



# Algoritmos e Estrutura de Dados II

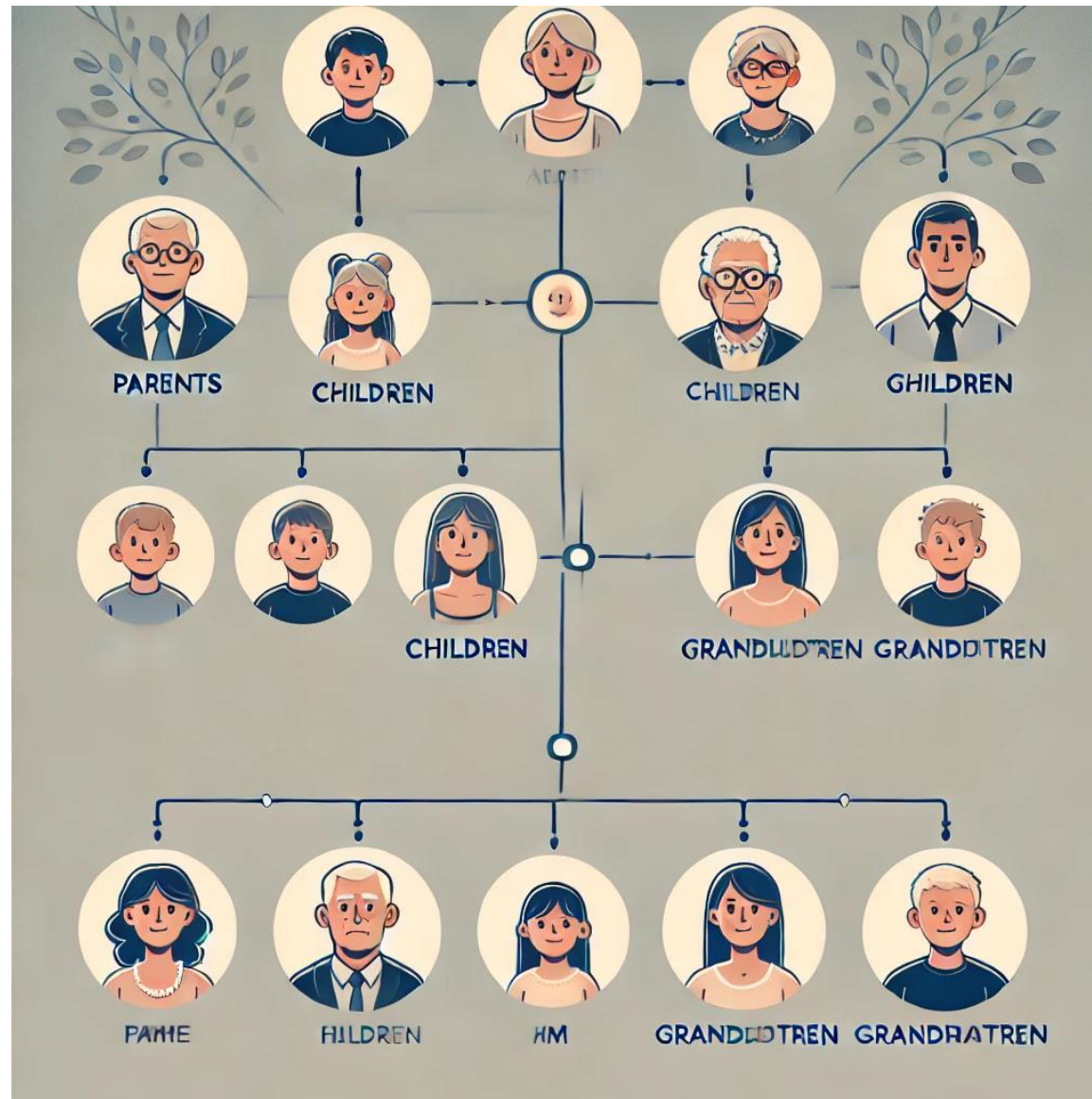
*Aulas 13 e 14*  
*Árvores*

