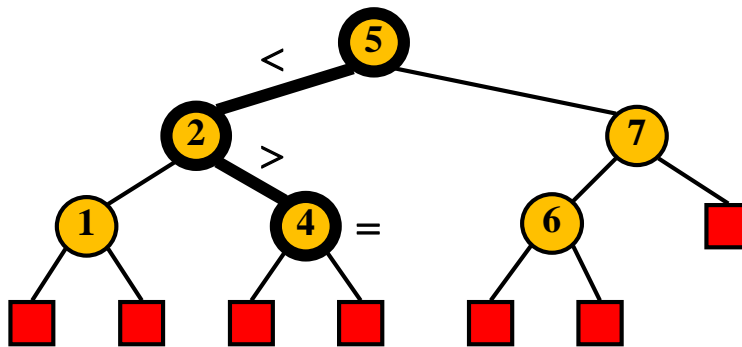


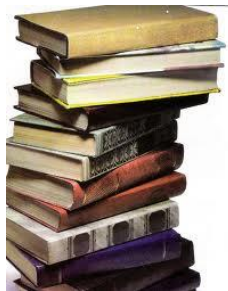
# Algoritmos e Estrutura de Dados – IV

## Unidade 6 – Árvore Binária de Pesquisa

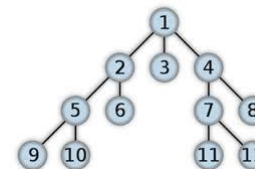


Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP





# Bibliografia

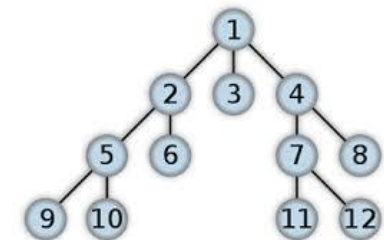
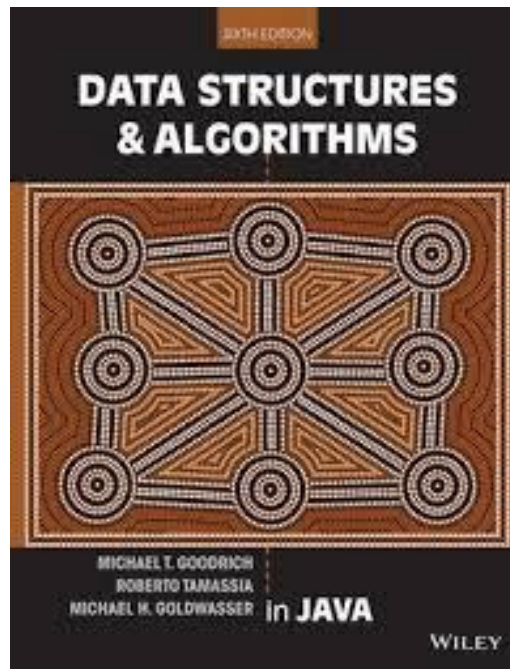


- Data Structures and Algorithms in Java – Sixty Edition – Roberto Tamassia – Michael T. Goodrich – John Wiley & Sons, Inc
- Head First Java, 2nd Edition by Kathy Sierra and Bert Bates
- Estrutura de Dados e Algoritmos – Bruno R. Preiss, Editora Campus, 2001
- Estrutura de Dados e Algoritmos em Java – Robert Lafore, Editora Ciência Moderna, 2005
- Algoritmos e Estrutura de Dados – Niklaus Wirth – Editora Prentice Hall do Brasil, 1989
- Estrutura de Dados e Algoritmos em C++, Adam Drozdek – Thompson
- Introdução à Estrutura de Dados, Celes, Cerqueira, Rangel - Elsevier



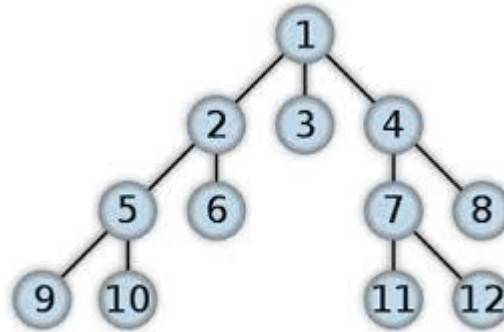
## Leitura Recomendada

- ⊕ Data Structures and Algorithms in Java (\*), Roberto Tamassia and Michael T. Goodrich, Sixty Edition – **2014** , Seção **8.2**

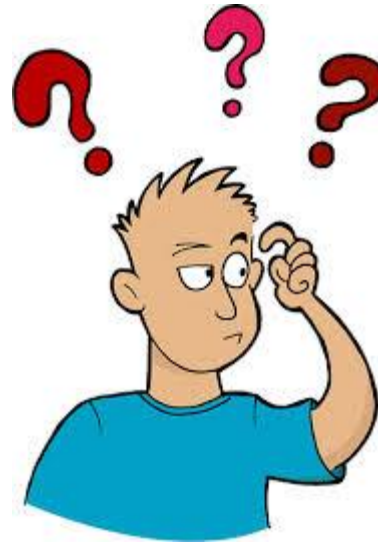


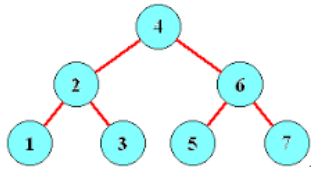
(\*) Em português, Estrutura de Dados e Algoritmos em Java





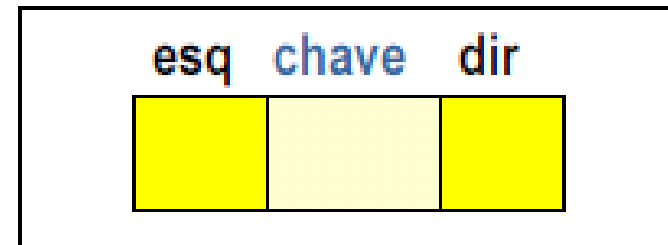
O que é árvore binária de pesquisa ?





# Árvore Binária de Pesquisa

- Também conhecida por:
  - Árvore Binária de Busca
  - Árvore Binária Ordenada
  - Search Tree (em inglês)
- Apresentam uma relação de ordem entre os nós.
- A ordem é definida por um campo **chave** (key).
- Não permite chaves duplicadas.

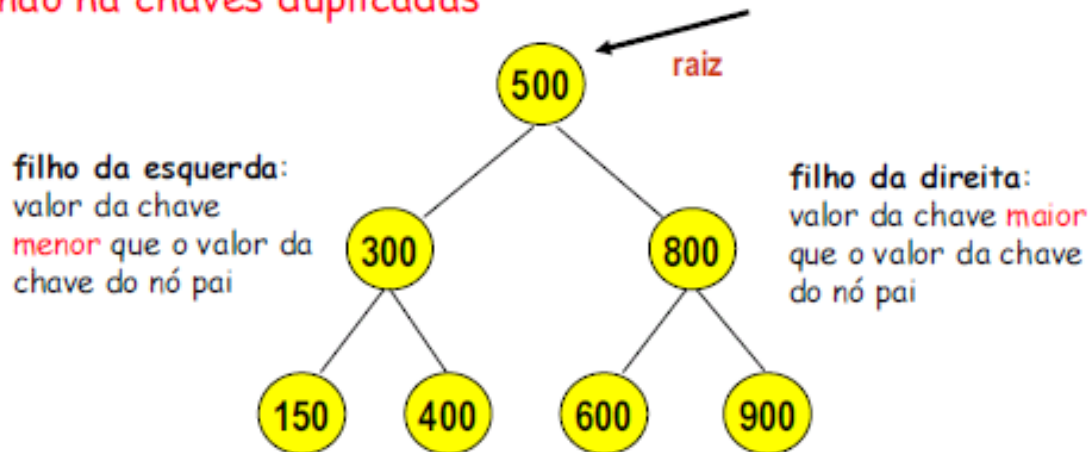


# Árvore Binária de pesquisa

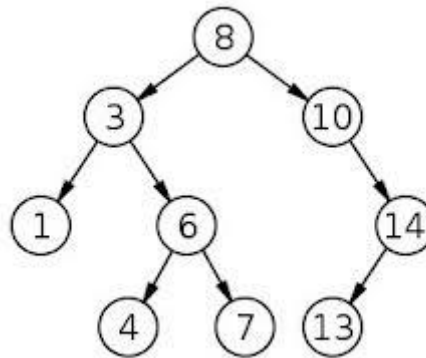
## Definição de Niklaus Wirth:

Árvore que se encontra organizada de tal forma que,  
para cada nó  $t_i$ , todas as chaves da sub-árvore:  
à **esquerda** de  $t_i$  são **menores** que  $t_i$  e  
à **direita** de  $t_i$  são **maiores** que  $t_i$ .

não há chaves duplicadas



# Inserção em Árvores Binárias de Pesquisa



# Carga da Árvore Binária de pesquisa

```
int[] valores = { 17,49,14,23,27,15,2,1,34,10,12 } ;
```

```
String[] nomes = {  
"Paulo","Ana","José","Rui","Paula","Bia","Selma","Carlos","Silvia","Teo","Saul" } ;
```

**A partir das listas acima, implementar  
a árvore binária de busca.**



```
while( n < (docum  
{  
  
    n++;  
    calc = ev  
    i++  
    i++
```





# Inserção em uma árvore de busca binária

Lembrando que ...

