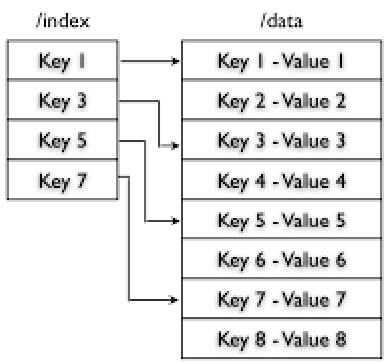
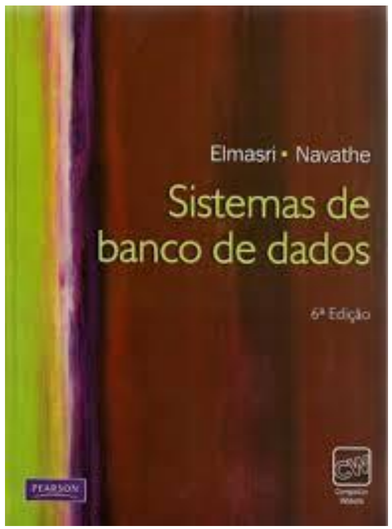
Unidade 10 – Estruturas de Indexação em Banco de Dados
Parte 2

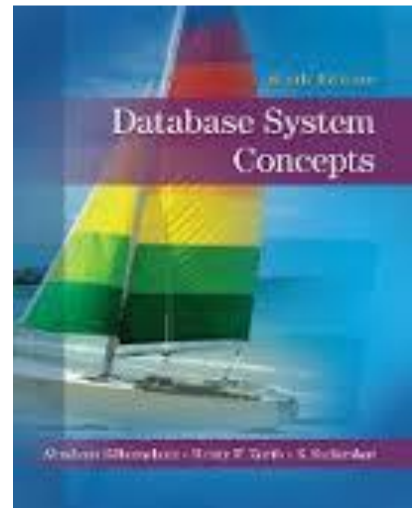


Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

Bibliografia



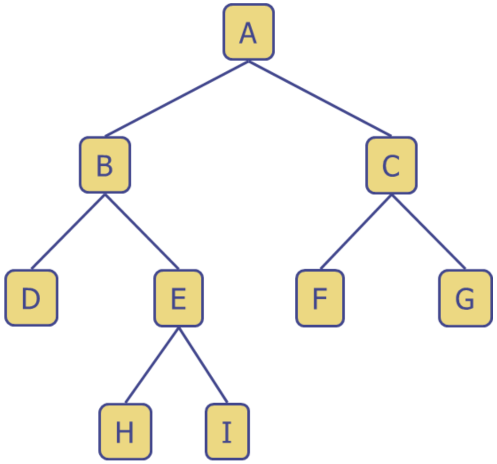
Sistemas de Banco de Dados
Elmasri / Navathe 6ª edição



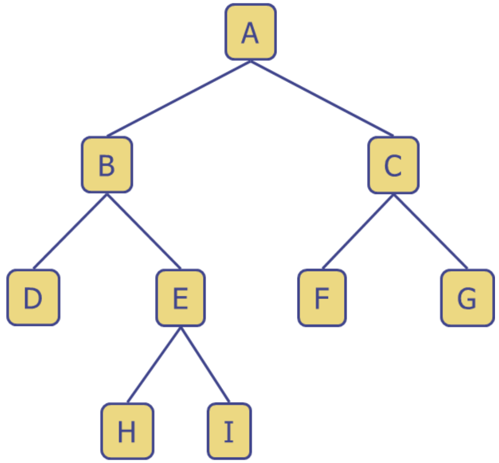
Sistema de Banco de Dados
Korth, Silberschatz – Sixth Editon

Árvore Binária

- É uma árvore ordenada com as seguintes propriedades:
  - Todo nó tem no máximo 2 filhos.
  - Cada filho é rotulado como sendo **filho a esquerda** ou **filho a direita**.
  - Um filho a esquerda precede o filho a direita na ordenação dos filhos de um nó.
  - Assim, filhos formam um par ordenado.

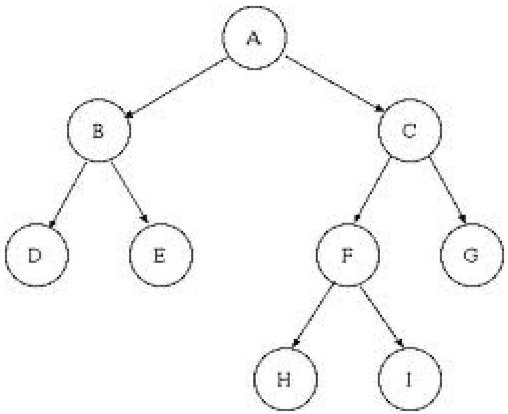


Árvore Binária



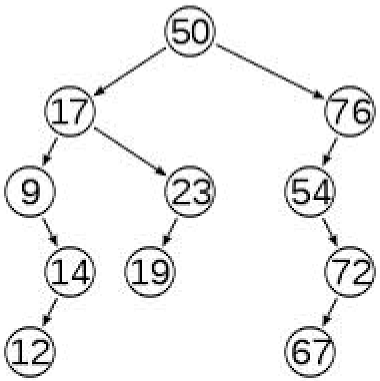
Árvore Binária própria

- Uma árvore binária é **própria** se cada nó tem 0 ou 2 filhos.
- Em uma árvore binária **própria** cada nó interno tem exatamente 2 filhos.



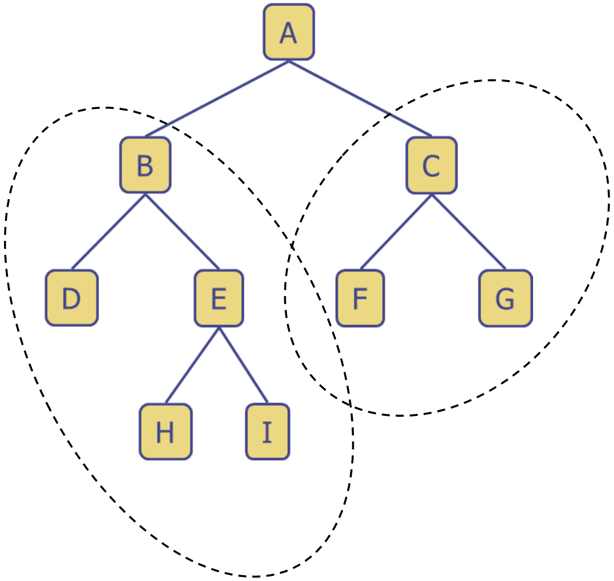
Árvore Binária Imprópria

- Uma árvore é **imprópria** se não for própria, ou seja, a árvore tem pelo menos um nó com apenas um filho.



Definição Recursiva

- Uma **árvore binária** é:
  - Uma árvore que consiste de apenas um nó, ou
  - Uma árvore cuja raiz tem um par ordenado de filhos, onde cada qual é uma árvore binária.



ADT – Árvore Binária

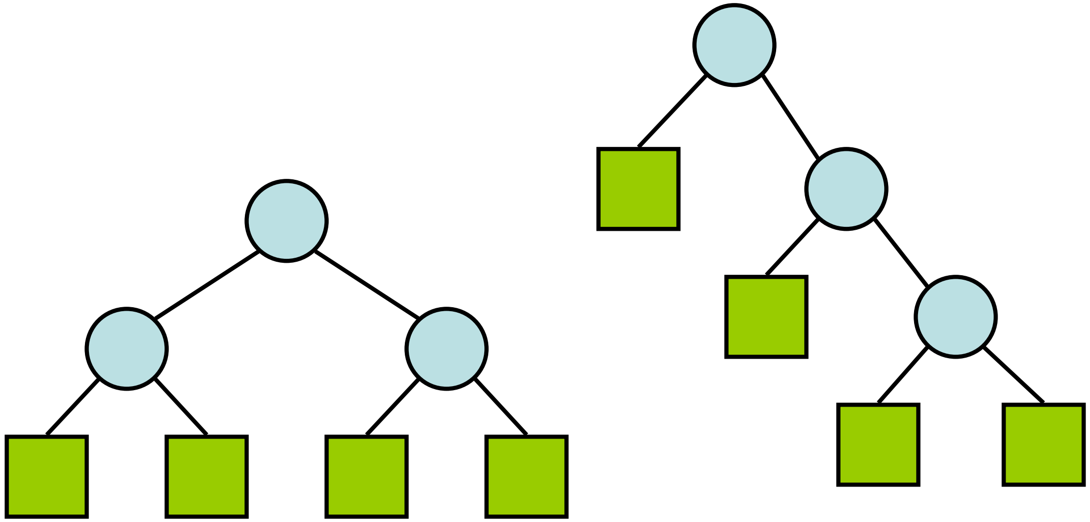
- A árvore binária estende a ADT Árvore, isto é , herda todos os métodos vistos no capítulo anterior (árvores genéricas).

- Adicionalmente, suporta os seguintes métodos:

```
left(): retorna o filho esquerdo de um nó
right(): retorna o filho direito de um nó
hasLeft(): testa se o nó tem filho a esquerda
hasRight(): testa se o nó tem filho a direita
inorder(): percurso inorder
```

Árvore Binária Própria - Propriedades

- Notação
  - n número de nós
  - e número de nós externos
  - i número de nós internos
  - h altura
  - b número de arestas

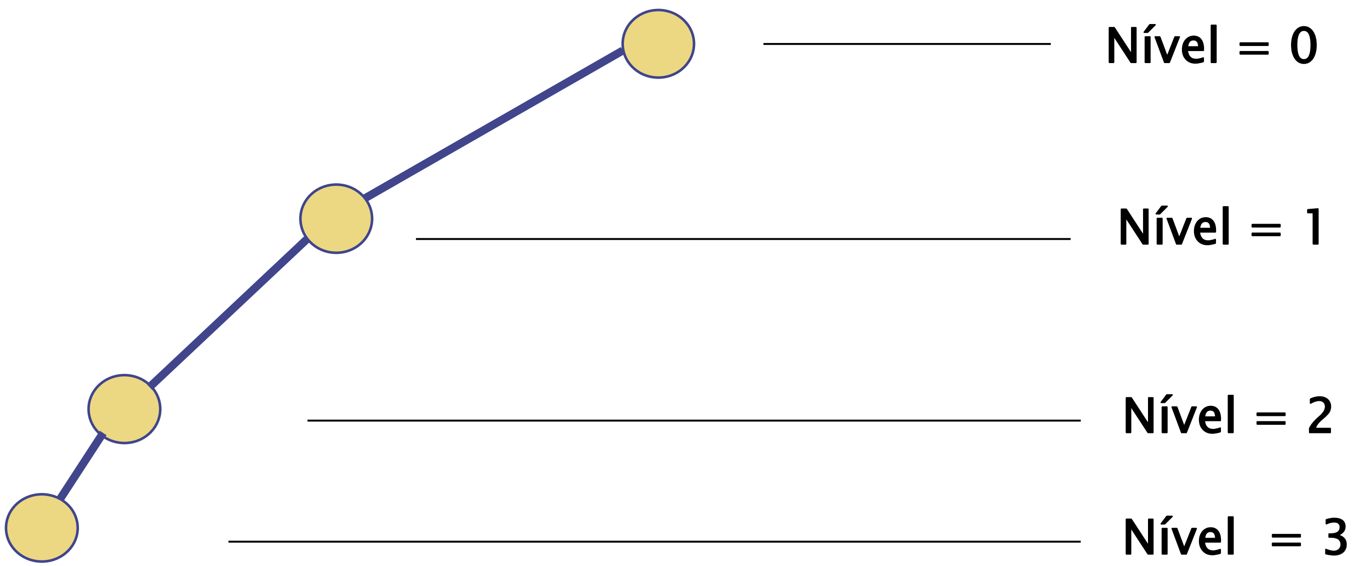


- Propriedades
  - $\mathbf{e} = \mathbf{i} + 1$
  - $\mathbf{n} = 2\mathbf{e} - 1$
  - $\mathbf{h} \leq \mathbf{i}$
  - $\mathbf{h} \leq (\mathbf{n} - 1)/2$
  - $\mathbf{e} \leq 2^{\mathbf{h}}$
  - $\mathbf{h} \geq \log_2 \mathbf{e}$
  - $\mathbf{h} \geq \log_2 (\mathbf{n} + 1) - 1$

Número mínimo de nós

- O número mínimo de nós em uma árvore binária de altura **h**, é **n ≥ h+1**.

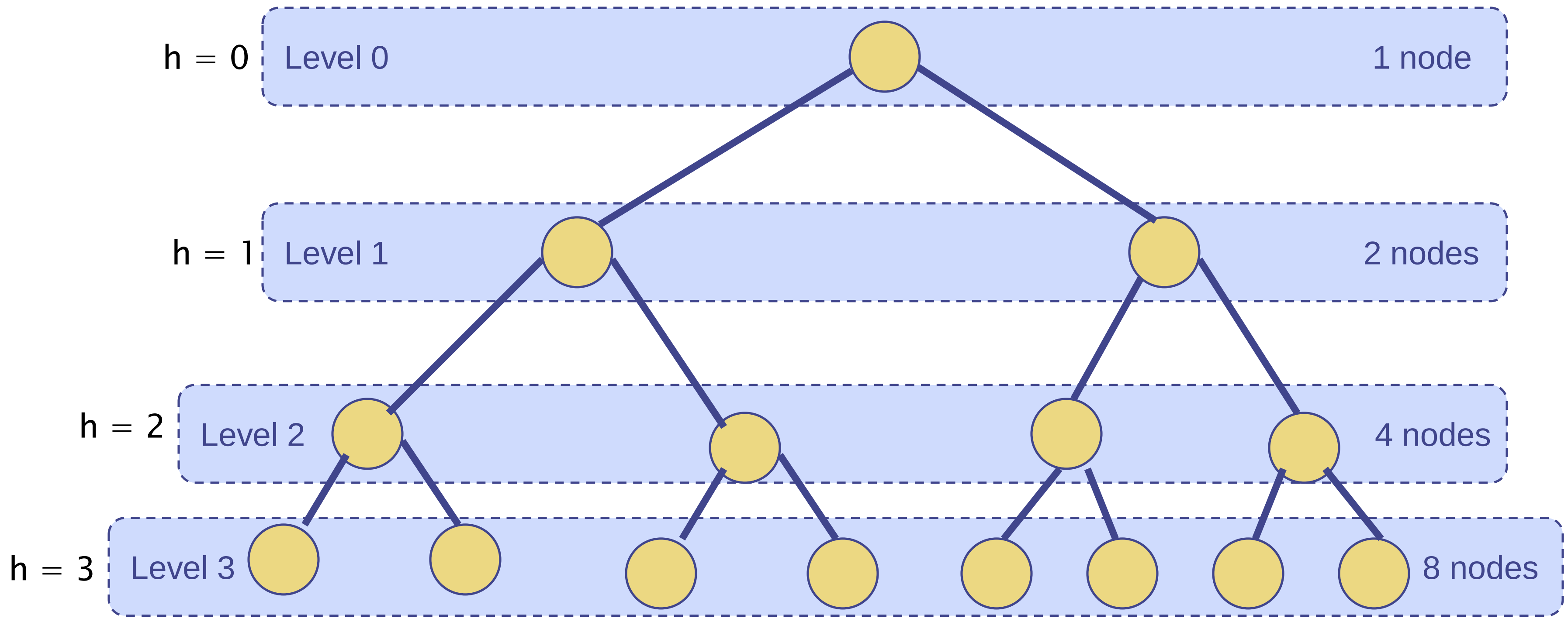
- Ao menos um nó em cada um dos níveis d.