Prof. Dr. Dilermando Piva Jr  
2º Semestre - CDN



# O que essas figuras tem em comum?

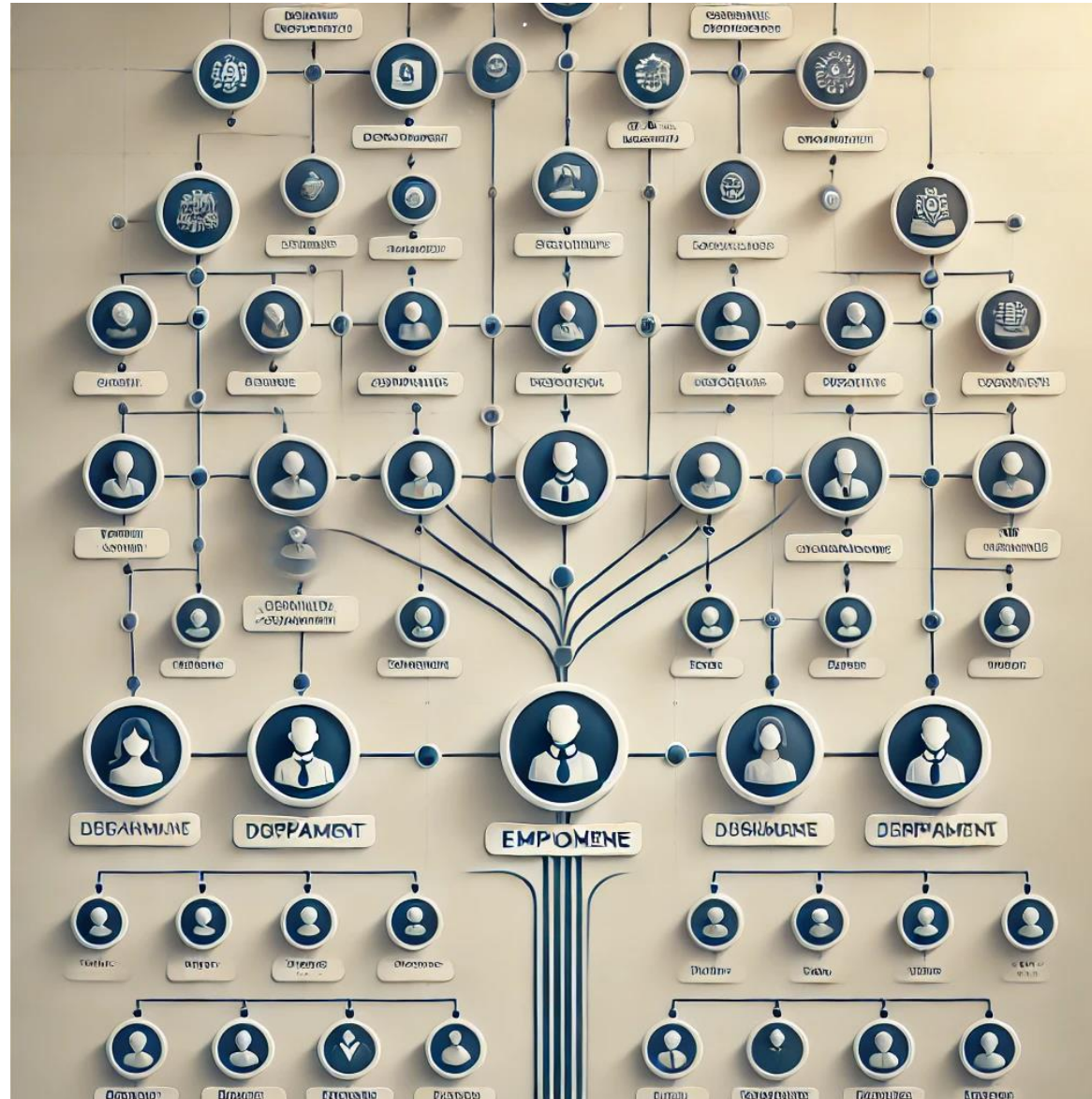