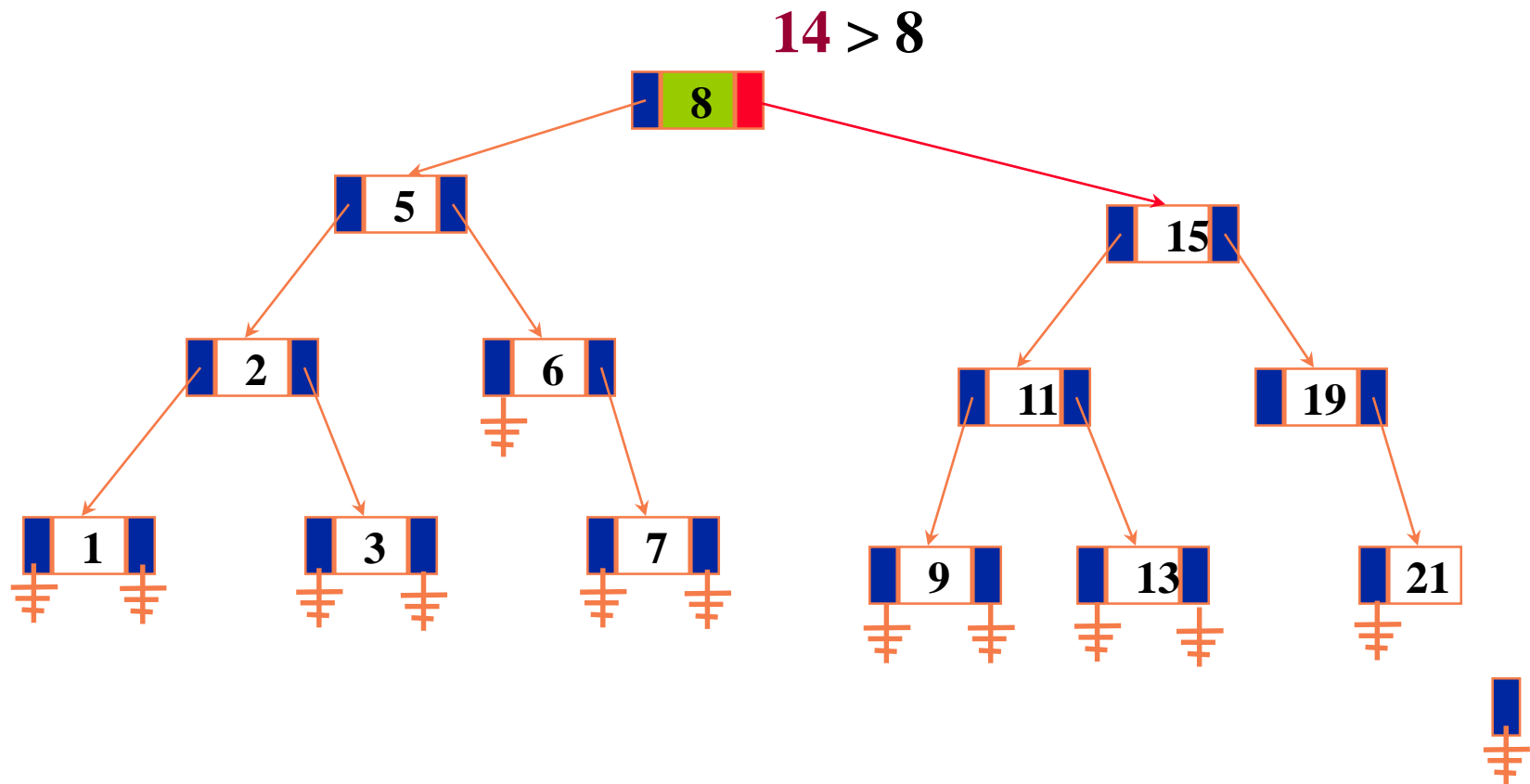
- A sub-árvore da **direita** de um nó deve possuir chaves **maiores** que a chave do pai.
- A sub-árvore da **esquerda** de um nó deve possuir chaves **menores** que a chave do pai.

## *Princípio Básico*

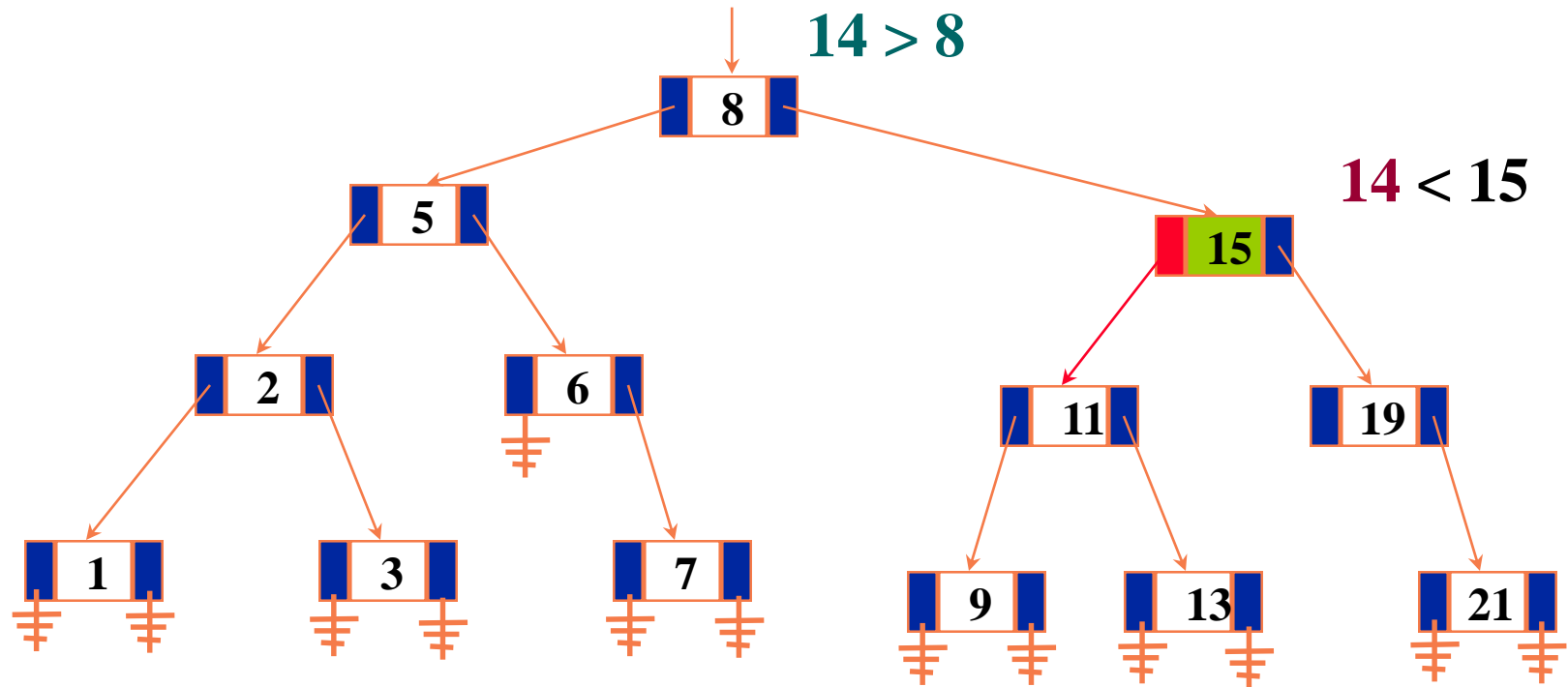
- ◆ Percorrer a árvore até encontrar um nó sem filho, de acordo com os critérios acima.



