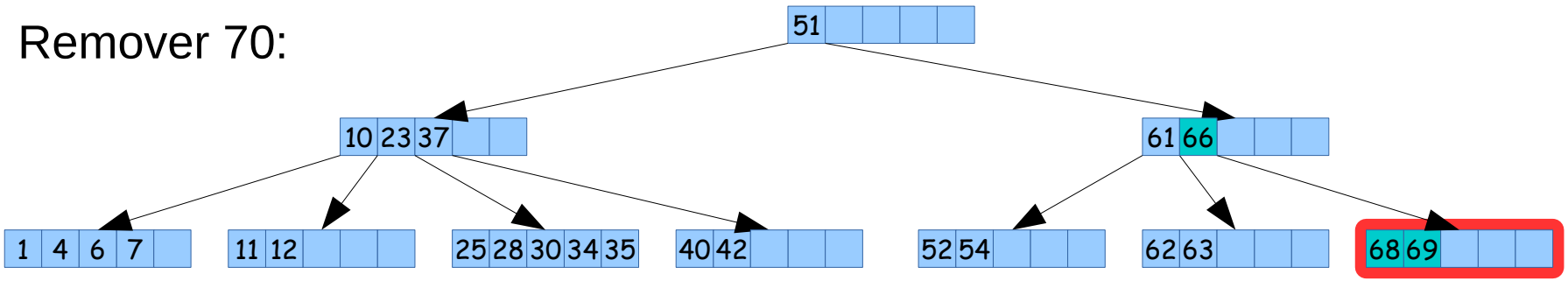
Removendo nó vazio com irmão cheio



Procedimento

1. procuramos folha e removemos
2. pegamos “emprestado” do irmão

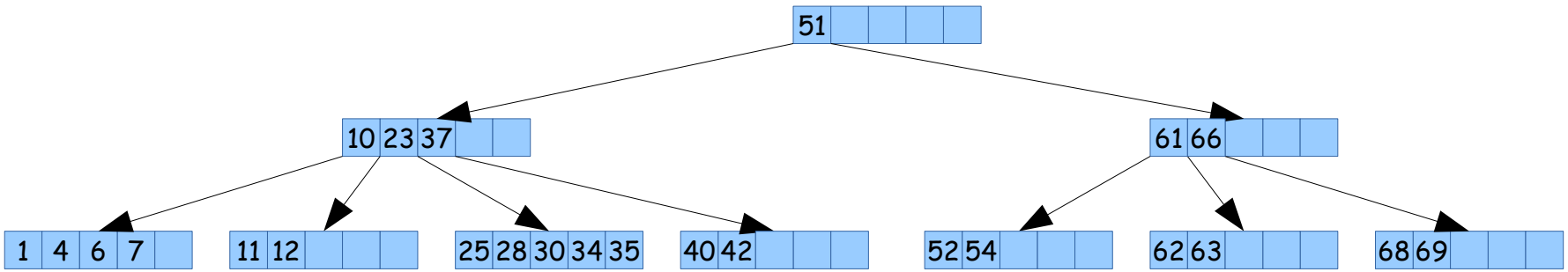
Removendo nó vazio com irmão cheio



Procedimento

1. procuramos folha e removemos
2. pegamos “emprestado” do irmão