Número mínimo de nós é ***h+1***

- **Altura de um nó: Tamanho do caminho de n até seu mais profundo descendente.**

Máximo número de nós

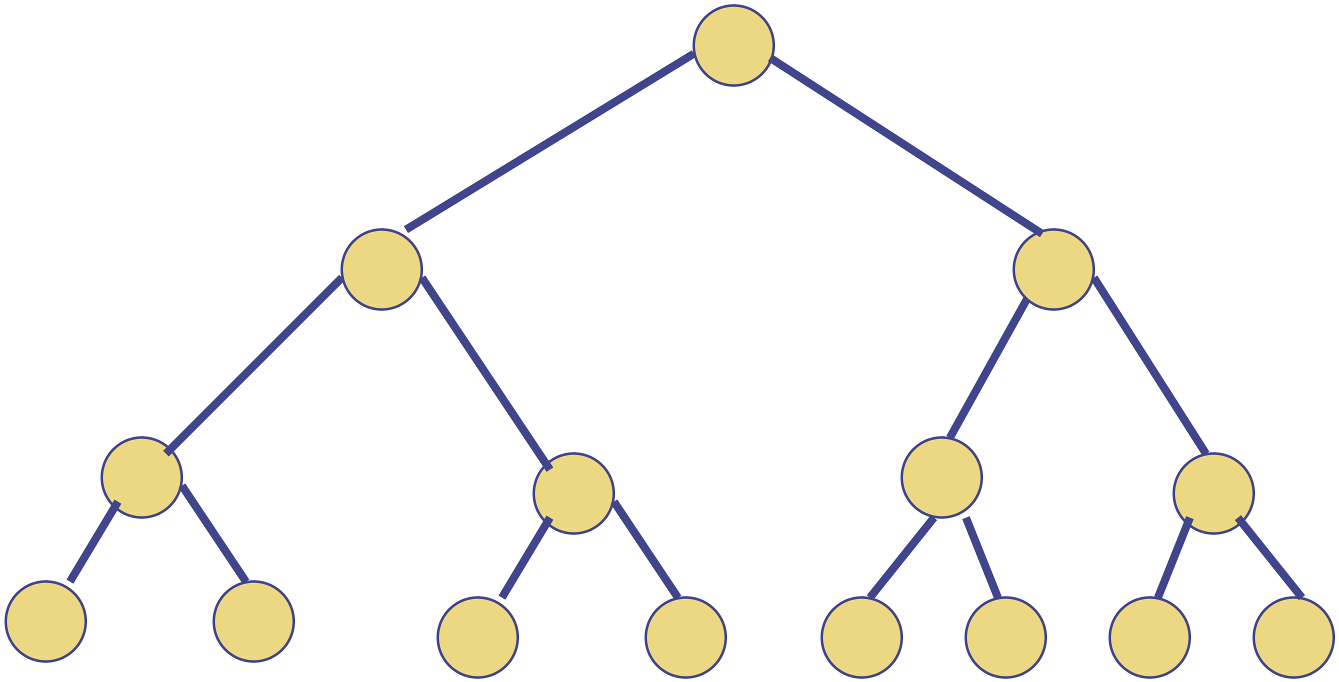


Máximo número de nós $= 1 + 2 + 4 + 8 + \ldots + 2^h$

$$n \leq 2^{h+1} - 1$$

Árvore Binária Completa (Full)

Uma árvore binária completa de altura *h* tem $2^{h+1} - 1$ nós.



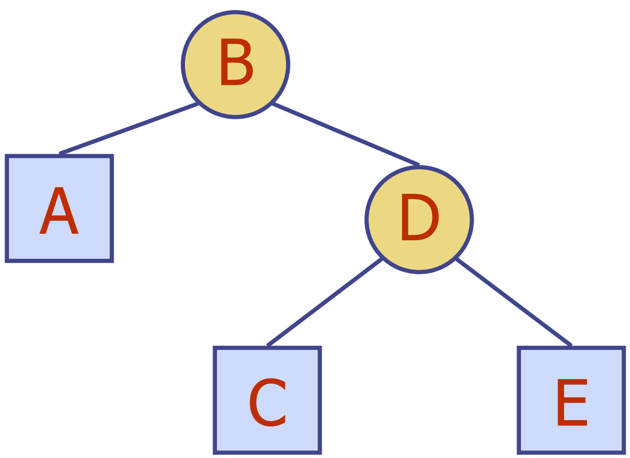
Árvore binária completa de altura 3

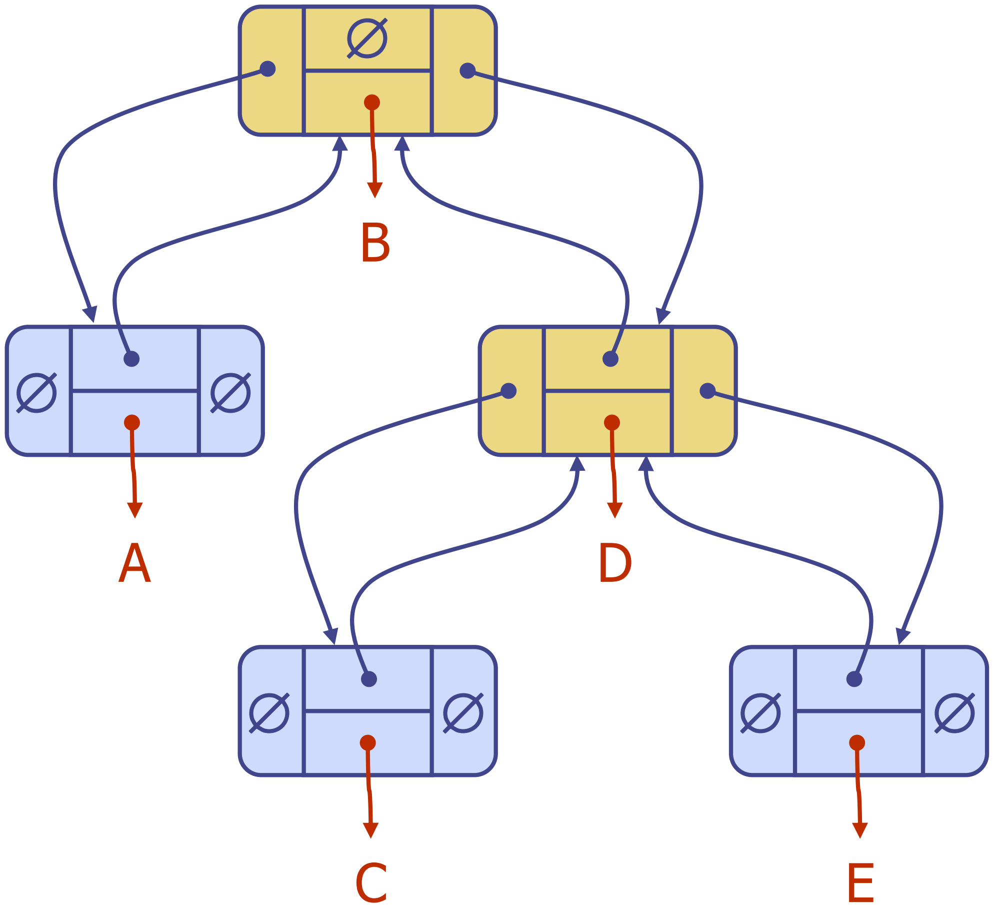
Representação de árvores binárias

1. **Linked Structure**
2. **Array List**

Representação por lista ligada

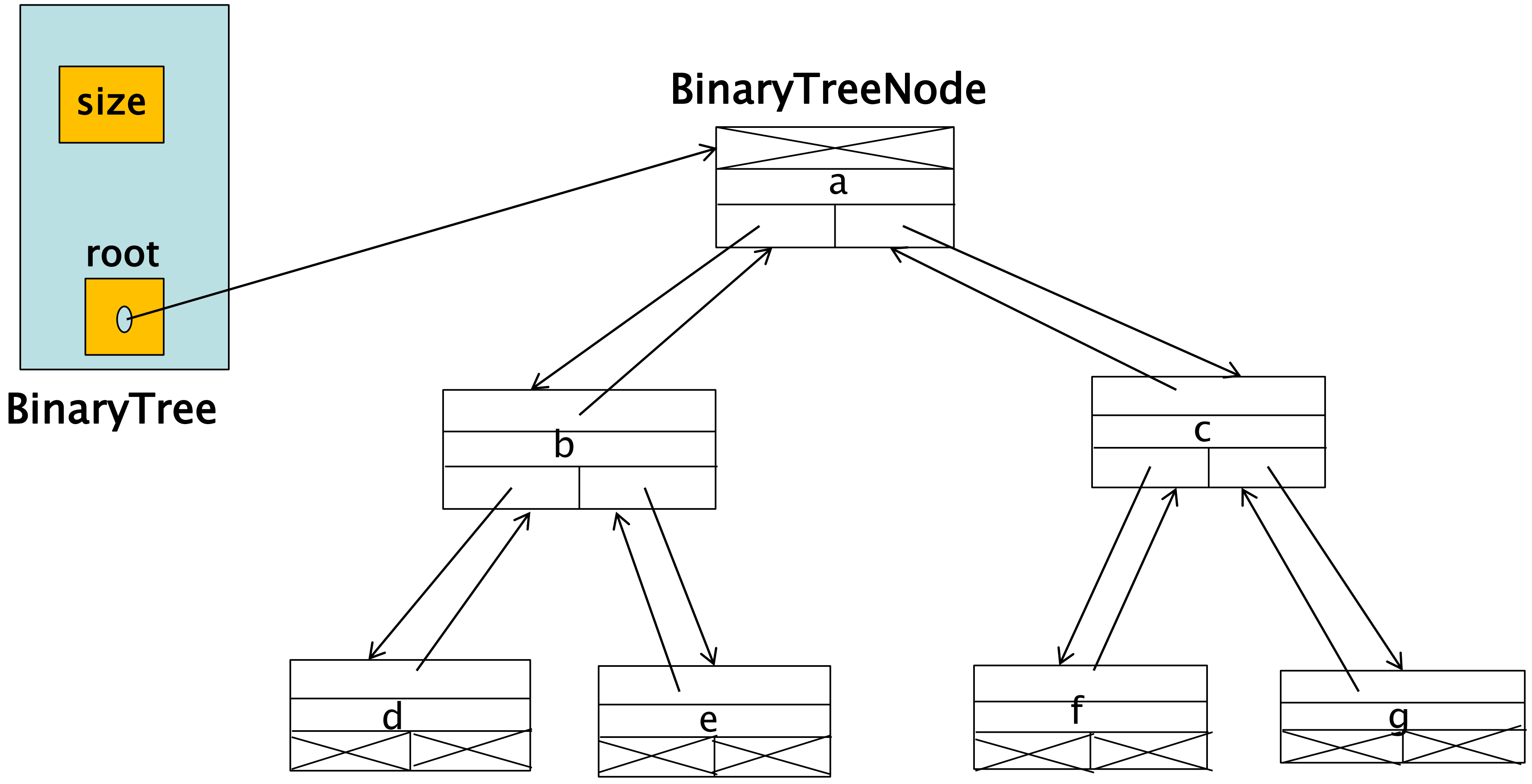
- Um nó é representado por um objeto armazenando:
  - Elemento
  - Nó pai
  - Nó Left child
  - Nó Right child





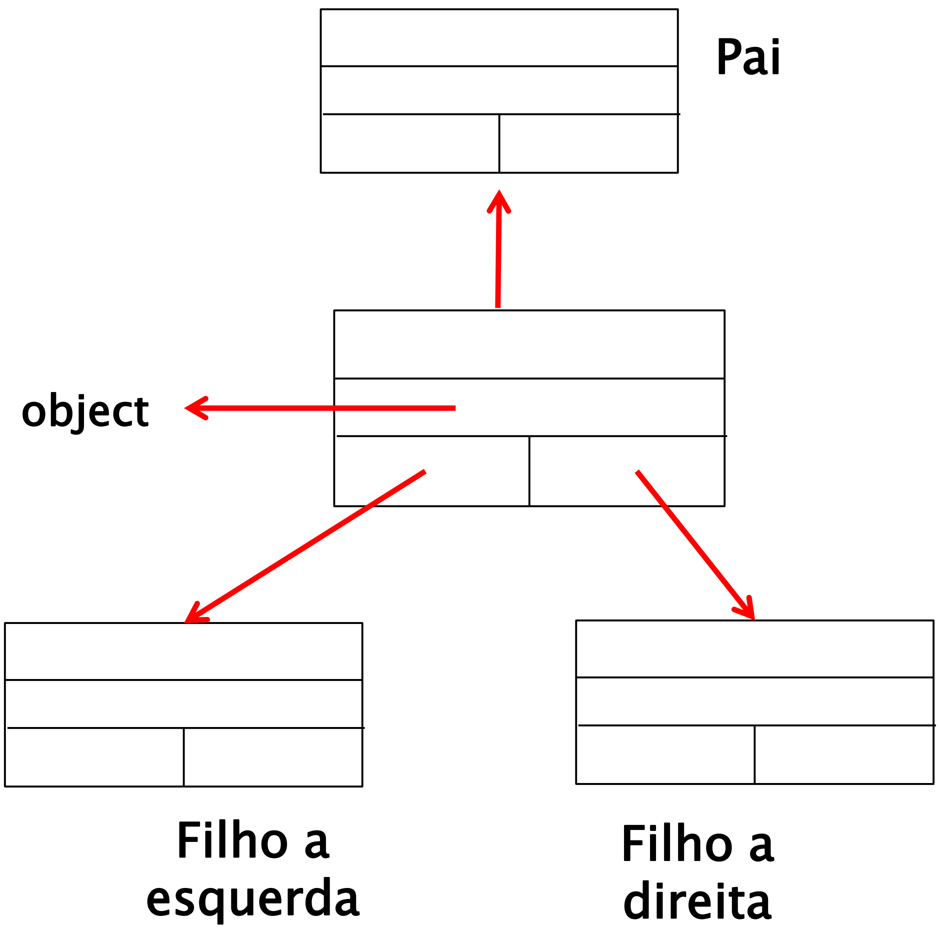
Representação por lista ligada

size -> #nós da árvore



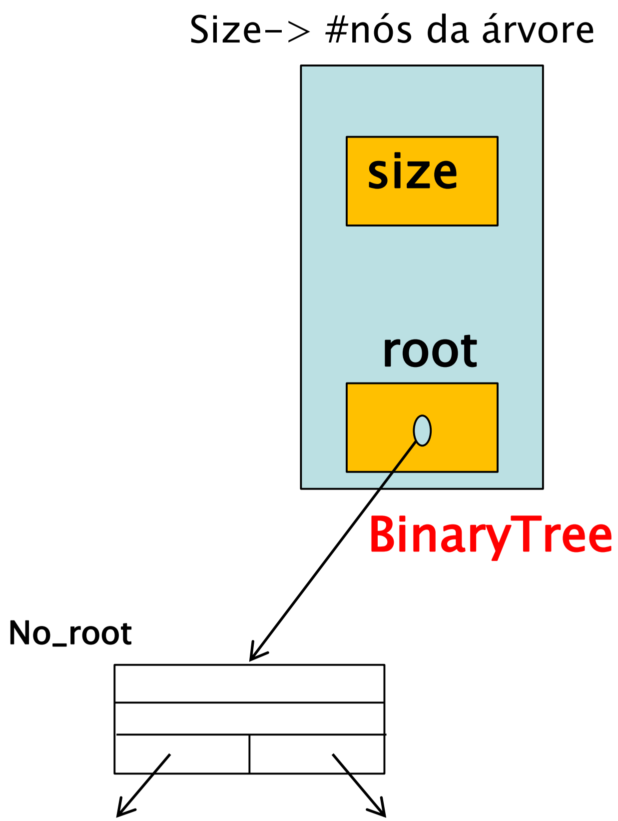
Representando nó da Árvore

- Cada nó tem quatro referências: item, pai, filho a esquerda e filho a direita.




```
Class BinaryTreeNode {

        Object item;
        BinaryTreeNode parent;
        BinaryTreeNode left;
        BinaryTreeNode right;

}
```