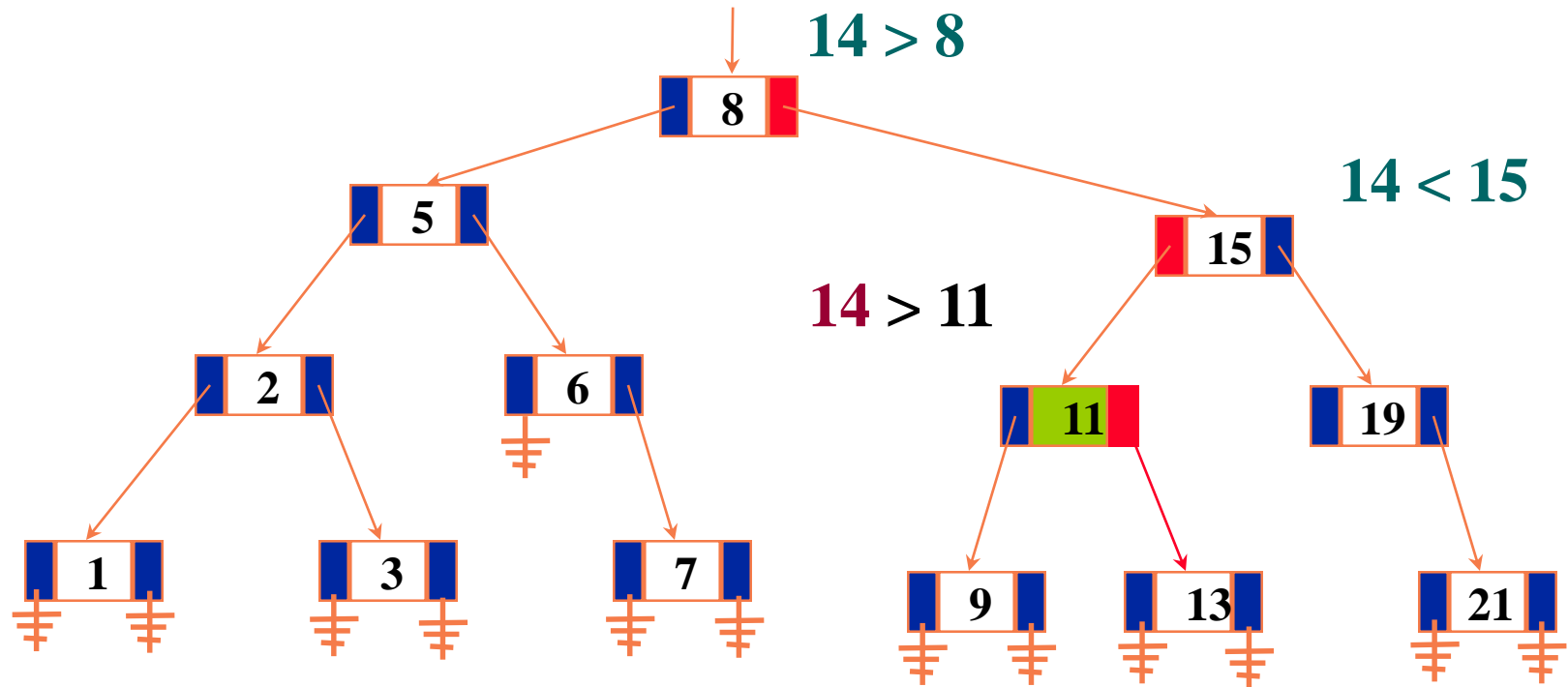
# Exemplo – Inserção de elemento com chave 14



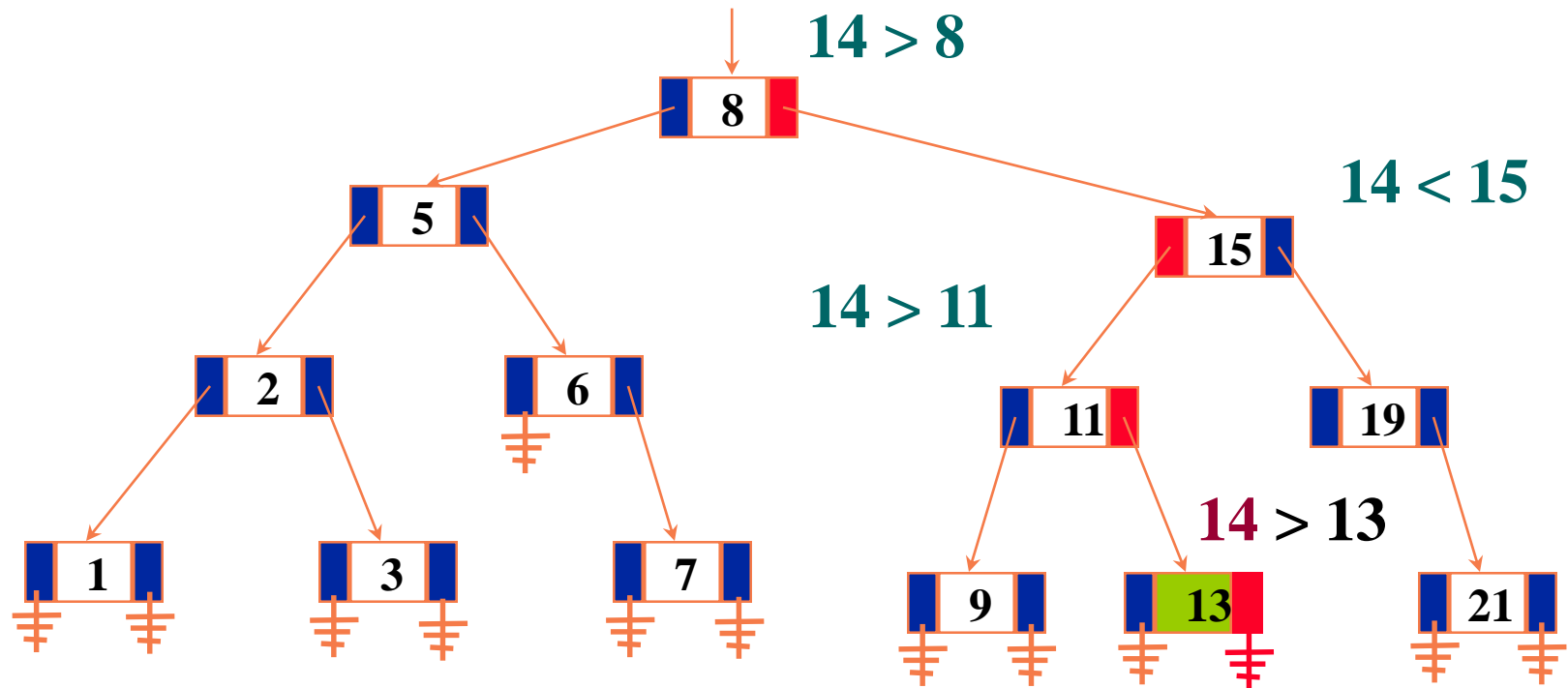
# Exemplo – Inserção de elemento com chave 14