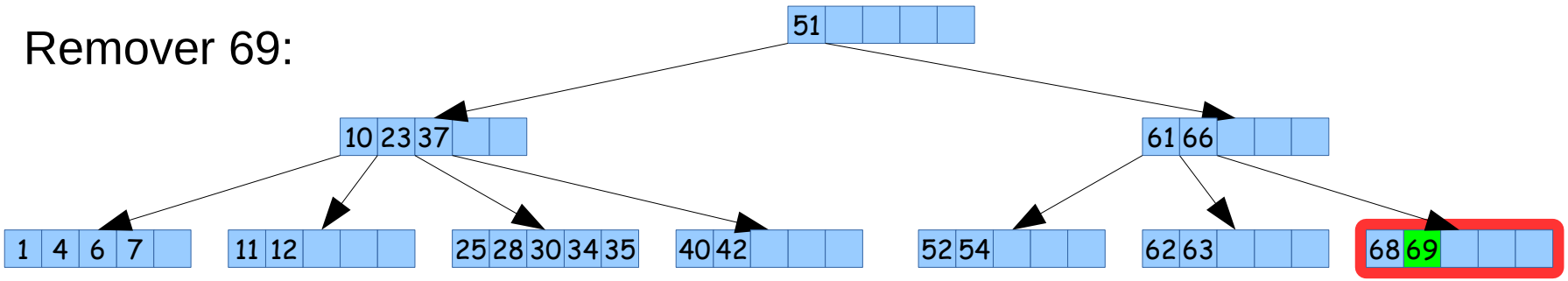
Removendo nó vazio com irmão cheio



Procedimento

1. procuramos folha e removemos
2. pegamos “emprestado” do irmão

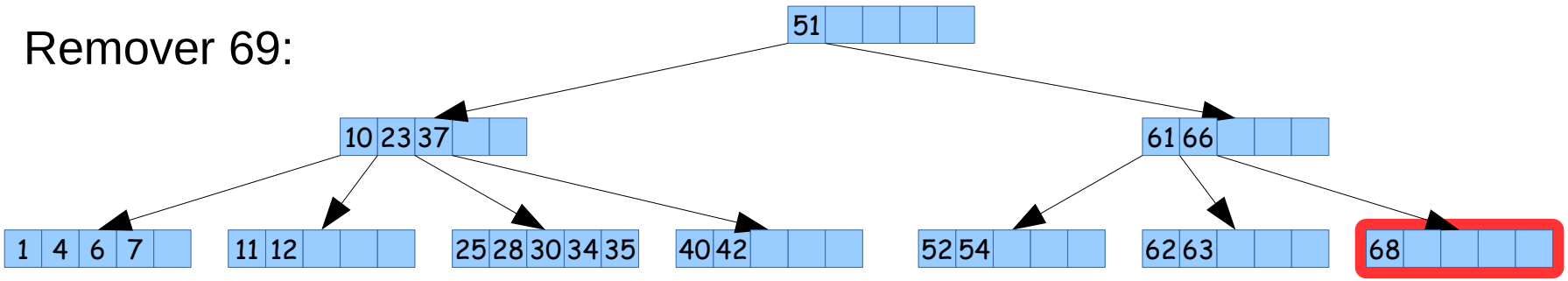
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha

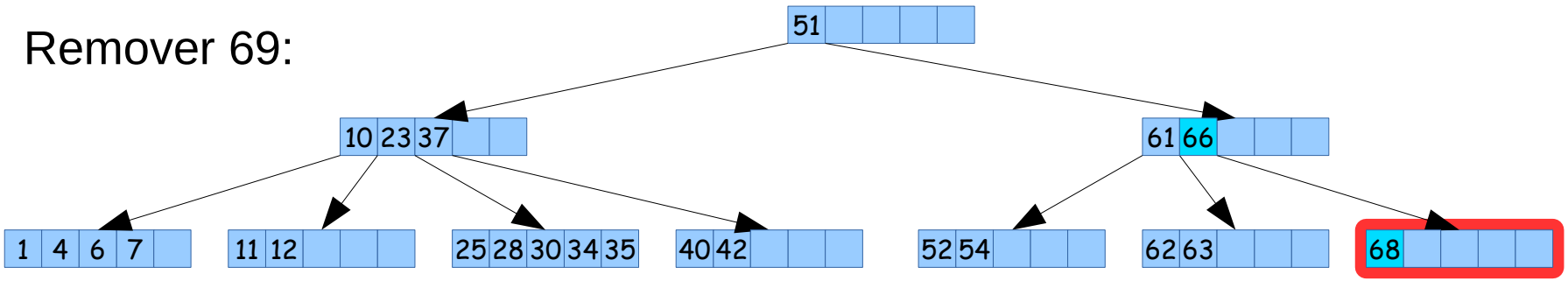
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos

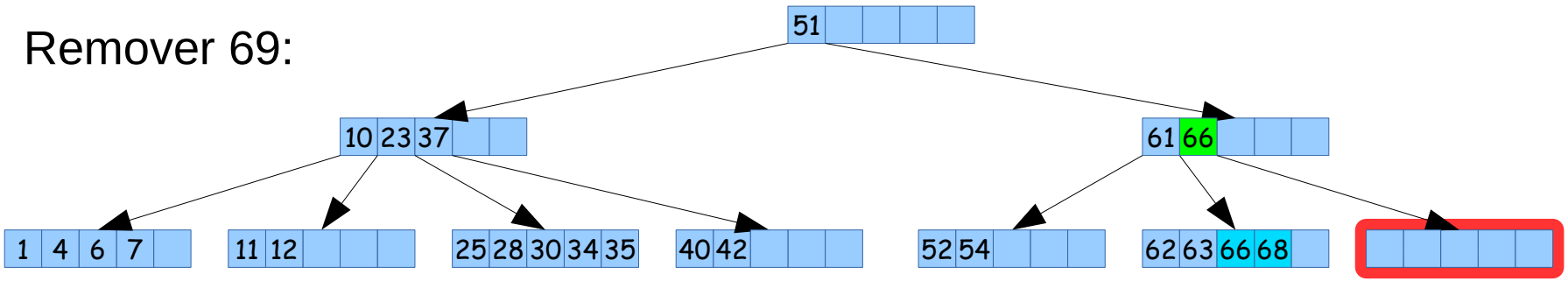
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão

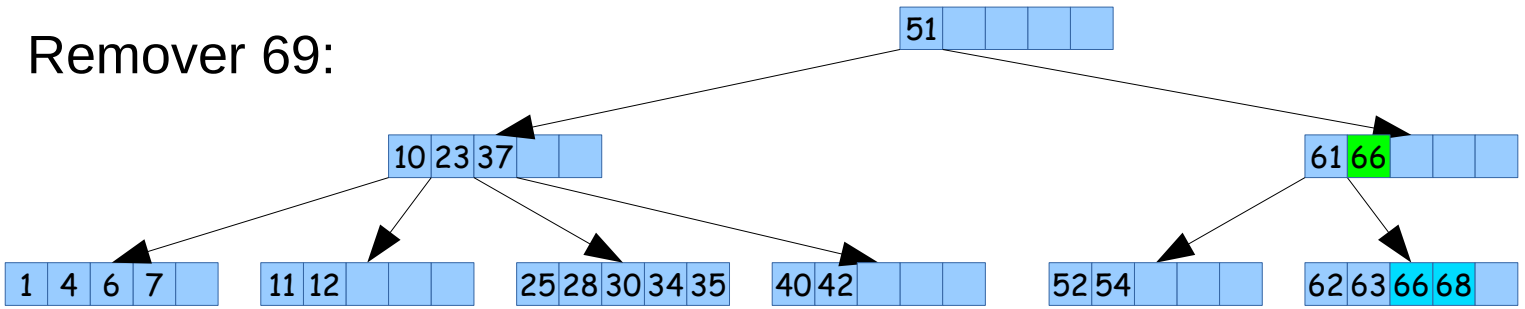
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão

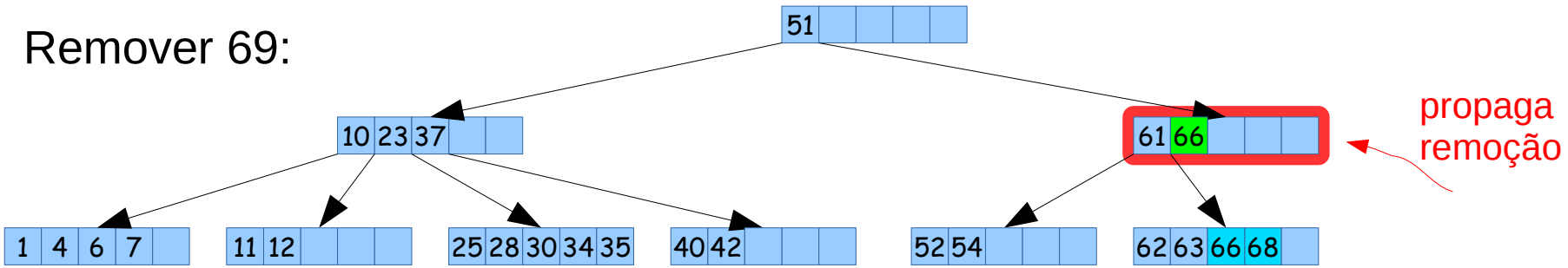
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão
3. removemos o nó