Representando a Árvore

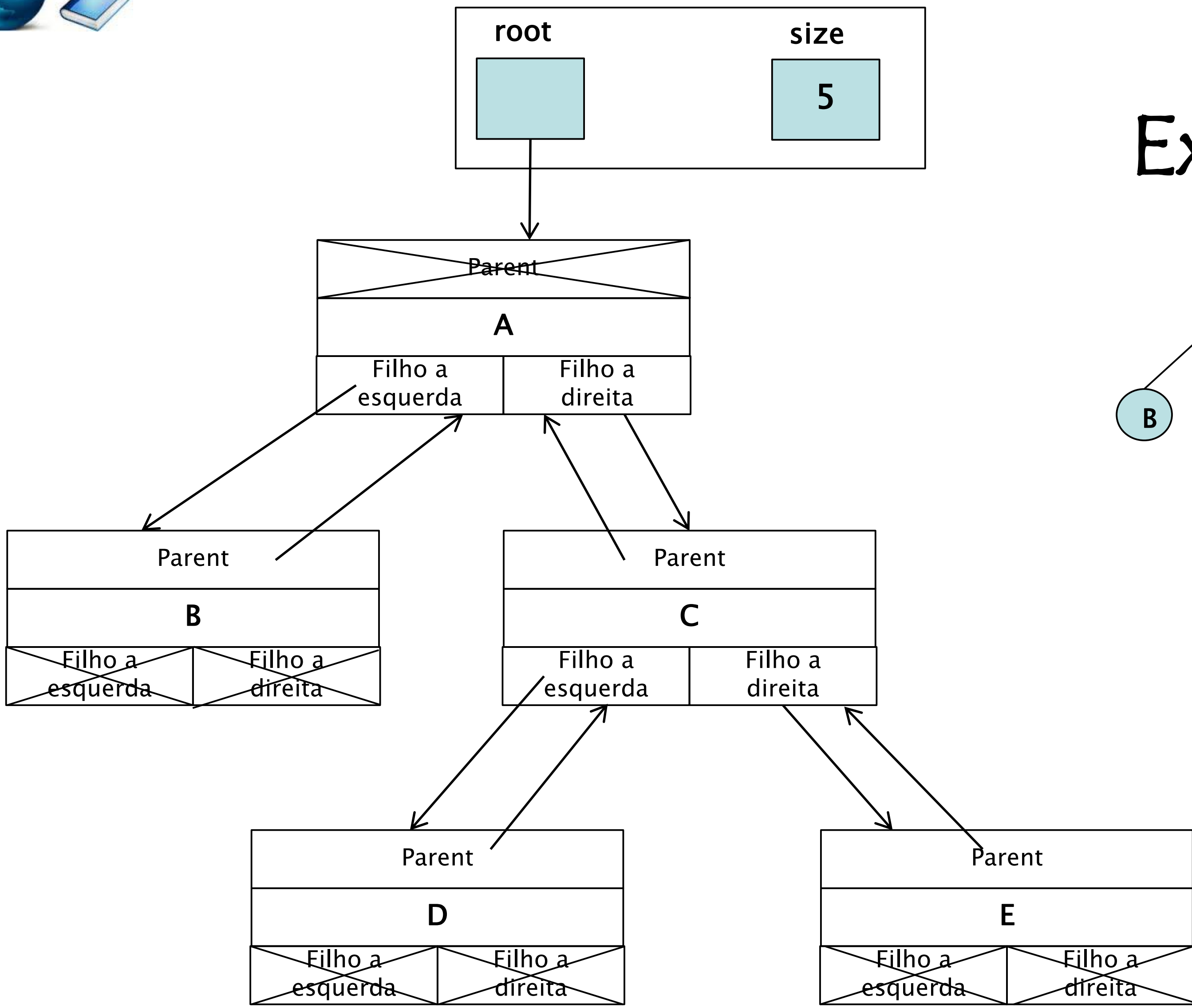


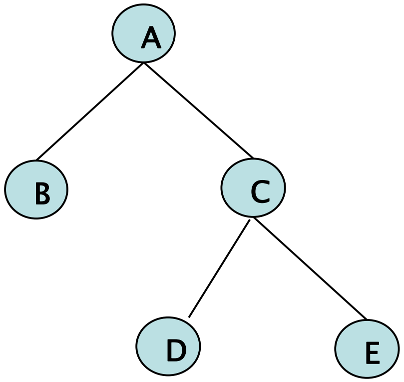

```
Class BinaryTree {

        BinaryTreeNode root;
        int size;

}
```

Representando Nó da árvore

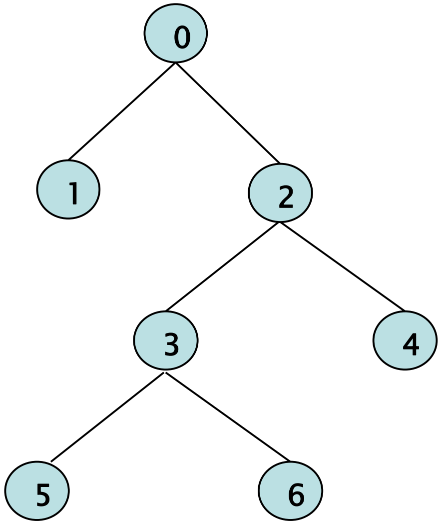
<table>
<tr><td colspan="2">Parent</td></tr>
<tr><td colspan="2">Item</td></tr>
<tr><td>Filho a esquerda</td><td>Filho a direita</td></tr>
</table>

Exemplo


root
size
5
Parent
A
Filho a esquerda
Filho a direita
Parent
B
Filho a esquerda
Filho a direita
Parent
C
Filho a esquerda
Filho a direita
Parent
D
Filho a esquerda
Filho a direita
Parent
E
Filho a esquerda
Filho a direita


A
B
C
D
E

Exemplo



```
public class BinaryTree {

        BinaryTreeNode root;
        int size;

        public BinaryTree() {

                this.root = null;
                this.size = 0;
        }

        public void insert_root(int valor) {

                BinaryTreeNode node = new BinaryTreeNode(valor);
                this.root = node;
                this.size = 1;

        }
```

```java
   public BinaryTreeNode ret_Root() {
         return (this.root);
   }

   public int size() {
         return this.size;
   }

   public boolean isEmpty() {

         if (this.size == 0 )
                 return true;
         else return false;
   }
}
```

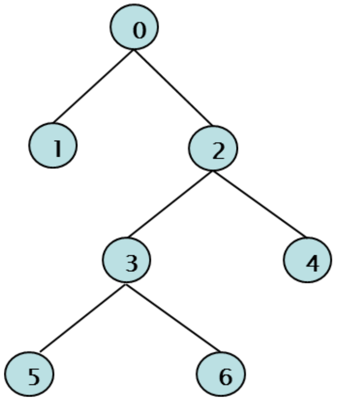
```java
package maua;

        public class BinaryTreeNode {

                int item;
                BinaryTreeNode parent;
                BinaryTreeNode left;
                BinaryTreeNode right;


        public BinaryTreeNode(int item) {

                this.item = item;
                this.parent = null;
                this.left = null;
                this.right = null;

}
```

```java
public BinaryTreeNode left() {
        if (this.left == null)
                return null;
        else return this.left ;

}

public boolean  isLeft() {
        if (this.left == null)
                return false;
        else return true ;

}

public BinaryTreeNode right() {
        if (this.right == null)
                return null;
        else return this.right ;

}
```

```
public boolean  isRight() {
        if (this.right == null)
                return false;
        else return true ;

}

public void binaryPreorder() {

        System.out.println(this.item);
        if (this.isLeft())
                this.left.binaryPreorder();
        if (this.isRight())
                this.right.binaryPreorder();
}
```

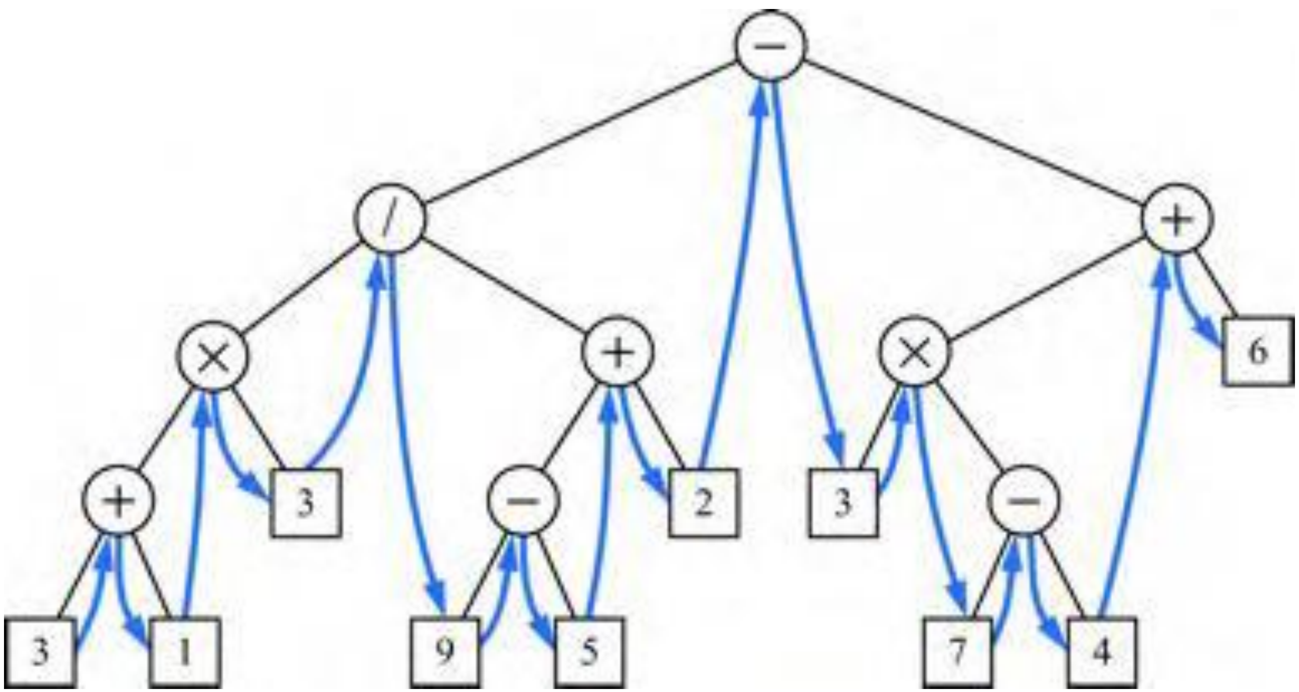
```java
  public void binaryPostorder() {


        if (this.isLeft())
                this.left.binaryPostorder();
        if (this.isRight())
                this.right.binaryPostorder();
        System.out.println(this.item);

  }

  public void binaryInorder() {


        if (this.isLeft())
                this.left.binaryInorder();
        System.out.println(this.item);
        if (this.isRight())
                this.right.binaryInorder();
  }

}
```

```
package maua;

        public class Teste_BinaryTreeNode {

        public static void main(String[] args ) {

                BinaryTree x = new BinaryTree();
                x.insert_root(0);

                BinaryTreeNode no_1 = new BinaryTreeNode(1) ;
                BinaryTreeNode no_2 = new BinaryTreeNode(2) ;
                BinaryTreeNode no_3 = new BinaryTreeNode(3) ;
                BinaryTreeNode no_4 = new BinaryTreeNode(4) ;
                BinaryTreeNode no_5 = new BinaryTreeNode(5) ;
                BinaryTreeNode no_6 = new BinaryTreeNode(6) ;

                x.root.left = no_1;
                x.root.right = no_2;
                no_2.left = no_3;
                no_2.right = no_4;
                no_3.left = no_5;
                no_3.right = no_6;

                x.root.binaryPreorder();
                x.root.binaryPostorder();
                x.root.binaryInorder();

        }

}
```

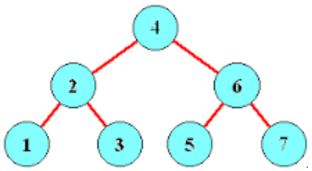


Travessia Inorder

- Representa um método de travessia adicional válido para árvores binárias.
- Nesta travessia, visitamos um nó entre as chamadas recursivas das subárvores esquerda e direita.

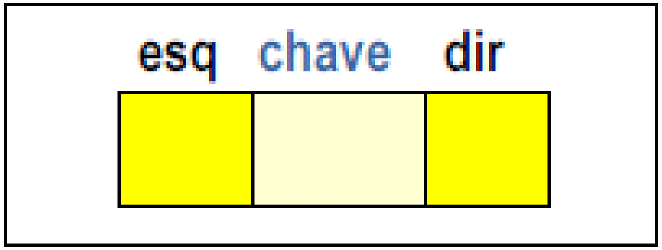


Árvore Binária de Pesquisa



- 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**.

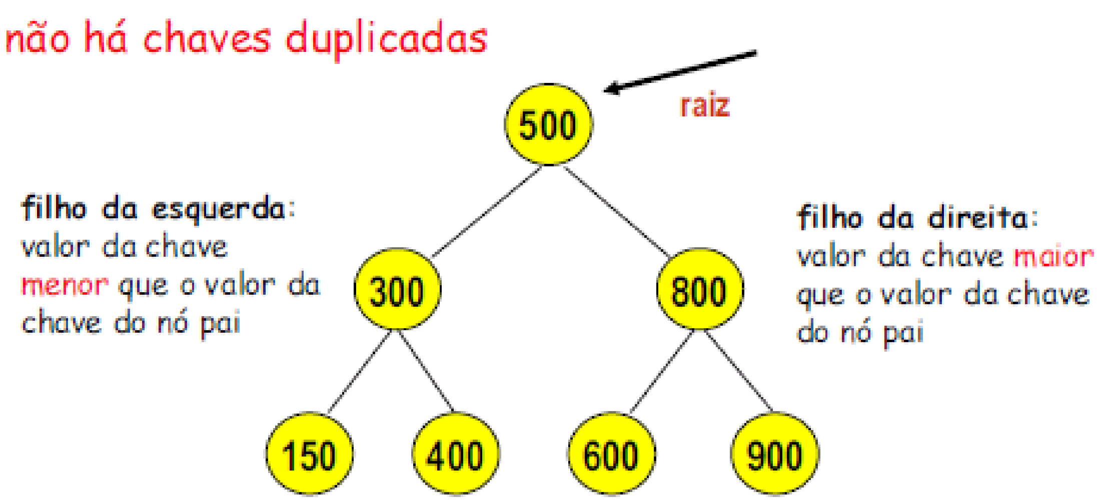


Árvore Binária de pesquisa

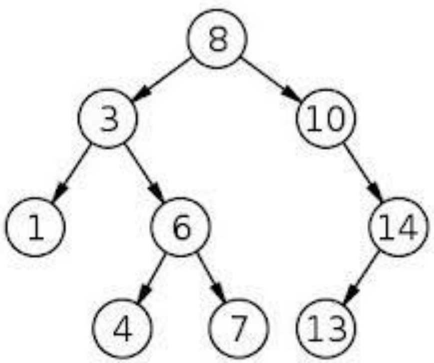
- **Definição de Niklaus Wirth:**

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.