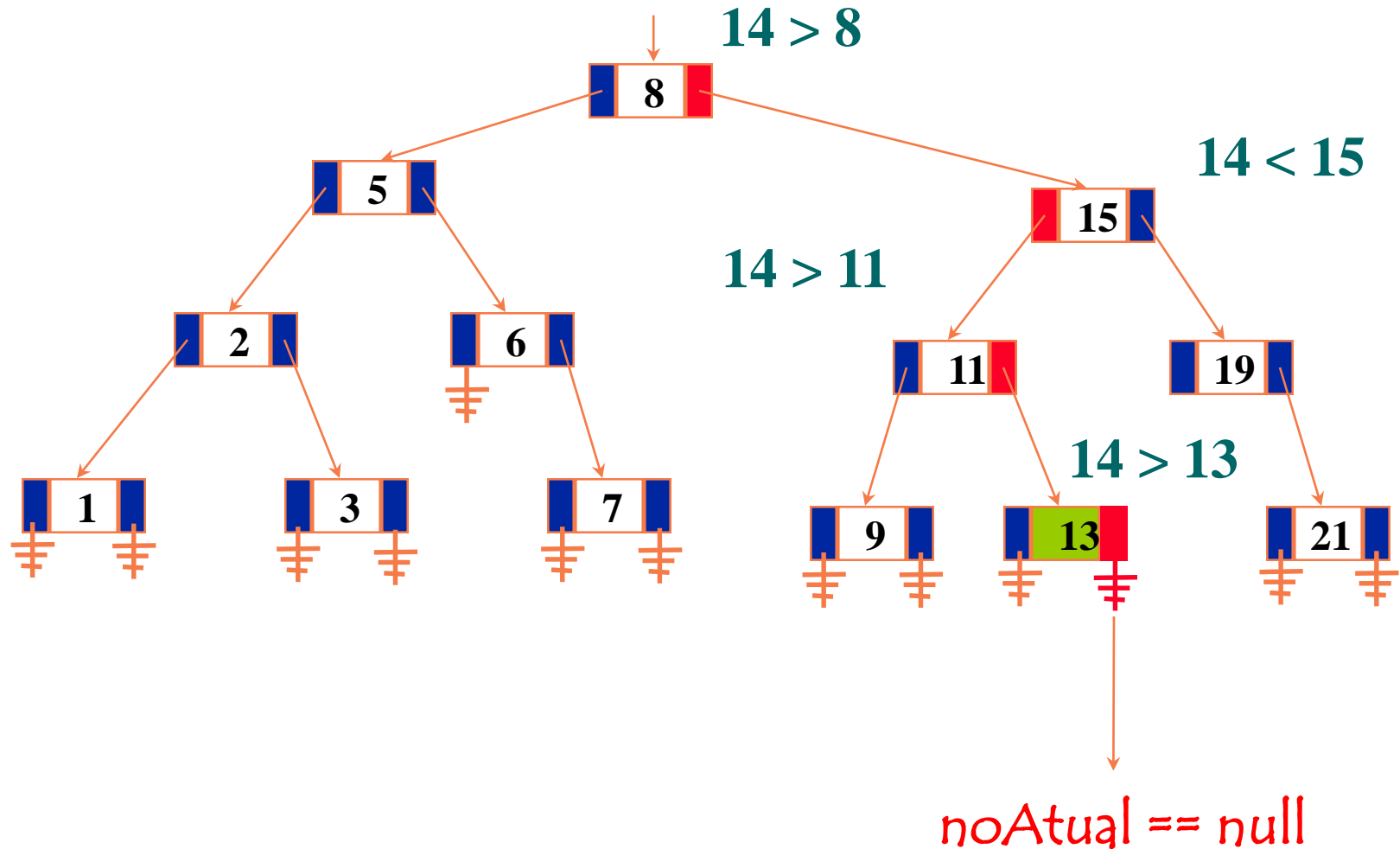
# Exemplo – Inserção de elemento com chave 14



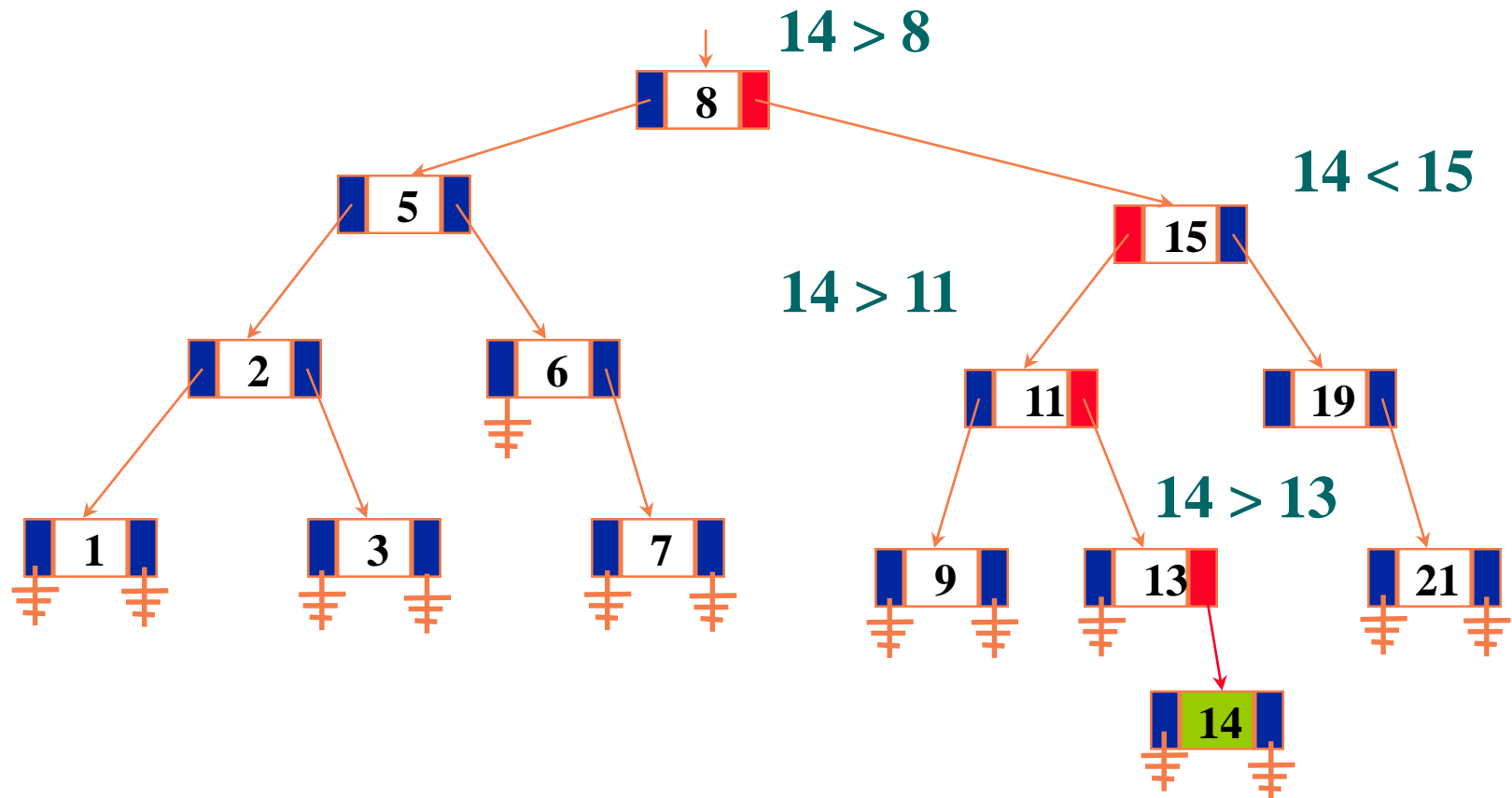
# Exemplo – Inserção de elemento com chave 14



# Exemplo – Inserção de elemento com chave 14



# Exemplo – Inserção de elemento com chave 14



# Implementação da função addnode()



```
public void addNode(int chave, String nome) {

    SearchTreeNode newNode = new SearchTreeNode(chave,nome);
    if (root == null)
        this.insert_root(newNode);
    else {
        SearchTreeNode NodeTrab = this.root;
        NodeTrab = this.root;
        while (true) {
            if (chave < NodeTrab.key) {
                if (NodeTrab.left == null) {
                    NodeTrab.left = newNode;
                    newNode.parent = NodeTrab;
                    newNode.nome = nome;
                    return;
                }
                else NodeTrab = NodeTrab.left;
            }
            else {
                if (NodeTrab.right == null) {
                    NodeTrab.right = newNode;
                    newNode.parent = NodeTrab;
                    newNode.nome = nome;
                    return;
                }
                else NodeTrab = NodeTrab.right;
            }
        }
    }
}
```





# Busca em árvores de pesquisa

- ✓ Em uma árvore binária é possível encontrar qualquer chave existente **X** atravessando-se árvore:
  - ✓ sempre **à esquerda** se **X** for **menor** que a chave do nó visitado e
  - ✓ sempre **à direita** toda vez que for **maior**.
  - ✓ A escolha da direção de busca só depende de **X** e da chave que o nó atual possui.
  