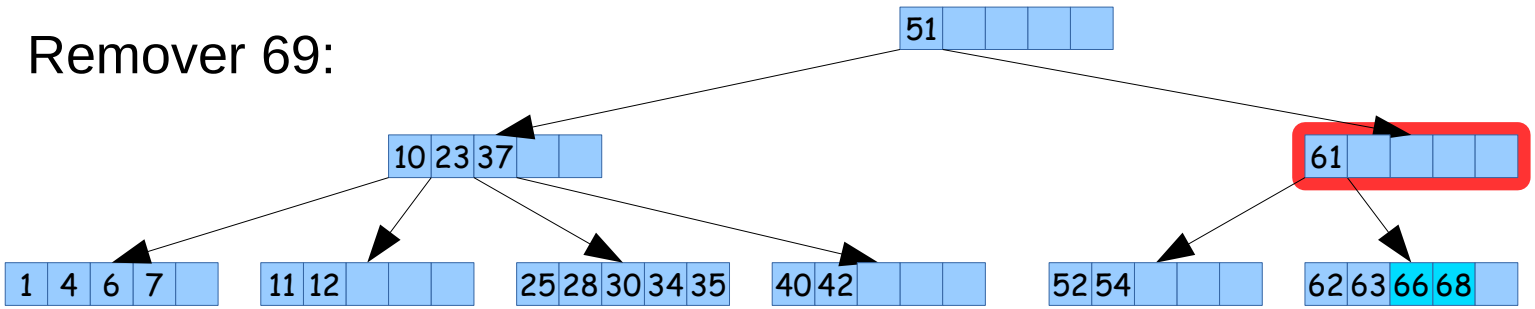
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão
3. removemos o nó
4. propagamos remoção

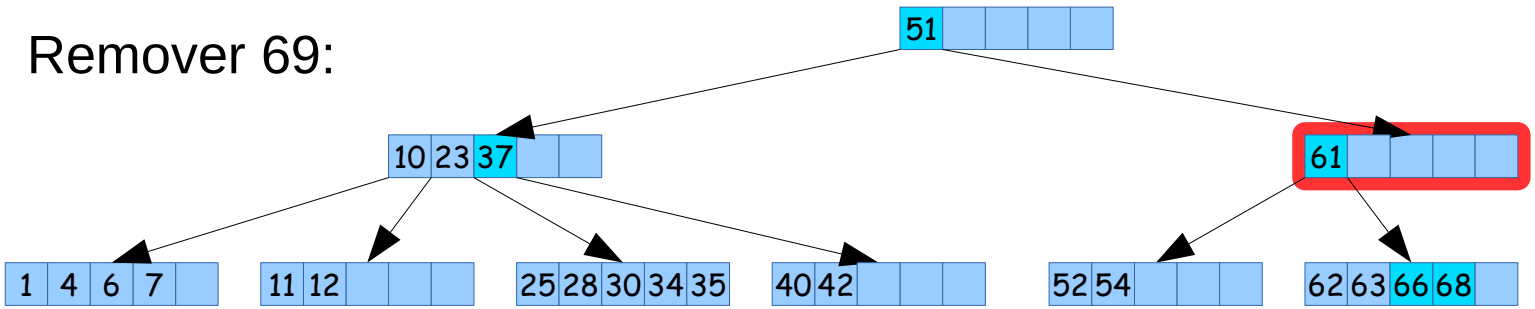
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão
3. removemos o nó
4. propagamos remoção

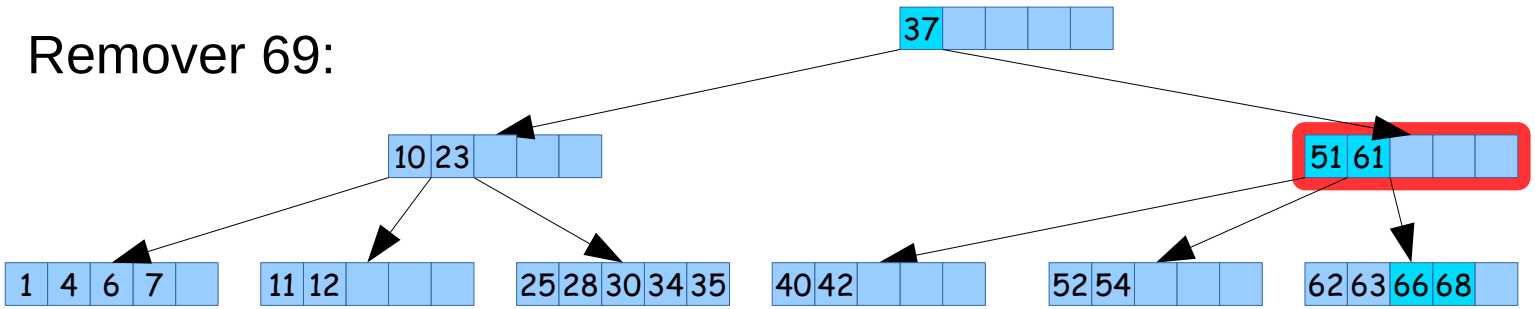
Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão
3. removemos o nó
4. propagamos remoção