Inserção em Árvores Binárias de Pesquisa

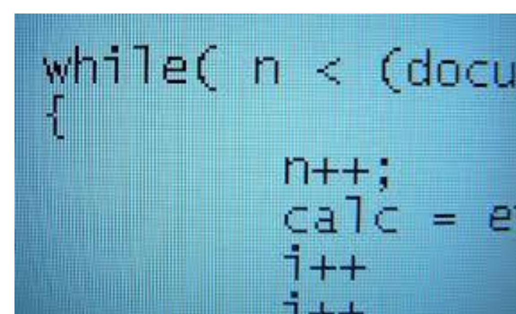


Carga da Árvore Binária de pesquisa

```
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.



Inserção em uma árvore de busca binária

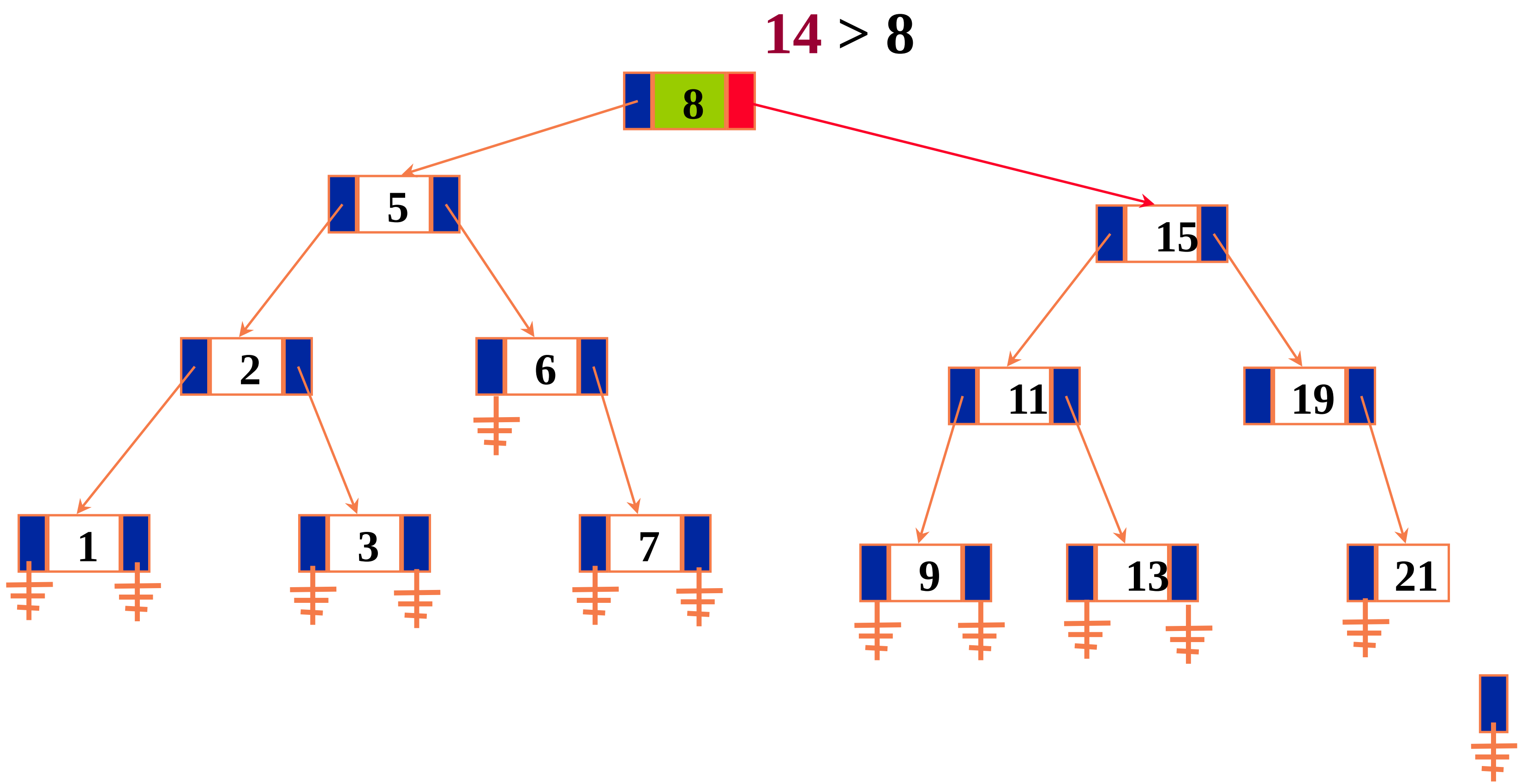
Lembrando que ...

- 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.

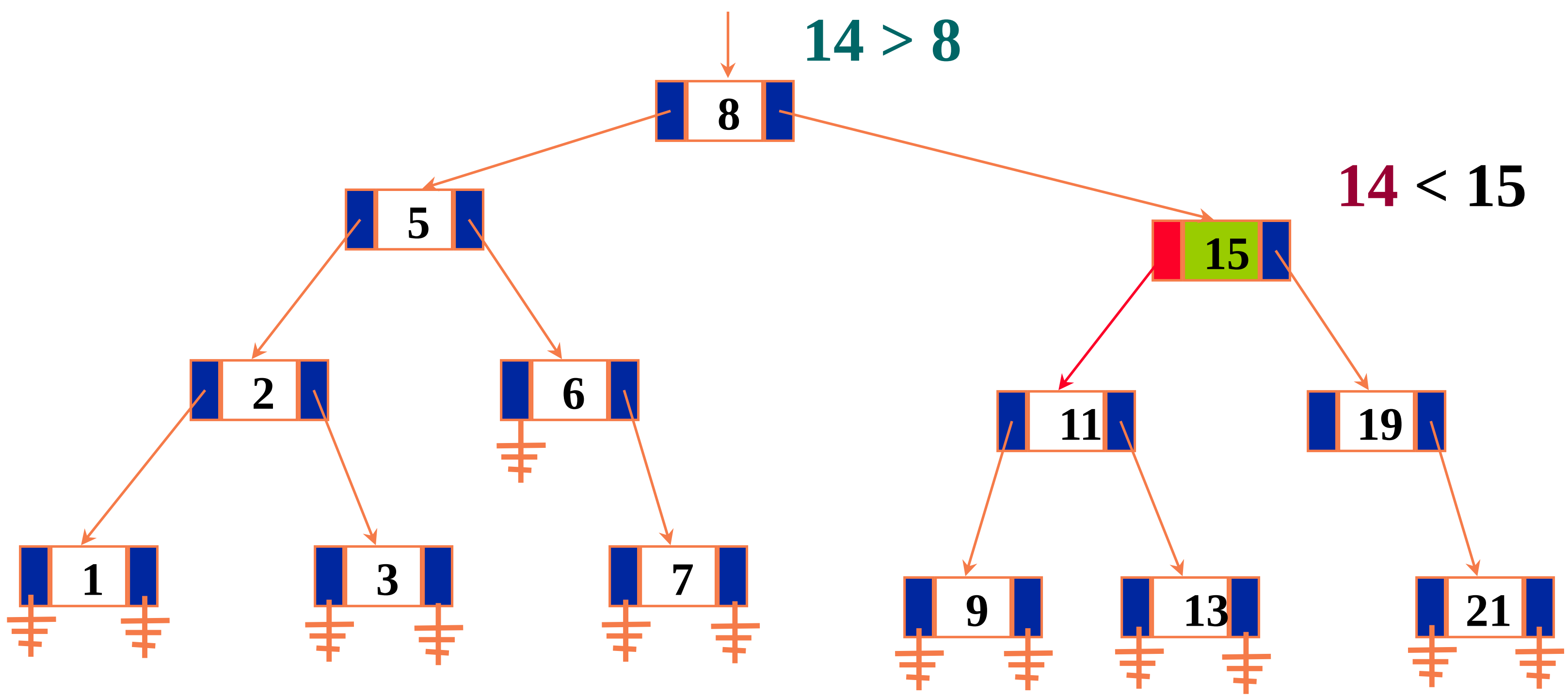
Princípio Básico

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

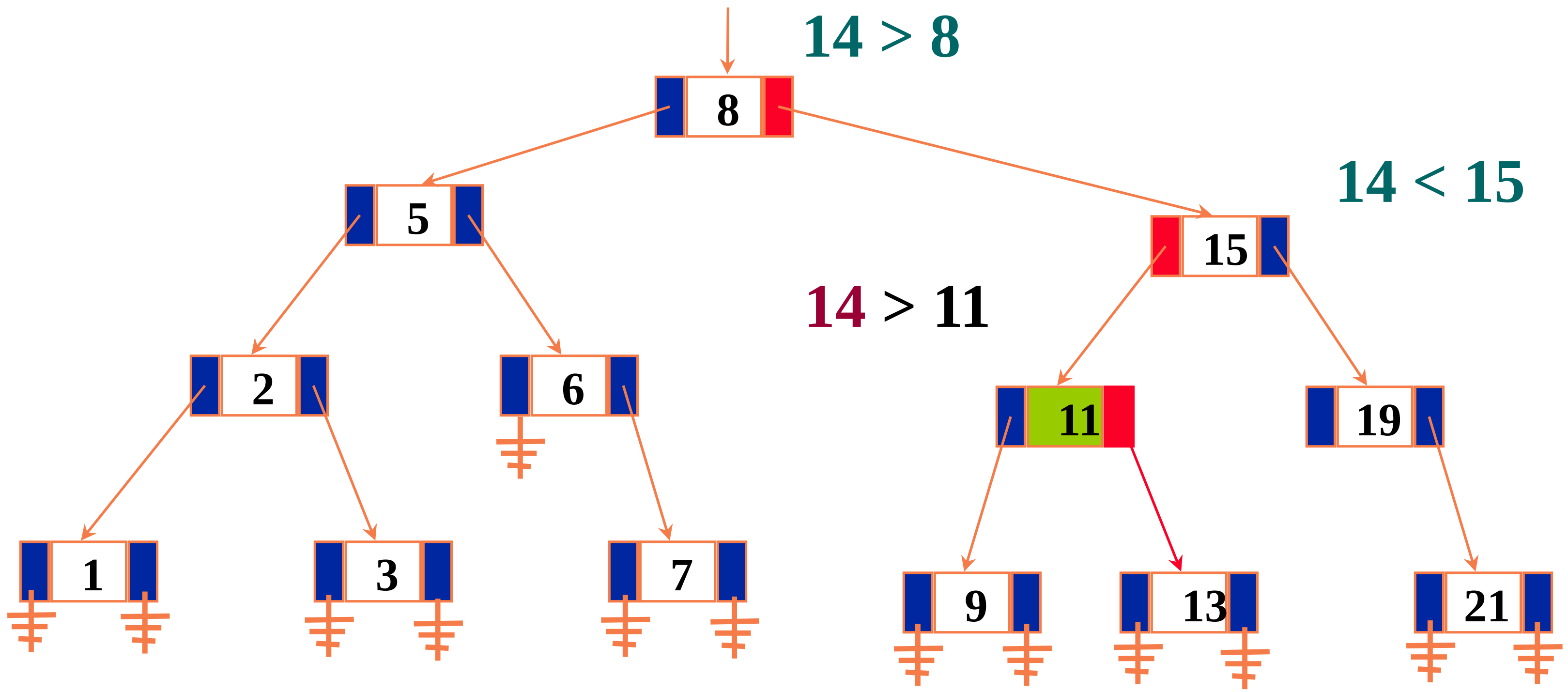
Exemplo – Inserção de elemento com chave 14



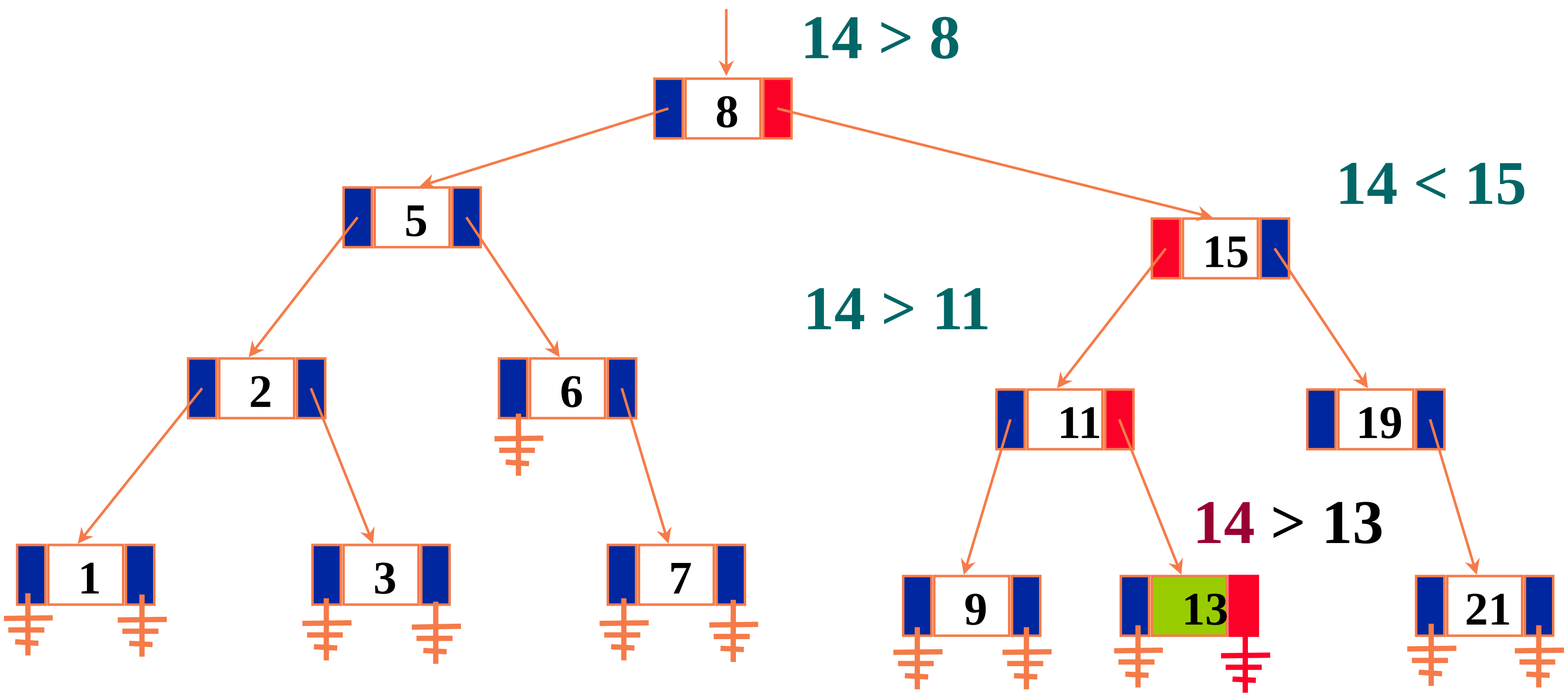
Exemplo – Inserção de elemento com chave 14



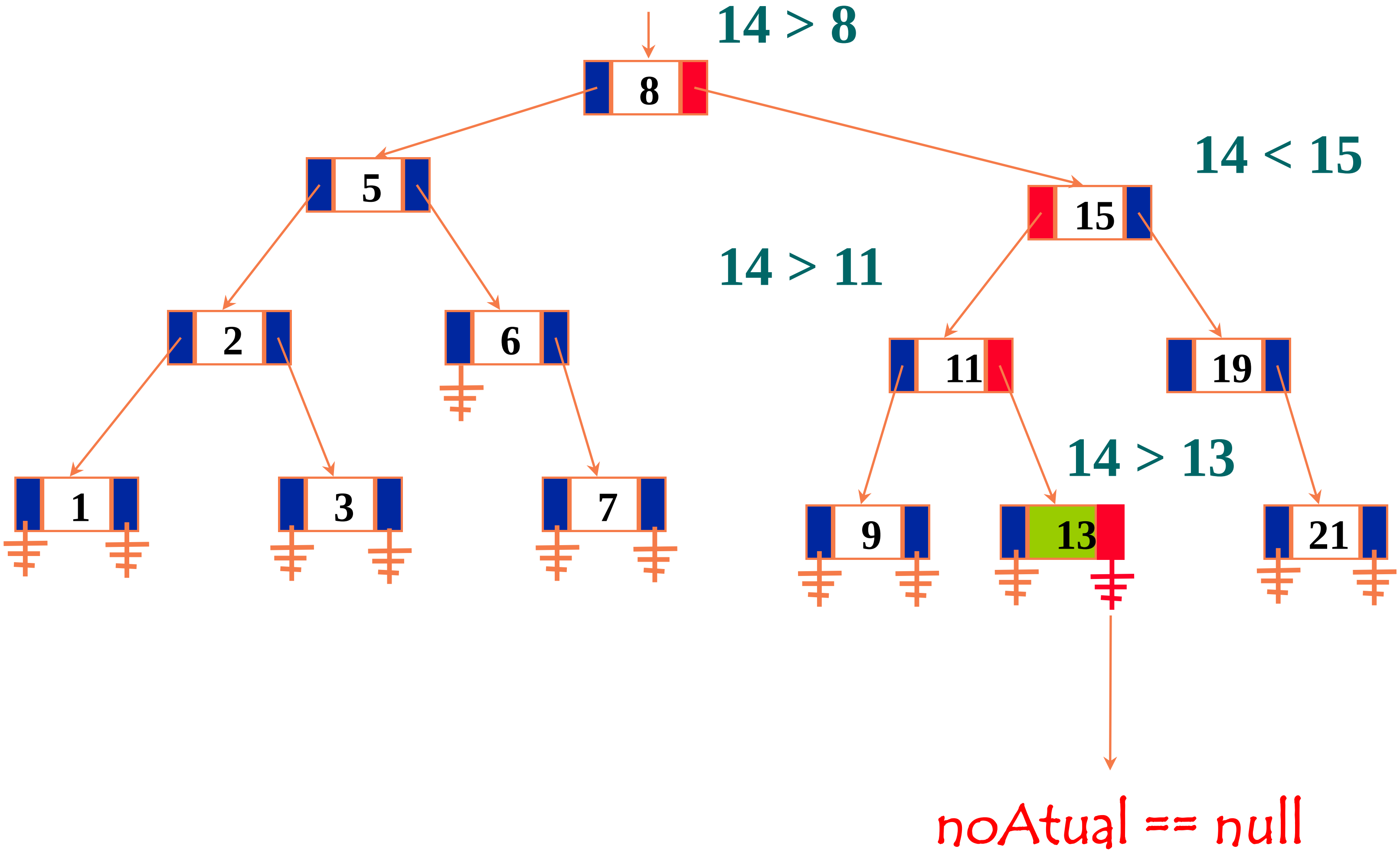
Exemplo – Inserção de elemento com chave 14



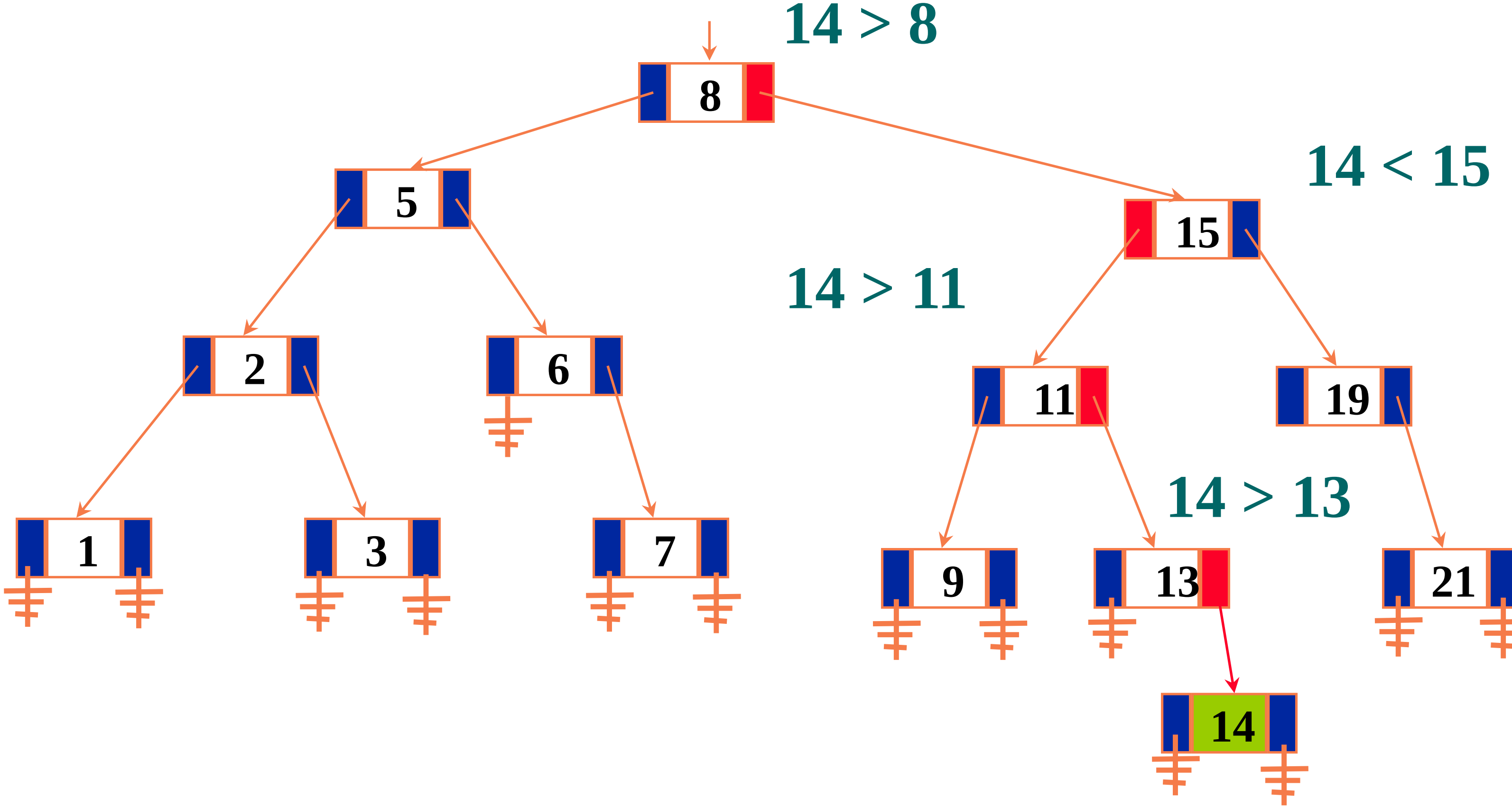