- ✓ A busca de um elemento em uma **árvore balanceada** com **n** elementos toma tempo médio menor que  $\log_2(n)$ , tendo a busca então  $O(\log_2 n)$ .



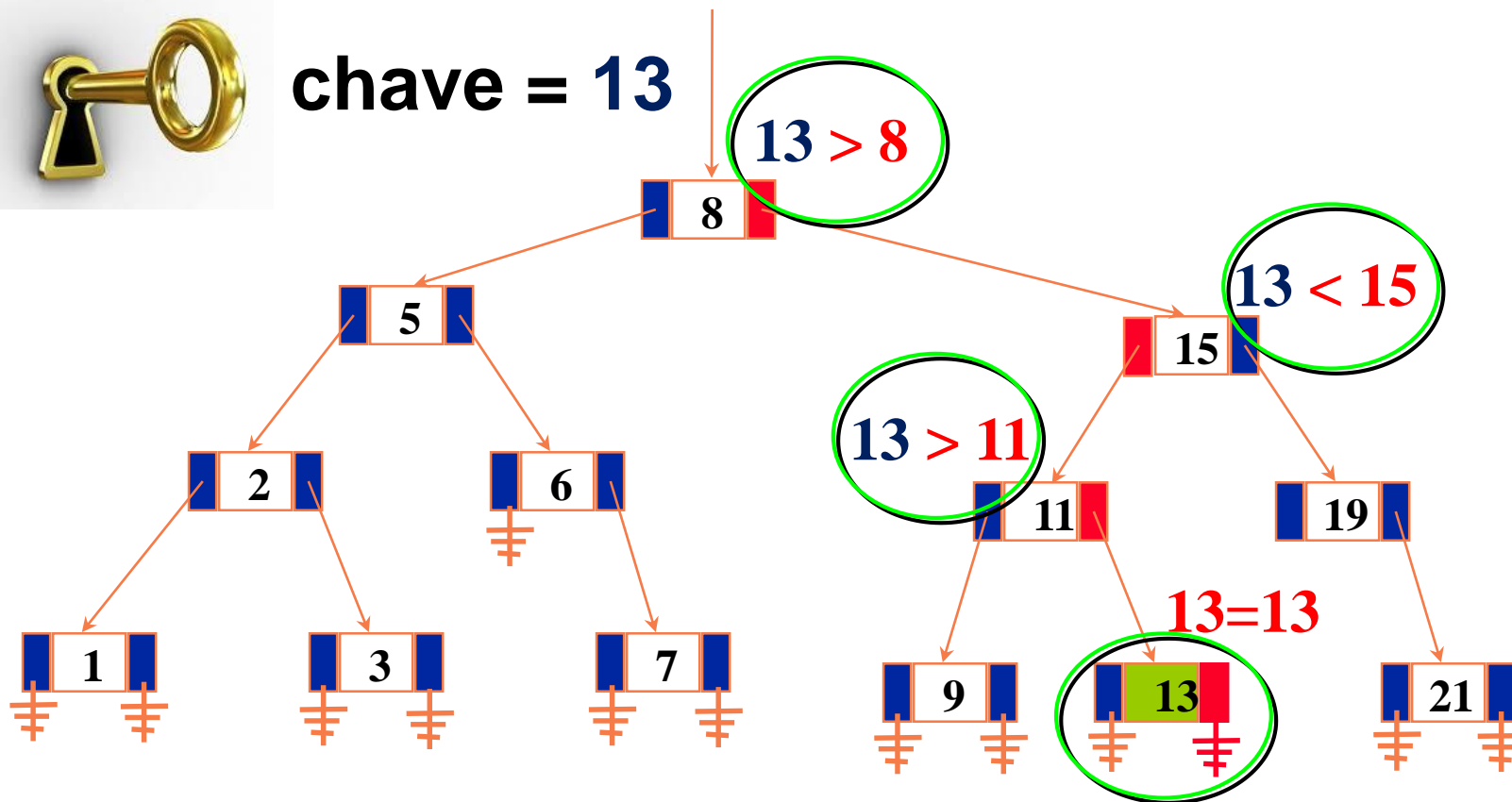
n	$\log_2(n)$
1	0,00
10	3,32
13	3,70
20	4,32
50	5,64
100	6,64
200	7,64
500	8,97



# Exemplo – Busca da chave 13



chave = 13



# Algoritmo iterativo de search em árvore de busca



```
Node buscaChave (int chave) {  
    Node noAtual = raiz; // inicia pela raiz  
  
    while (noAtual != null && noAtual.item != chave) {  
  
        if (chave < noAtual.key)  
            noAtual=noAtual.left ; // caminha p/esquerda  
        else  
            noAtual=noAtual.right; // caminha p/direita  
    }  
    return noAtual;  
}
```

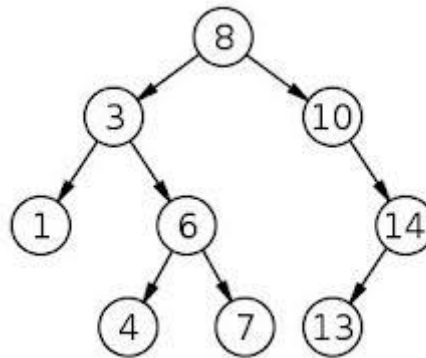


# Implementação – Busca

```
public SearchTreeNode Search_key(int key) {  
  
    SearchTreeNode nodeTrab = this.root; // inicia pela raiz  
  
    while (nodeTrab != null && nodeTrab.key != key) {  
  
        if (key < nodeTrab.key)  
            nodeTrab = nodeTrab.left;  
        else  
            nodeTrab = nodeTrab.right;  
    }  
    return nodeTrab;  
}
```