Removendo nó vazio com irmão vazio



Procedimento

1. procuramos folha e removemos
2. juntamos pai e imão
3. removemos o nó
4. propagamos remoção

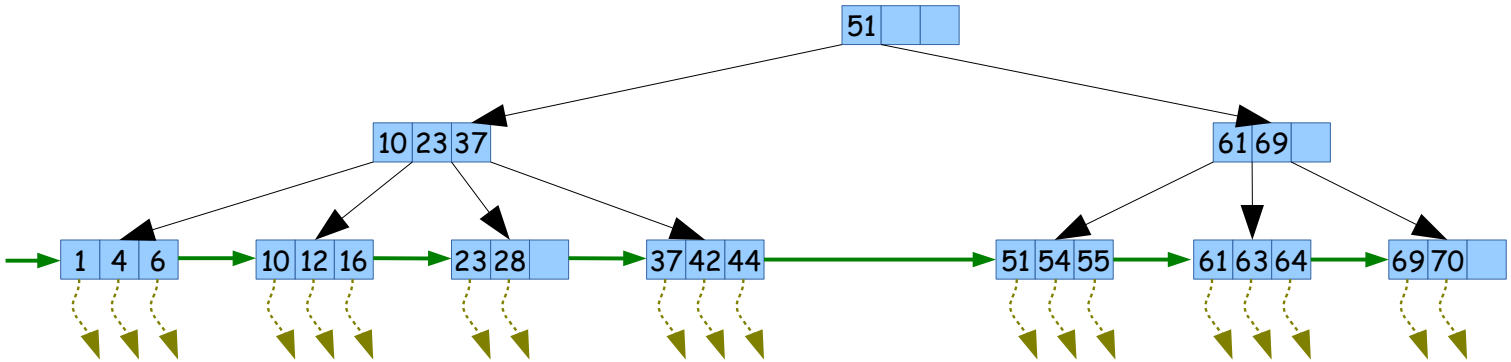
Alguns fatos

- A ocupação mínima dos nós da árvore é 50%
- Na prática, a ocupação é de cerca de $\approx 70\%$
- Todas operações são da ordem da altura da árvore: $O(\log_b n)$
- Para $b = 255$ e n até 1 bilhão , altura ≤ 4**!**
- É comum manter a raiz na memória.
- Existem algoritmos eficientes que controem a árvore dado um conjunto de elementos ordenados.

Variantes

Variantes da Árvore-B

- **Árvore-B***
  - mantém taxa de ocupação mínima de 2/3
- **Árvore-B+**
  - todos os registros são salvos nas folhas
  - as folhas mantém apenas um ponteiro para os dados
  - listar todos elementos em ordem é eficiente



Árvore-B+

Exercício

1. Implemente as sub-rotinas que faltam na inserção.
2. Esboce um algoritmo para a remoção de um elemento de uma Árvore-B.
3. Insira os elementos 8 e 9 na árvore do slide 9. Qual a árvore obtida?
4. Remova em ordem os elementos 52, 55, 55, 62, 63, 64 da árvore do slide 9. Qual a árvore obtida?

Exercício 2 - Aplicações de Árvore B

1. Uma aplicação típica de Árvores-B é a criação de índices em sistemas de gerenciamento de banco de dados (responda: o que é um índice?). Embora o índice natural corresponde à "chave", outros índices podem ser úteis. Do ponto de vista de estrutura de dados, é possível manter duas Árvores-B para o mesmo conjunto de dados? Quais complicações existem?
2. Com o avanço das tecnologias de memórias persistentes (disco de estado sólido, etc.), a latência é cada vez menor.
   - pesquise sobre quais memórias persistentes existem e seus tempos de latência, acesso, etc.;
   - de acordo com sua pesquisa, para cada tipo de memória, ainda é vantajoso usar uma Árvore-B? Justifique sua resposta.