Exemplo – Inserção de elemento com chave 14



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

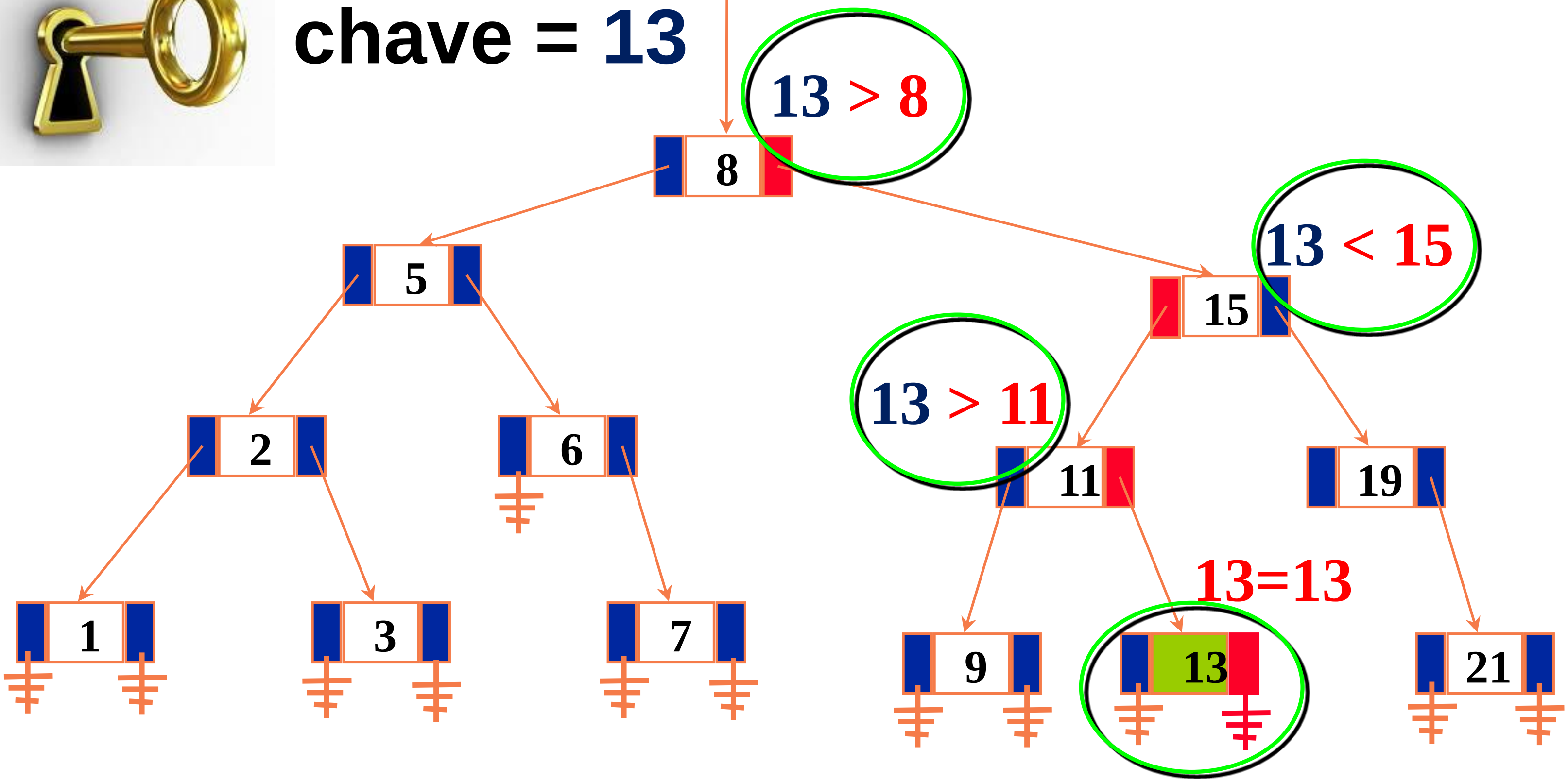
- Em uma árvore binária é possível encontrar qualquer chave existente **X** atravessando-se árvore:
  - sempre **à esquerda** se **X** for **menor** que a chave do nó visitado e
  - sempre **à direita** toda vez que for **maior**.
  - A escolha da direção de busca só depende de **X** e da chave que o nó atual possui.

- A busca de um elemento em uma árvore balanceada com **n** elementos toma tempo médio menor que $\log_2(\mathbf{n})$, tendo a busca então $O(\log_2 \mathbf{n})$.

n	log2 (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



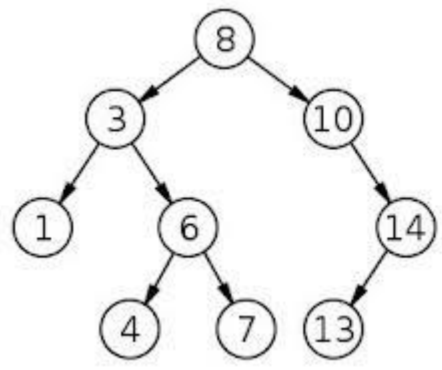
Algoritmo iterativo de search em árvore de busca