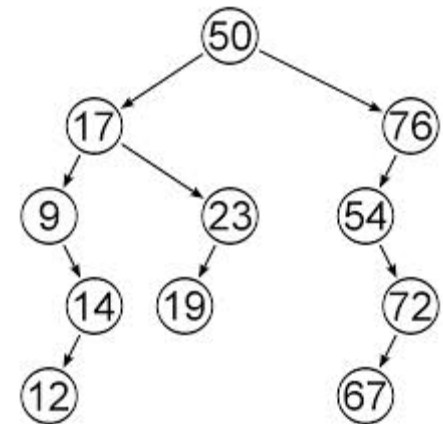


# Eliminação em Árvores Binárias de Busca



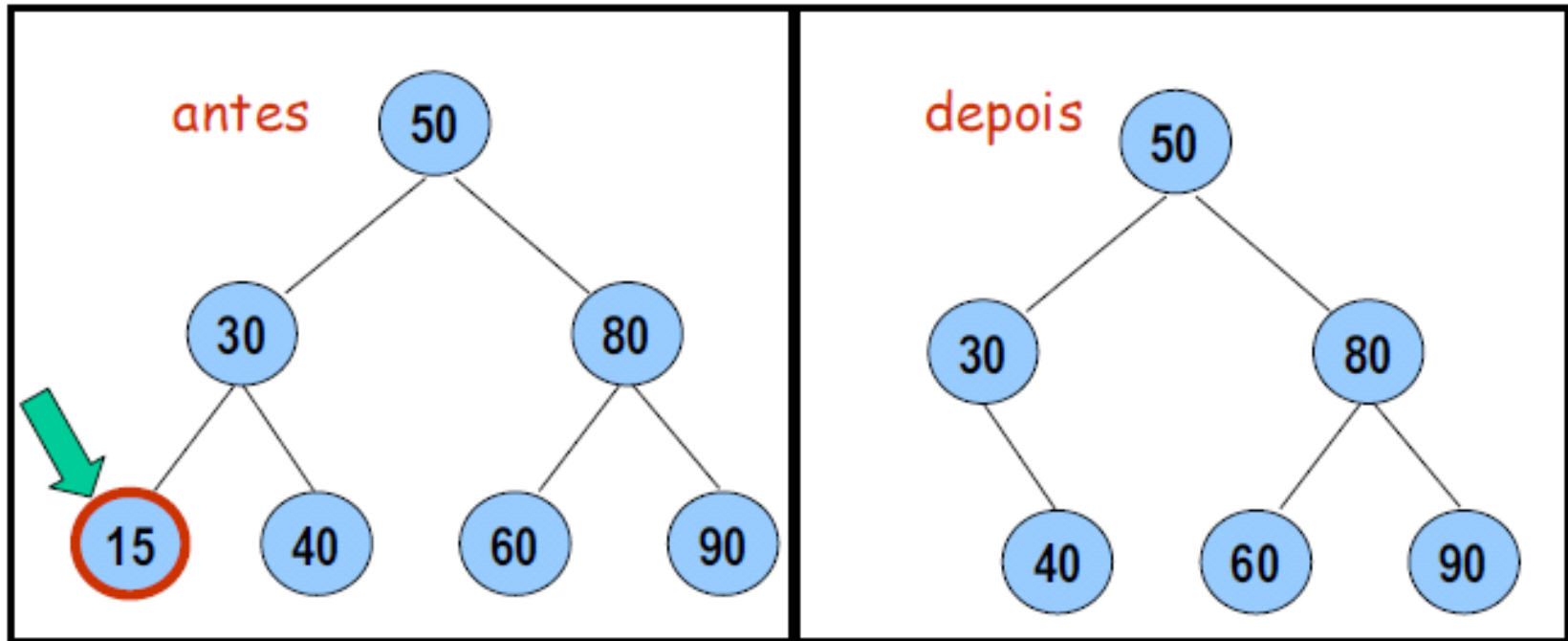
# Eliminação em Árvores Binárias de Busca

- ✓ A eliminação é mais complexa do que a inserção.
- ✓ A razão básica é que a característica organizacional da árvore não deve ser alterada.
  - A sub-árvore **direita** de um nó deve possuir chaves **maiores** que a do pai.
  - A sub-árvore **esquerda** de um nó deve possuir chaves **menores** que a do pai.
- ✓ Para garantir isto, o algoritmo deve “**remanejar**” os nós.



# Caso 1 – Remoção de nó folha

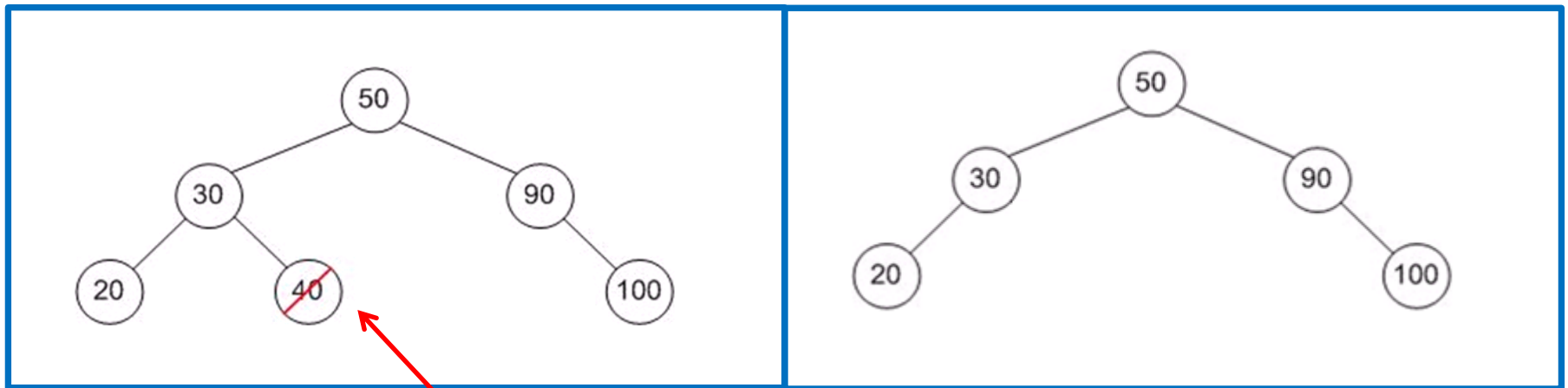
Caso mais simples, basta retirá-lo da árvore



Eliminação da chave 15



# Caso 1 – Outro exemplo



Eliminação da chave 40



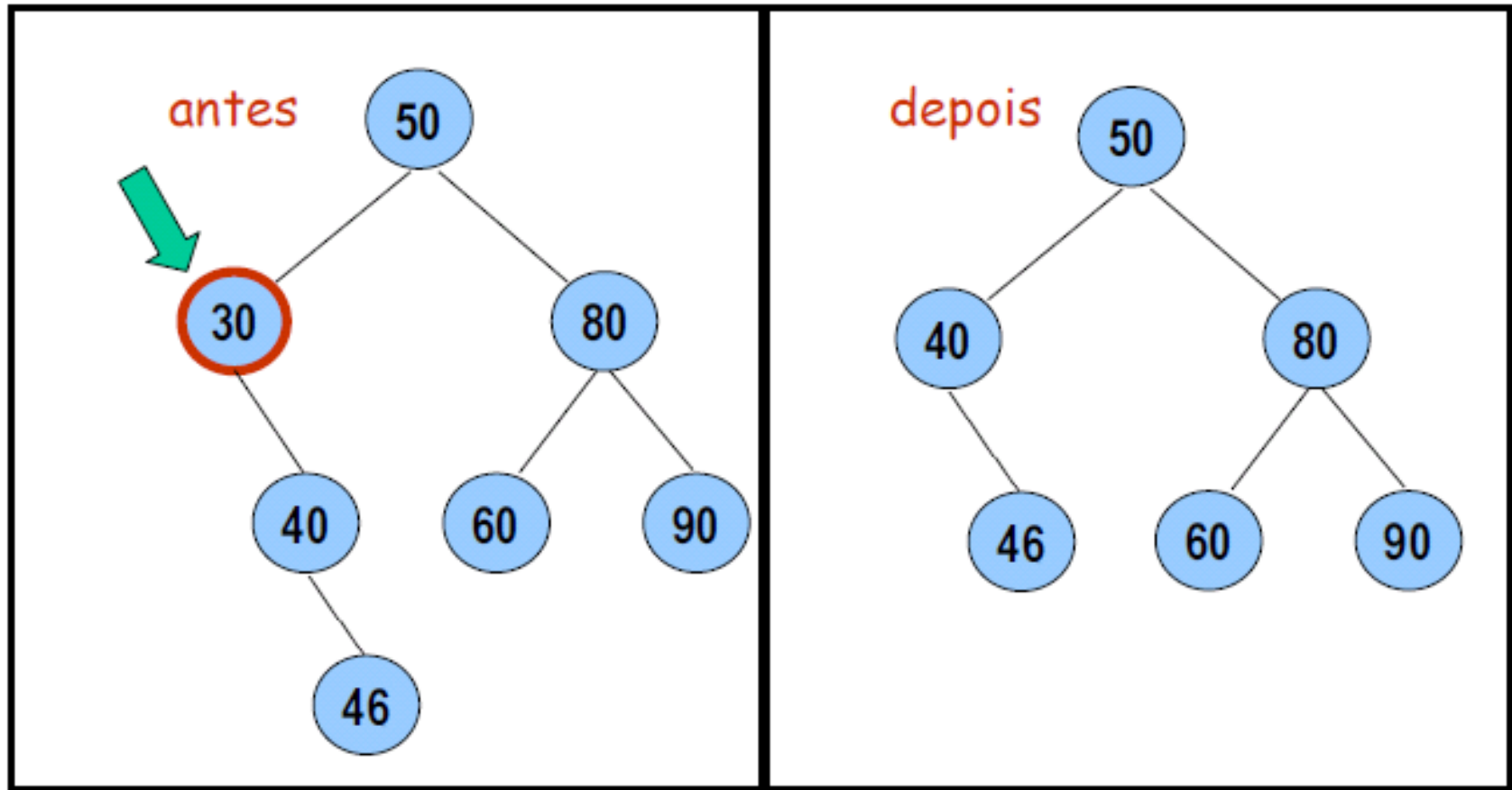


# Eliminação de nó que possui uma sub-árvore filha

- ✓ Se o nó a ser removido possuir somente uma sub-árvore filha:
  - ⊕ Move-se essa sub-árvore toda para cima.
  - ⊕ Se o nó a ser excluído é filho esquerdo de seu pai, o seu filho será o novo filho esquerdo deste e vice versa.