## Árvore Genealógica





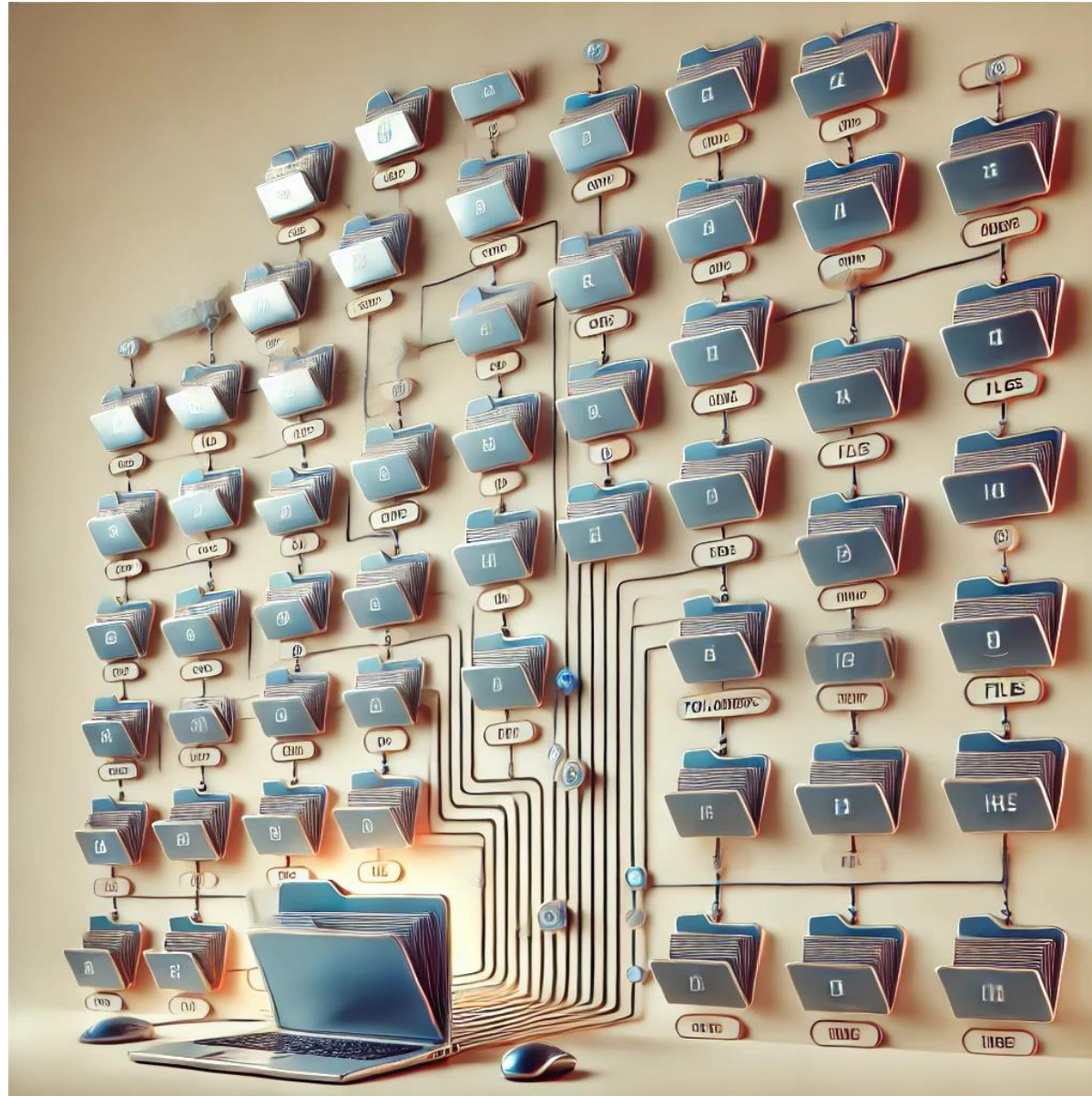
# O que essas figuras tem em comum?