```
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



QualitSys

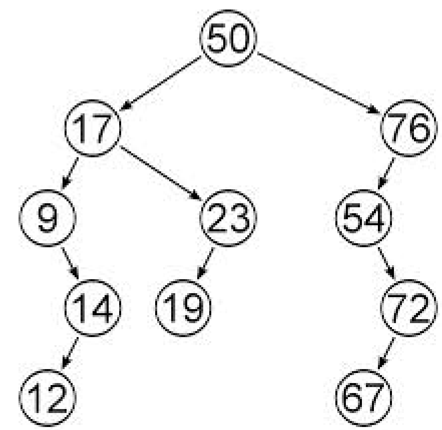
Eliminação em Árvores Binárias de Busca

✓ A eliminação é mais complexa do que a inserção.

✓ A razão básica é que a característica organizacional da árvore não deve ser alterada.

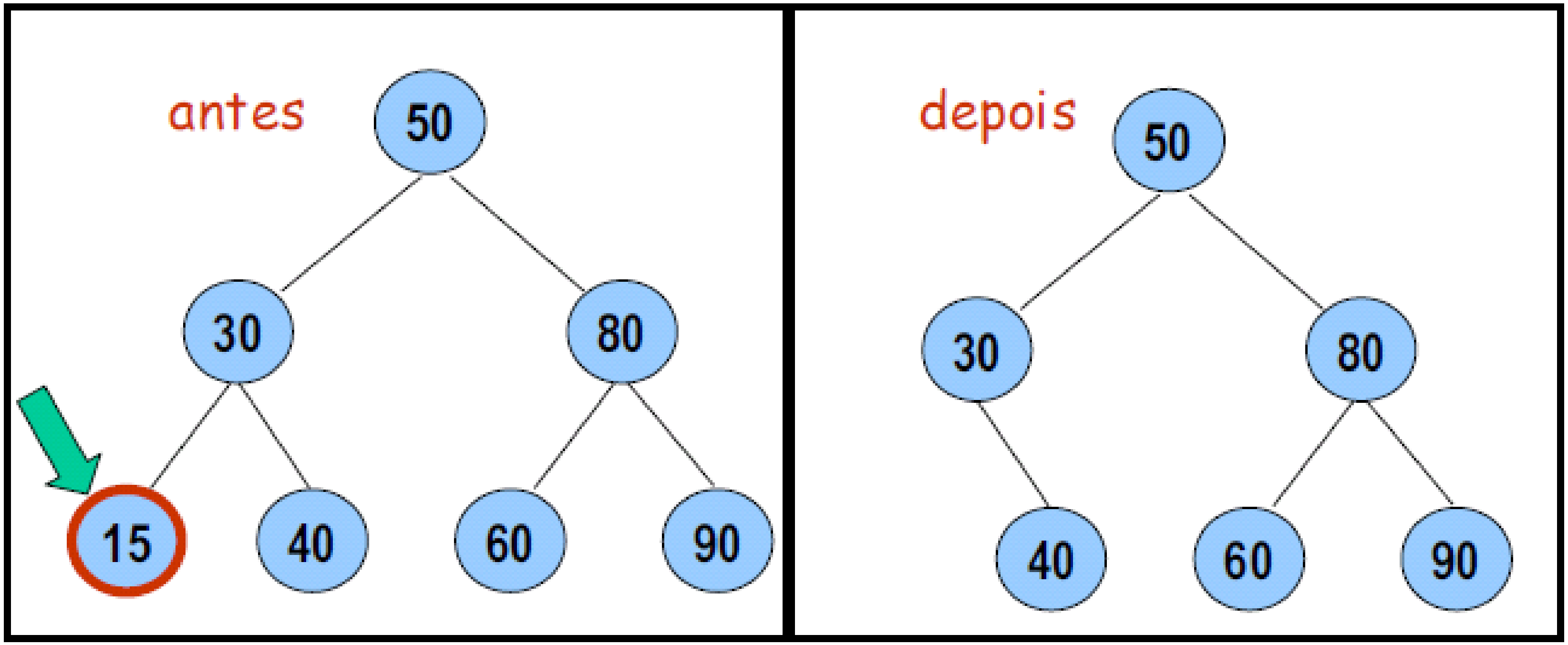
- A sub-árvore **direita** de um nó deve possuir chaves **maiores** que a do pai.
- A sub-árvore **esquerda** de um nó deve possuir chaves **menores** que a do pai.

✓ Para garantir isto, o algoritmo deve “**remanejar**” os nós.



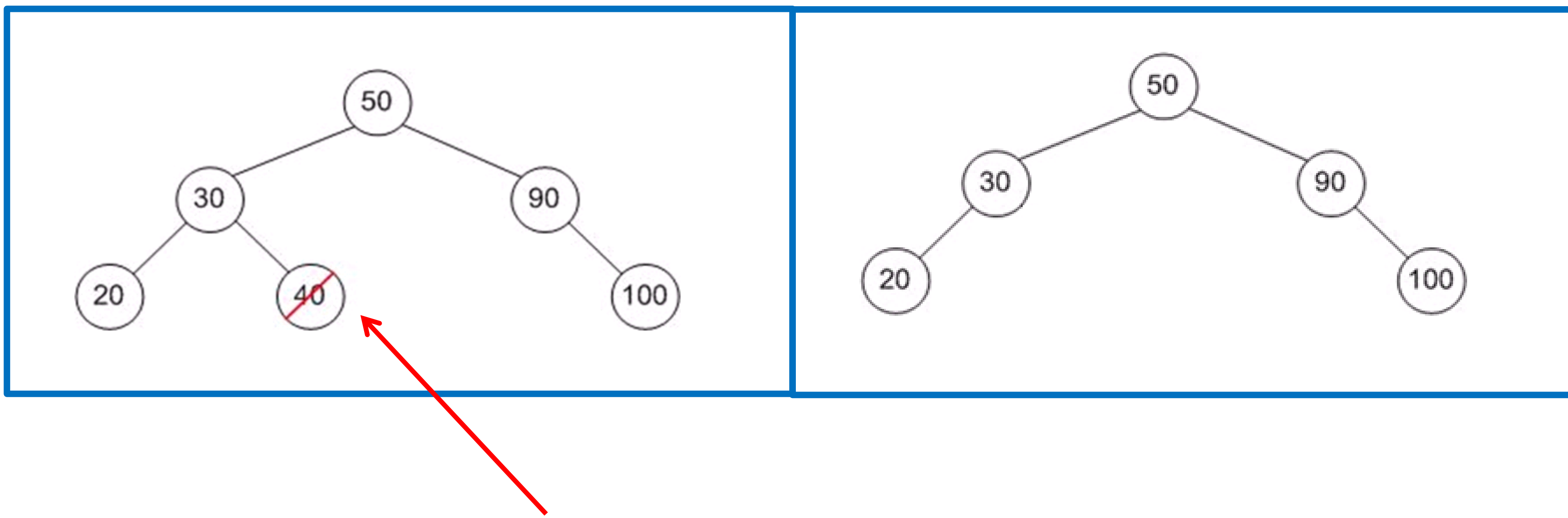
Caso 1 – Remoção de nó folha

Caso mais simples, basta retirá-lo da árvore



Eliminação da chave 15

Caso 1 – Outro exemplo

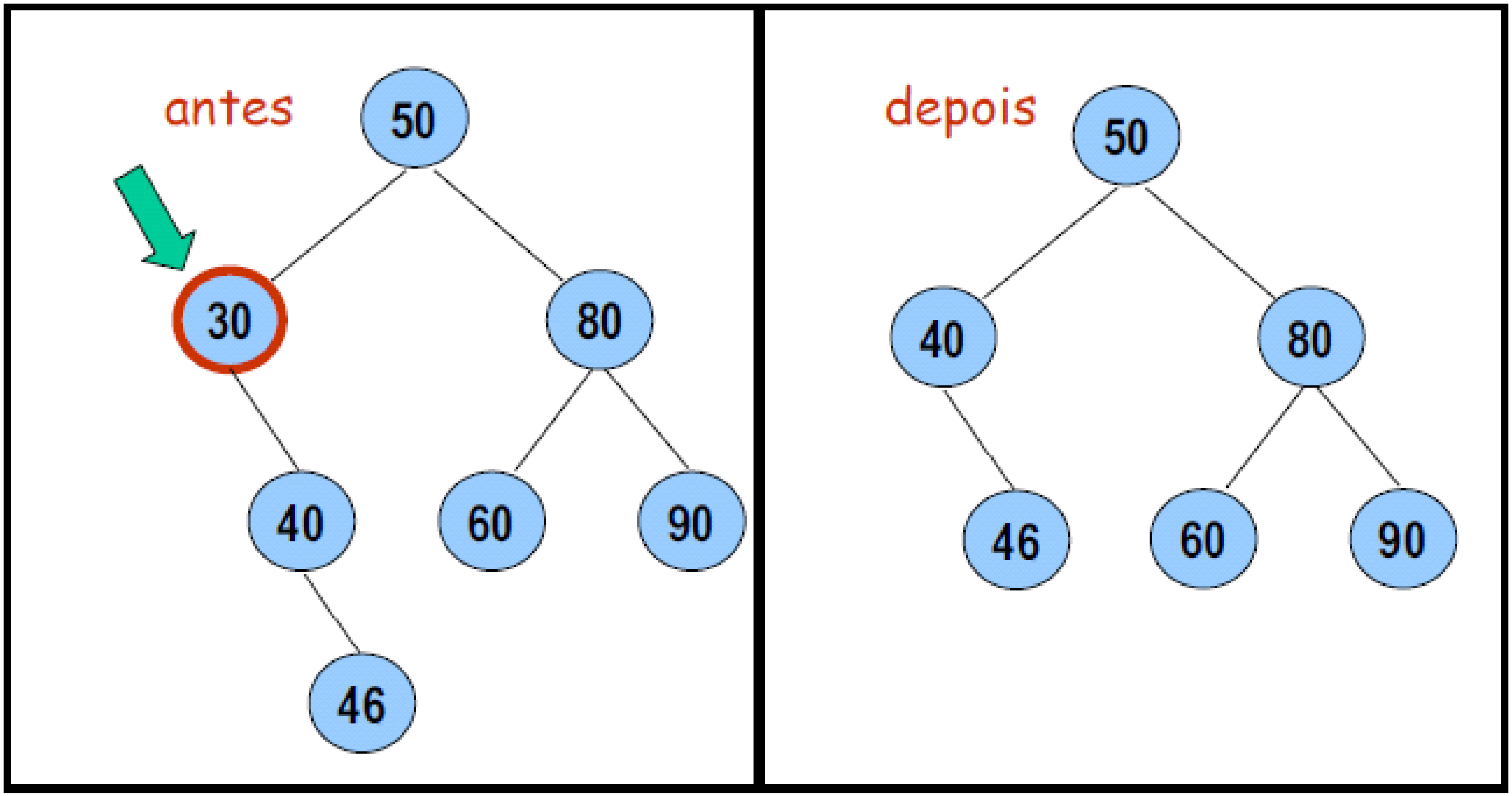


Eliminação da chave 40

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.*

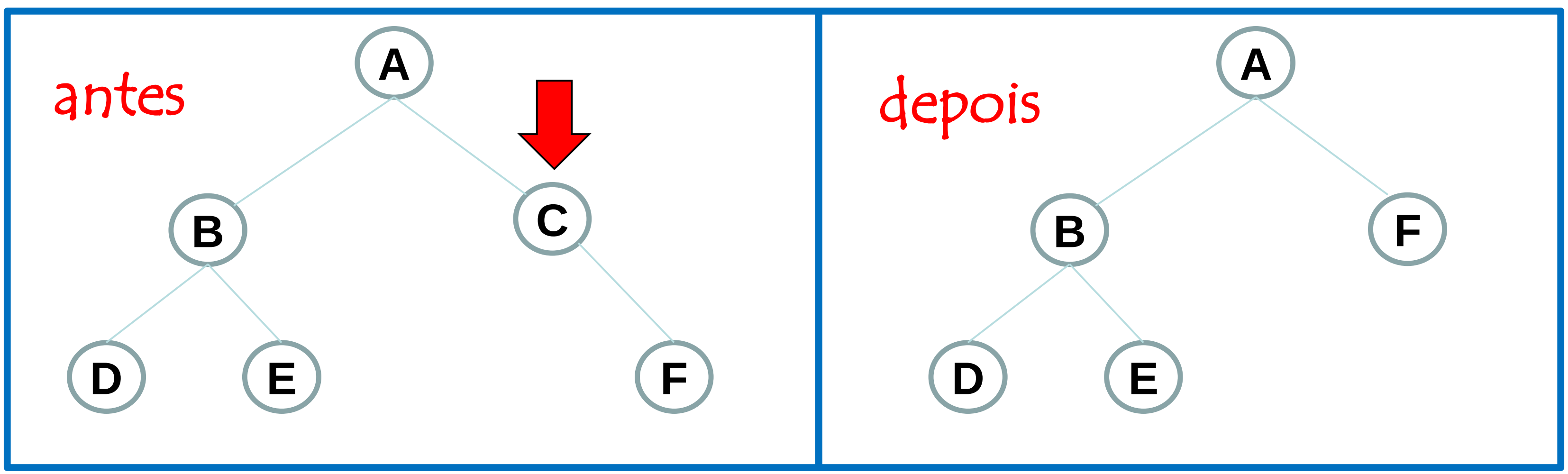
Caso 2 – O nó tem somente uma sub-árvore



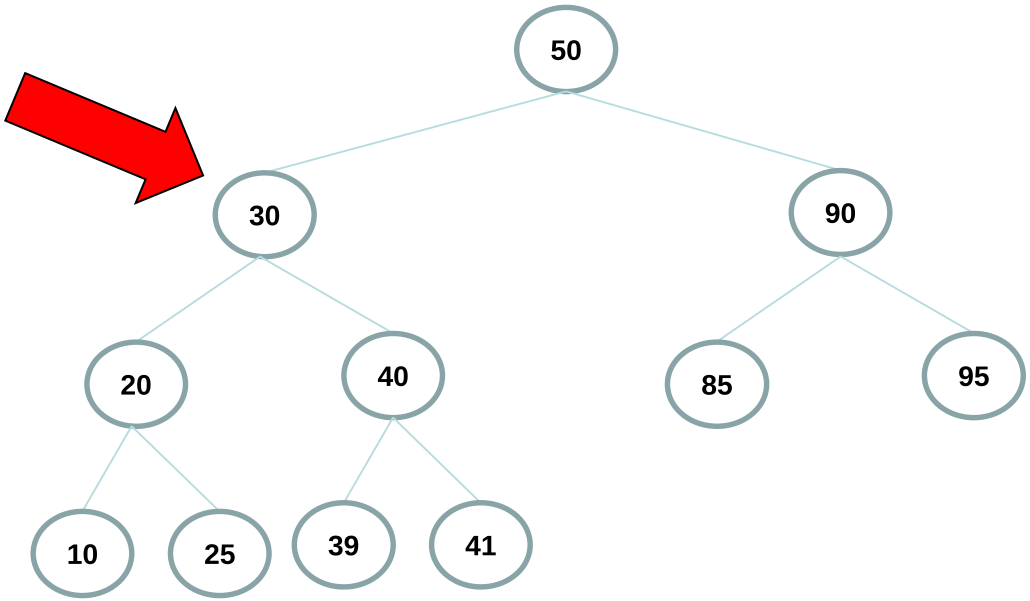
Eliminação da chave 30

O ponteiro do pai aponta para o filho deste

Caso 2 – Outro Exemplo



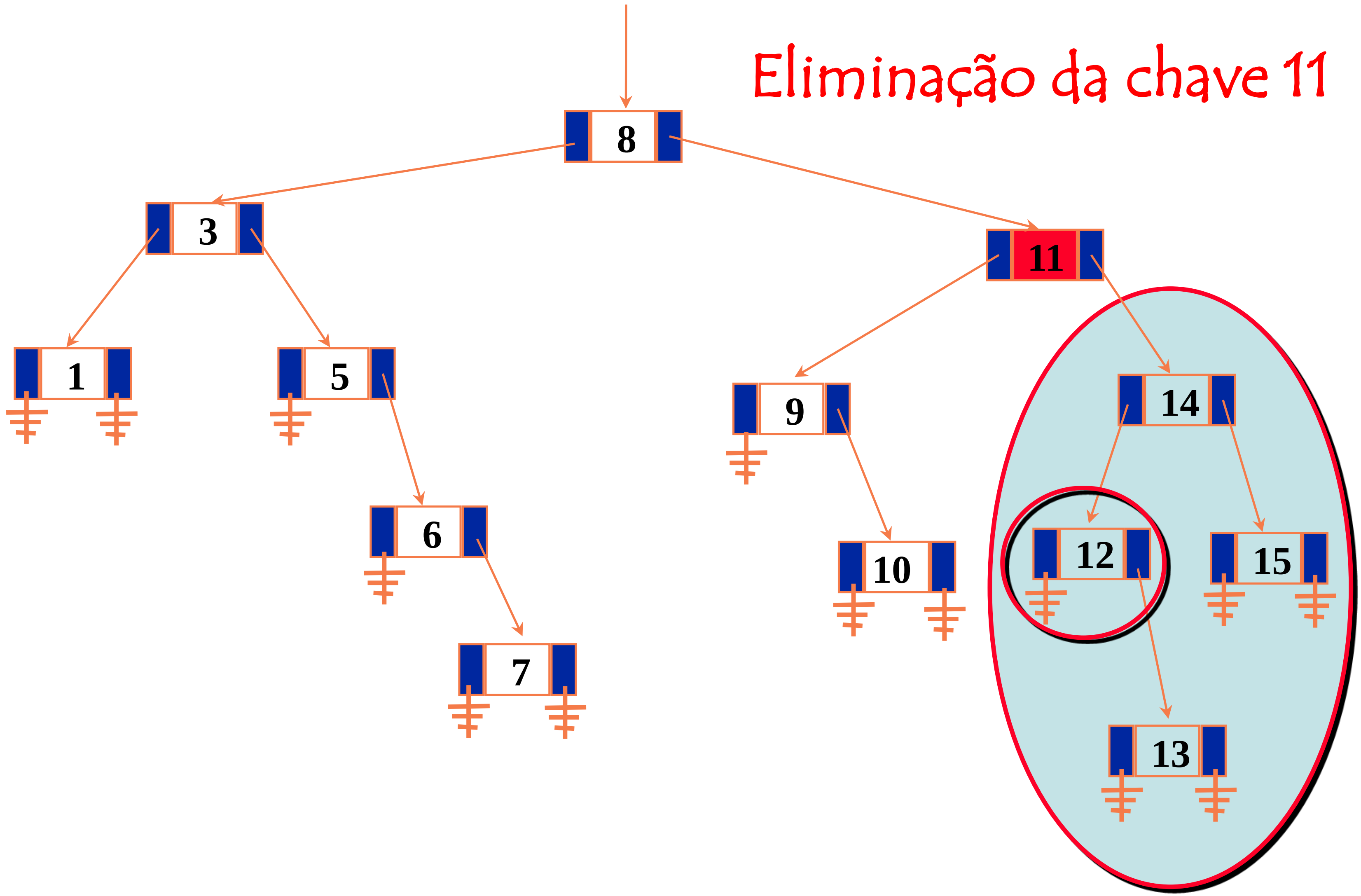
Eliminação de nó que possui duas sub-árvores filhas