## Caso 2 – O nó tem somente uma sub-árvore

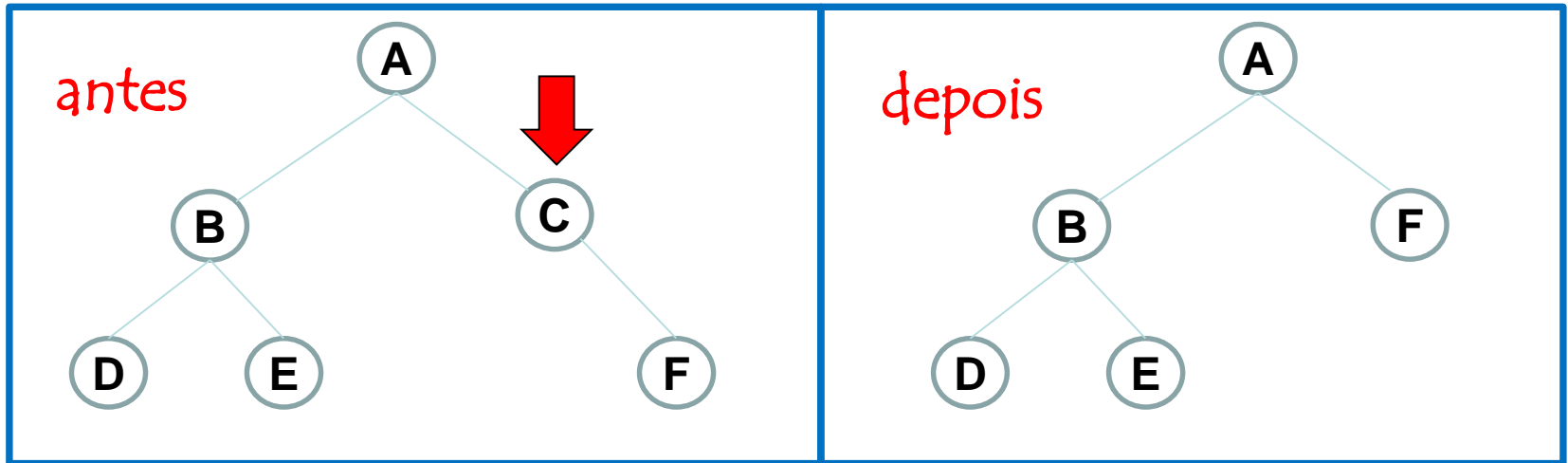


Eliminação da chave 30

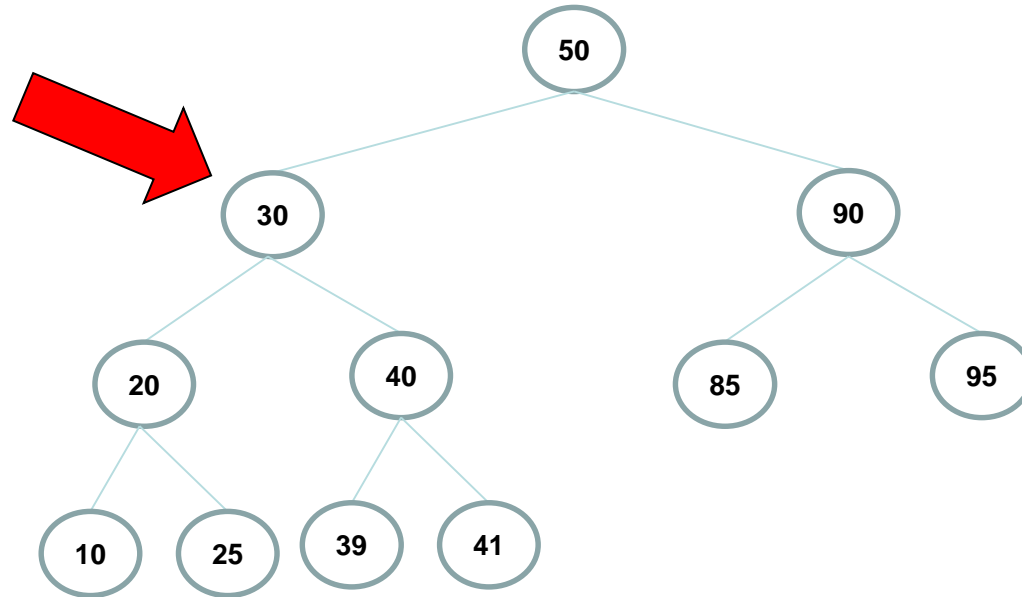
O ponteiro do pai aponta para o filho deste



## Caso 2 – Outro Exemplo



# Eliminação de nó que possui duas sub-árvores filhas

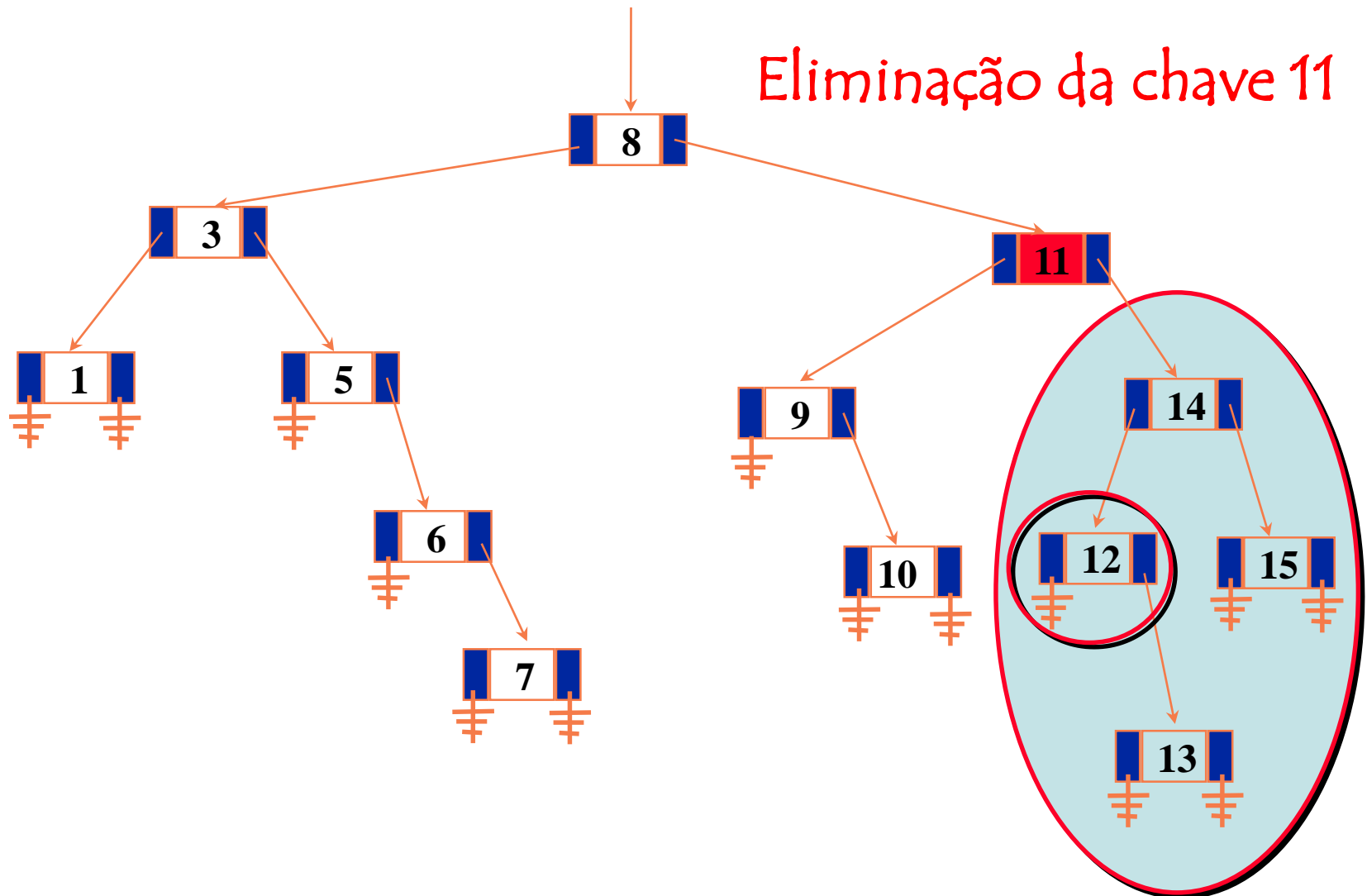


A estratégia geral (Mark Allen Weiss) é:

**Substituir a chave retirada pela menor chave da sub-árvore direita**

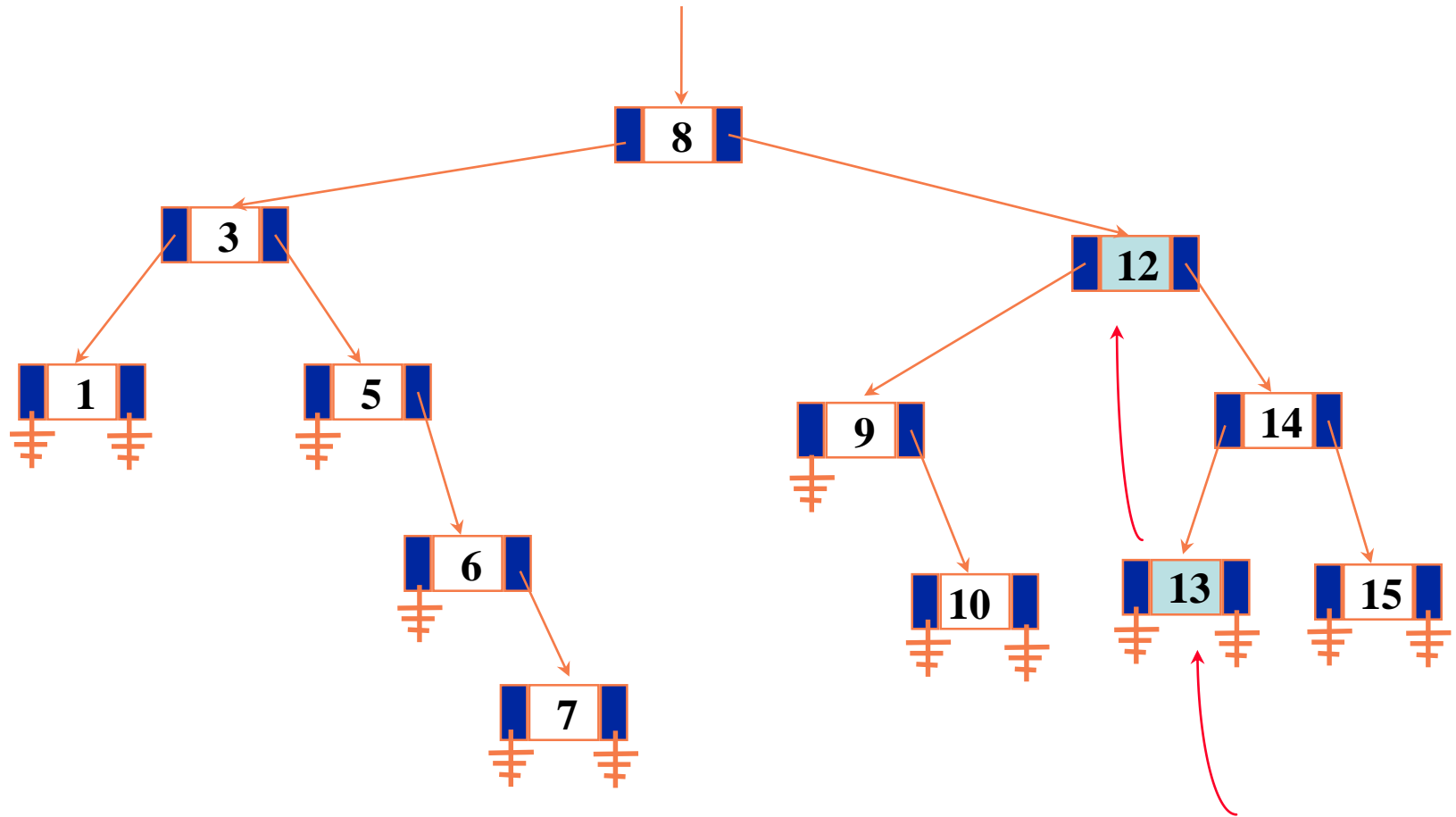


# Caso 3 – Não tem duas sub-árvores



**Substituir a chave retirada pela menor chave da sub-árvore direita**

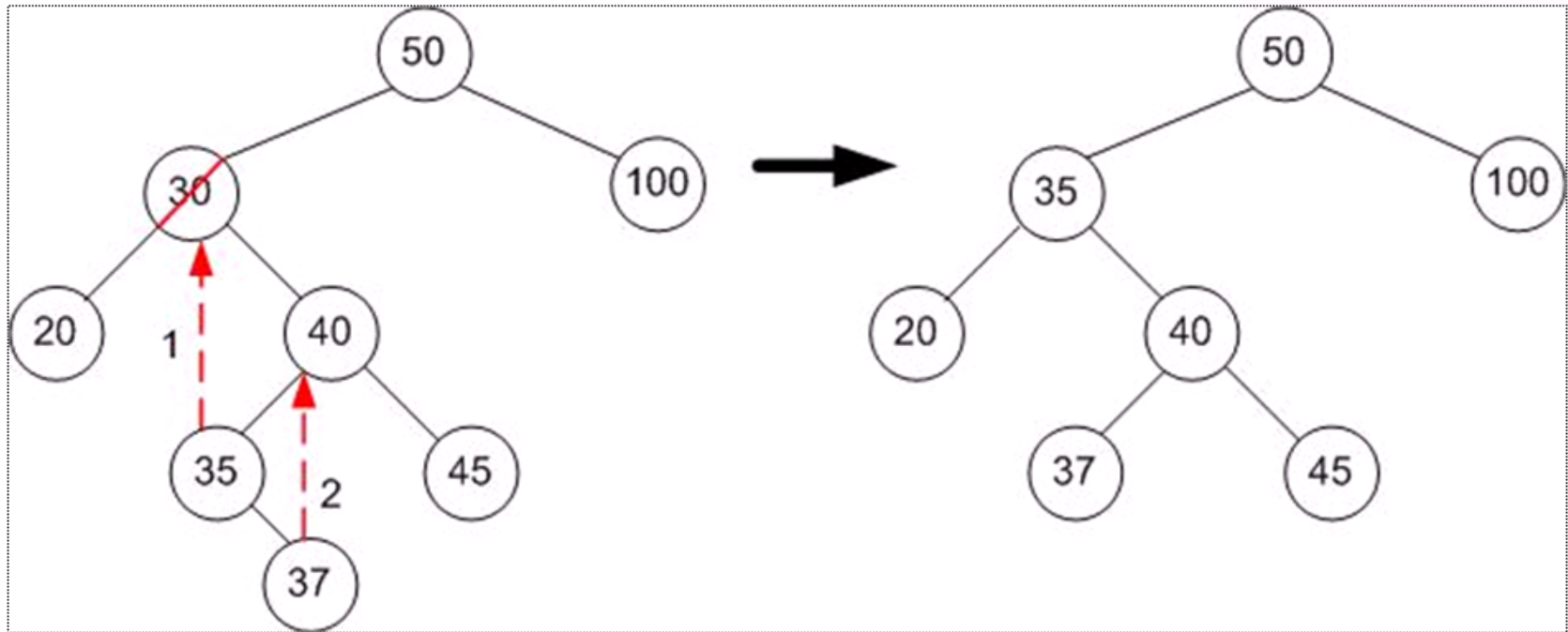
# Caso 3 – Nó tem duas sub-árvores



Eliminação da chave 11



# Caso 3 – Outro exemplo



Eliminação da chave 30