<u>A estratégia geral (Mark Allen Weiss)</u> é:

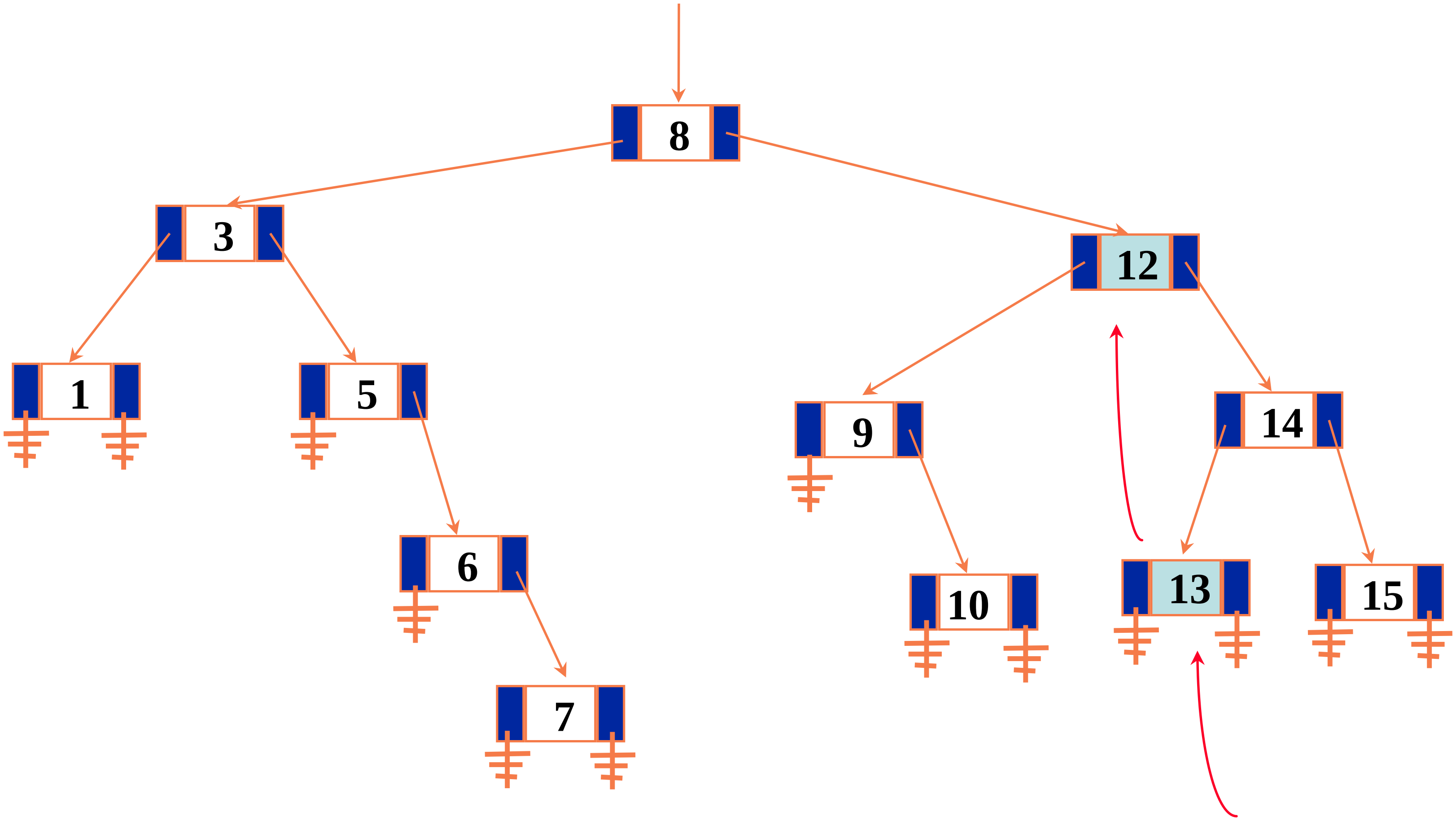
Substituir a chave retirada pela menor chave da sub-árvore direita

Caso 3 – Nó tem duas sub-árvores



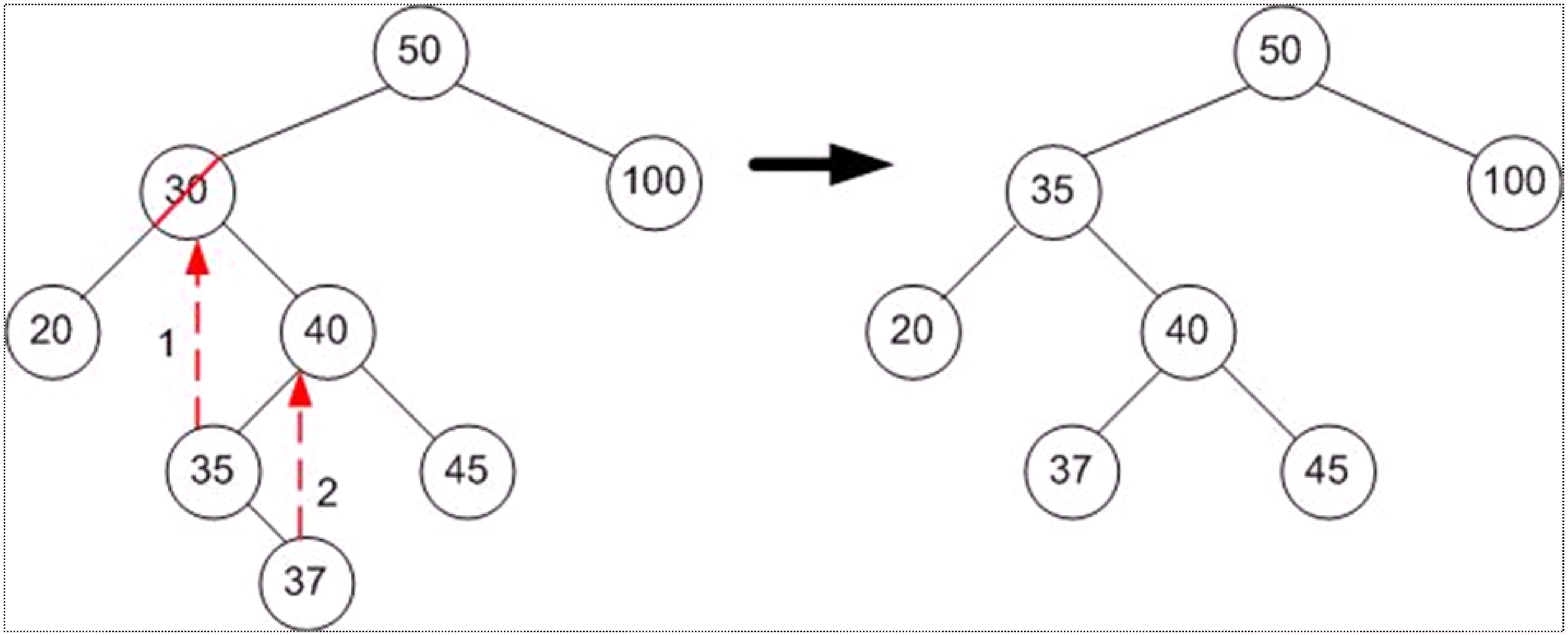
Substituir a chave retirada pela menor chave da sub-árvore direita

Caso 3 – Nó tem duas sub-árvores



Eliminação da chave 11

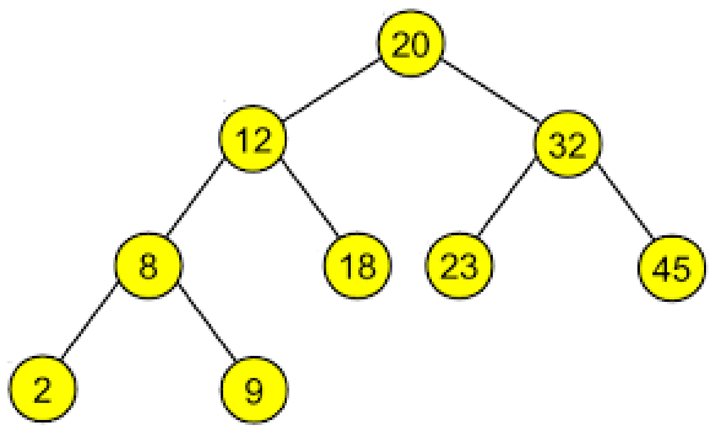
Caso 3 – Outro exemplo



Eliminação da chave 30

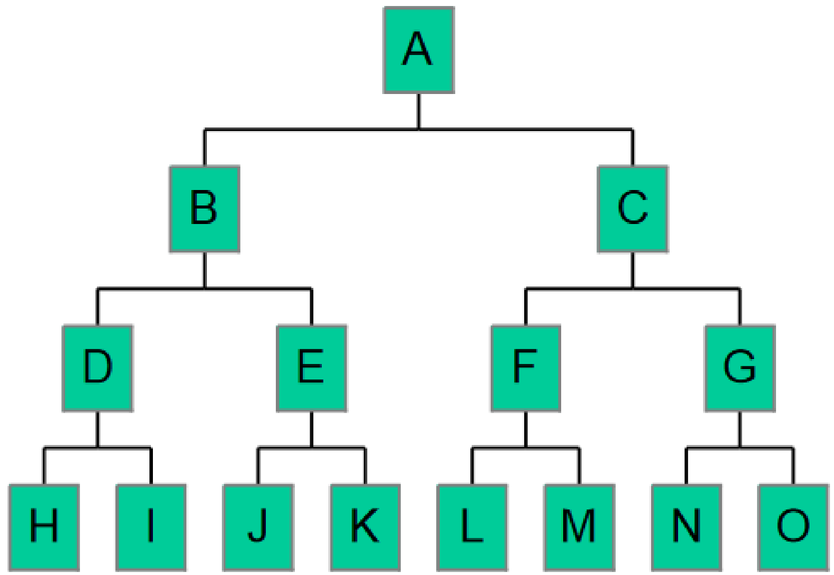
Árvore AVL

- É uma árvore de busca binária autobalanceada;
- Em tal árvore, as **alturas** das suas sub-árvores a partir de cada nó diferem de no **máximo 1 unidade**;
- O nome **AVL** vem de seus criadores (Adelson Velsky e Landis)



Buscas com árvores

- **Para grandes volumes de dados**, o emprego de árvores binárias de pesquisa trás o inconveniente de apresentarem **grande altura**.



- Indexação em grandes volumes de dados tem maior eficiência com o emprego de **Árvores n-árias de pesquisa.**

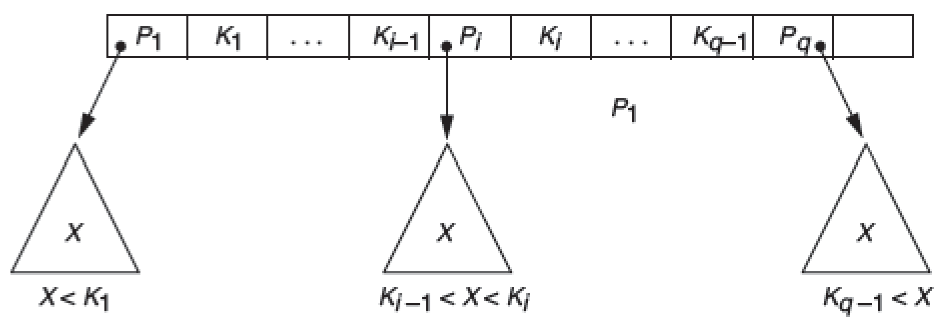
Fonte: Navathe – Capítulo 18

Árvores n-ária de Pesquisa

- Uma árvore n-ária de pesquisa de ordem **p** é uma árvore tal que cada nó contém **no máximo p-1** valores de pesquisa e **p** ponteiros na ordem:

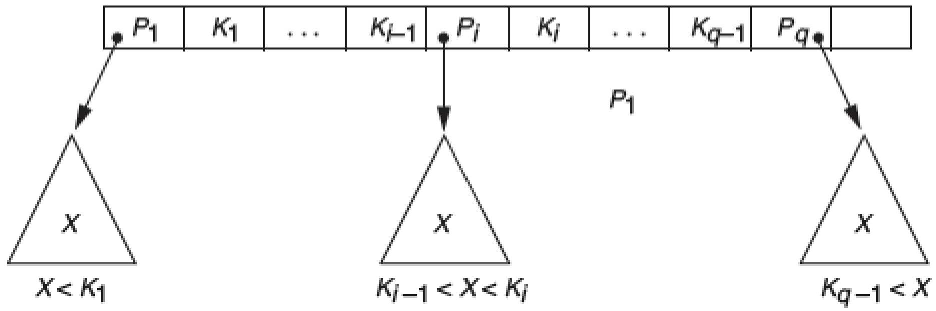
P_1, K_1, P_2, K_2, . . . , P_{q-1} , K_{q-1}, P_q , onde q <= p.

- Cada P_i é um ponteiro para um nó filho (ou **null**), e cada K_i é um valor de pesquisa de algum conjunto ordenado de valores.



Fonte: Navathe – Capítulo 18

Árvores n-ária de Pesquisa



- Em cada nó , $K_1 < K_2 < \; . . . < K_{q-1}$

- Para todos os valores **X** na subárvore apontada por Pi, temos:

$K_{i-1} <$ **X** $< K_i$ para $1 < i < q$;

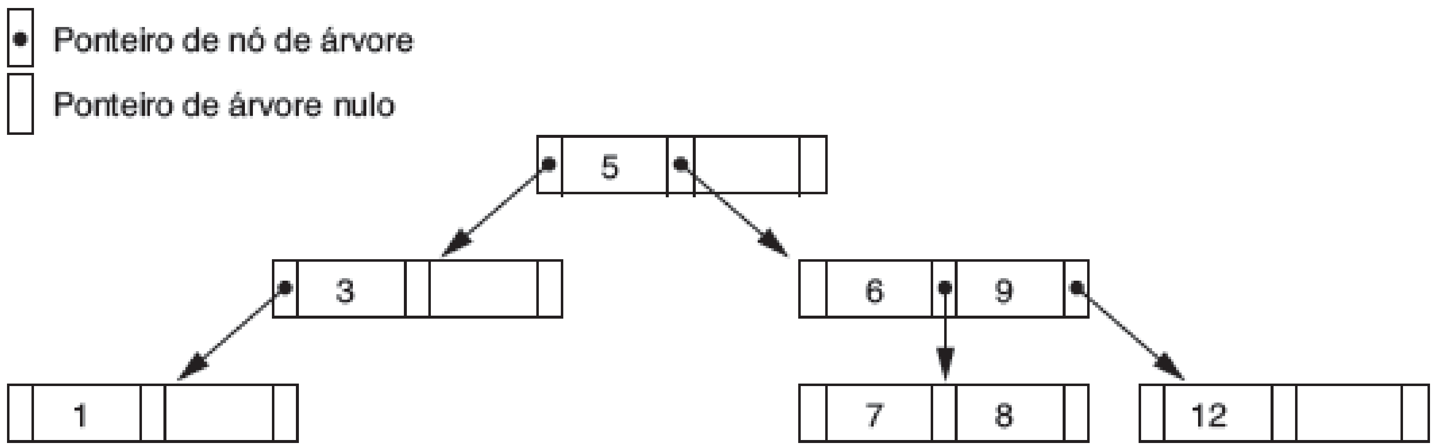
X $< K_i$ para $i = 1$; e

$K_{i-1} <$ **X** para $i = q$.

Obs. Sempre que se procura uma um valor **X**, deve-se seguir o ponteiro P_i apropriado, de acordo com as fórmulas acima.

Fonte: Navathe – Capítulo 18

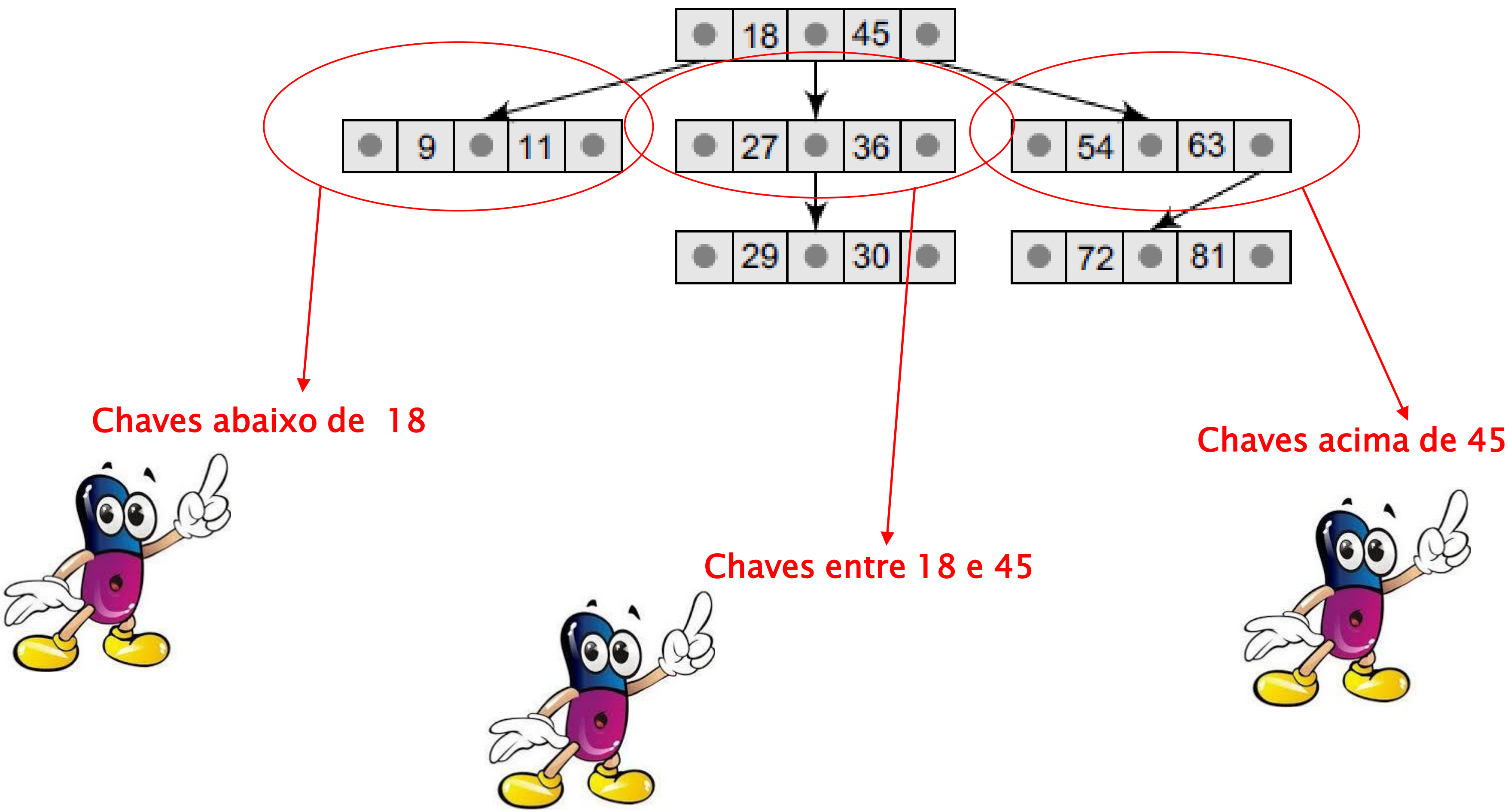
Árvores n-ária de Pesquisa - Exemplo



Uma árvore de pesquisa de ordem $p = 3$.

Fonte: Navathe - Capítulo 18

Árvores n-ária de Pesquisa - Exemplo



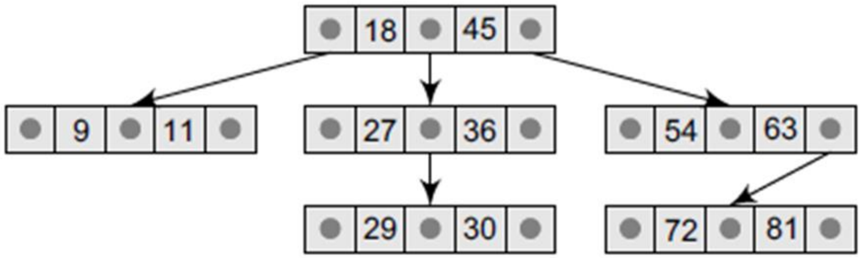
Árvores n-ária de Pesquisa

- Uma **árvore n-ária** de pesquisa pode ser usada como um mecanismo para procurar um registro armazenado em disco.

- Cada valor de chave na árvore é **associado** a um ponteiro para o registro no arquivo de dados que tem esse valor.

- Quando um registro é inserido no arquivo, deve-se **atualizar** a árvore de pesquisa com os correspondentes ponteiros para o registro.

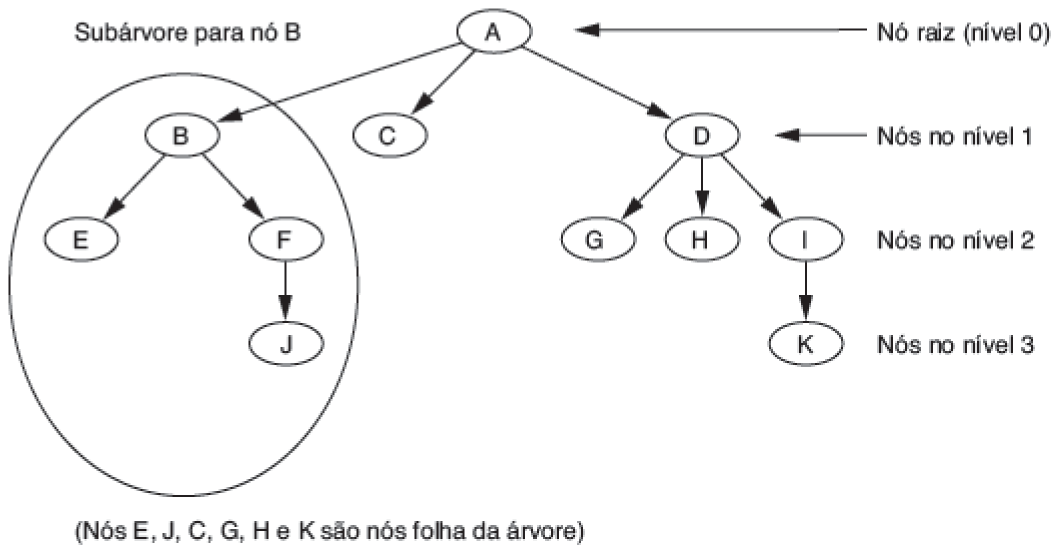
- Em geral, os algoritmos de inserção e deleção de registros **não** garantem que a árvore de pesquisa seja **balanceada**.



Fonte: Navathe – Capítulo 18

Árvores n-ária de Pesquisa Balanceada

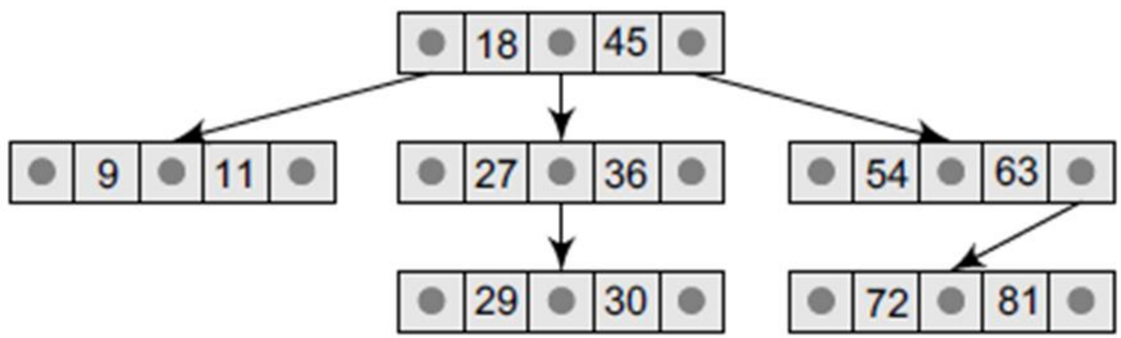
- Uma **árvore n-ária** de pesquisa é balanceada quando os nós folha estão no mesmo nível;



A árvore acima <u>não</u> é balanceada, pois há nós folha nos níveis 1, 2 e 3.

Importância do Balanceamento da árvore

- Garantir que os nós sejam igualmente distribuídos, de modo a **minimizar** a profundidade;
- Tornar a velocidade de pesquisa **uniforme**, de modo que o tempo médio para a busca de qualquer chave aleatória seja aproximadamente o mesmo.

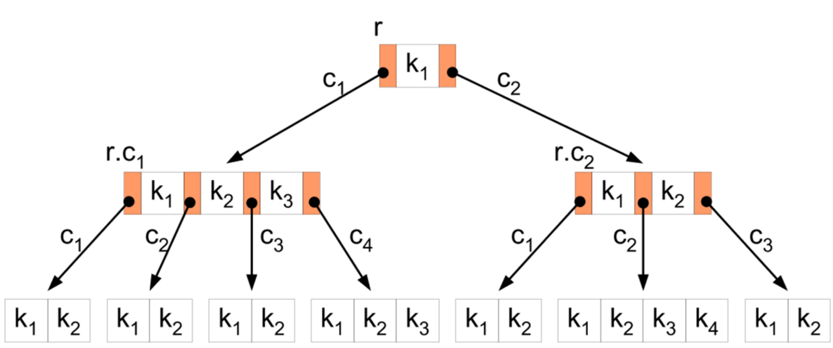


Problemas com a Árvore n-ária de pesquisa

- **Inclusões** e **exclusões** de registros podem causar muita **reestruturação** da árvore;
- Em caso de muitas exclusões, nós podem ficar quase vazios, desperdiçando espaço de armazenamento e aumentando o número de níveis.
- Árvores **B-tree** podem resolver esses problemas.

Árvore B-tree

- Estrutura de dados projetada para memória secundária;
- Permite a inserção, remoção e busca de chaves com complexidade logarítmica;
- **Por essa razão, é largamente empregada em sistemas de bancos de dados**;
- Inventada por Rudolf Bayer e Edward Meyers McCreight em 1971;
- Especula-se que o B venha da palavra balanceada;
- São árvores de pesquisa **n-ária**, porém com **restrições adicionais** que garantem o balanceamento da árvore.



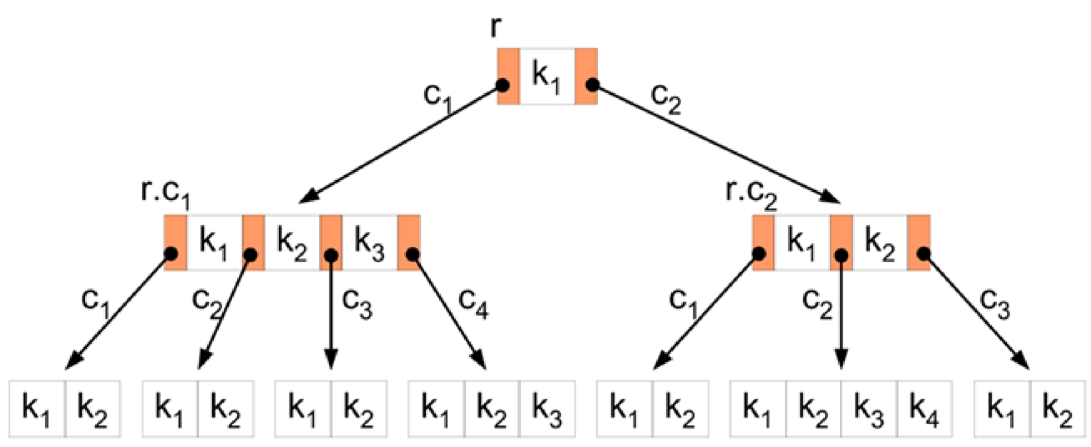
Fonte: Navathe – Capítulo 18

Árvore B-tree - Algoritmo

- Uma **B-tree** começa com um único nó raiz (que inicialmente também é folha) no nível 0;

- Quando o nó raiz está cheio (com p-1 valores de chave) e tentamos inserir outra entrada na árvore, o nó raiz se **divide** (**Split**) em dois nós no nível 1. Somente o valor do meio é mantido no nó raiz, e o restante dos valores é dividido por igual entre os dois nós;

- Quando um nó não-raiz está cheio e uma nova entrada é inserida nele, esse nó é dividido em dois nós no mesmo nível, e a entrada do meio é movida para o nó pai. Se o nó pai estiver cheio, ele também é dividido.



Fonte: Navathe - Capítulo 18