## Árvore (estrutura) Organizacional



# O que essas figuras tem em comum?

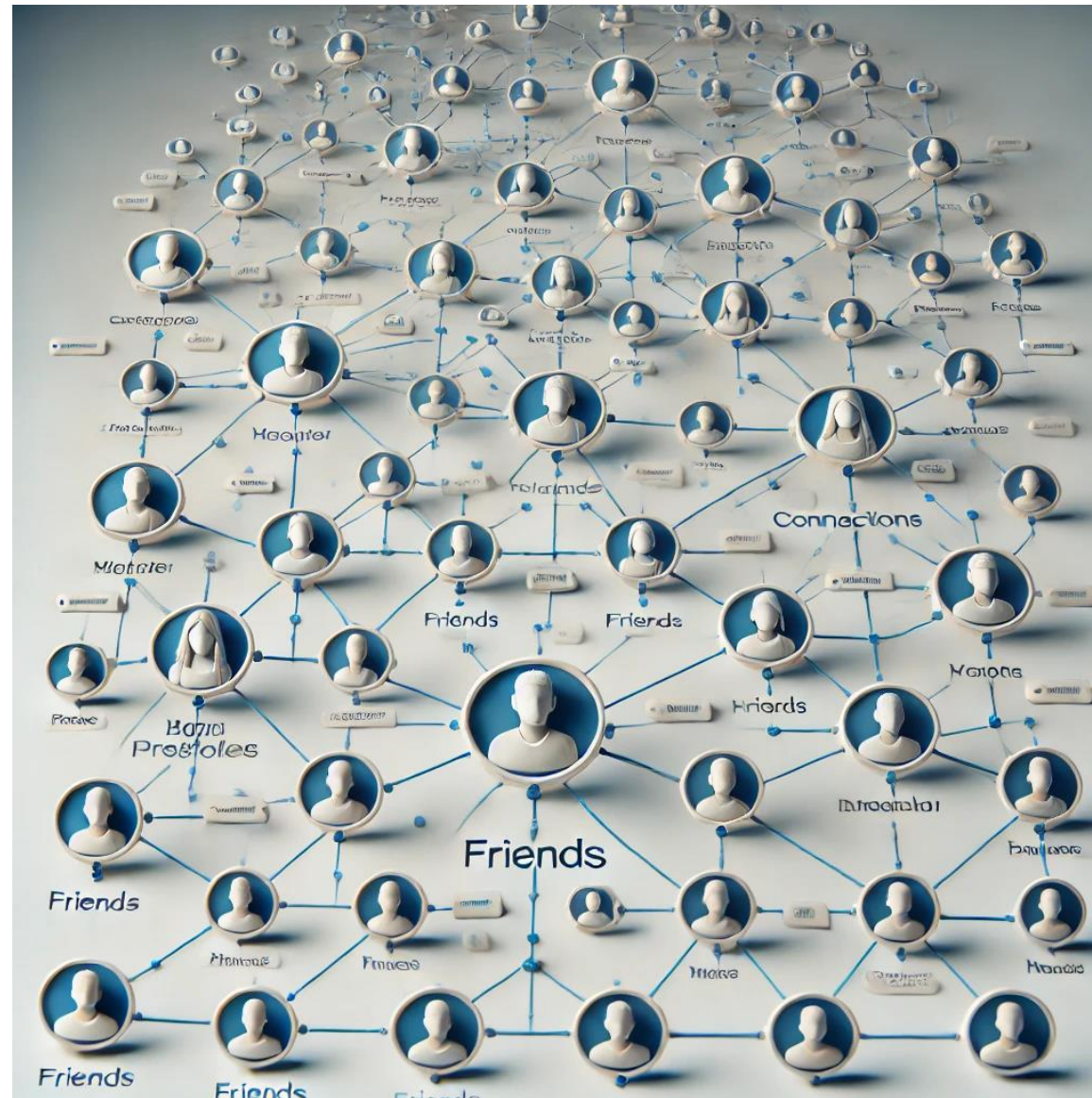
## Árvore De Diretórios





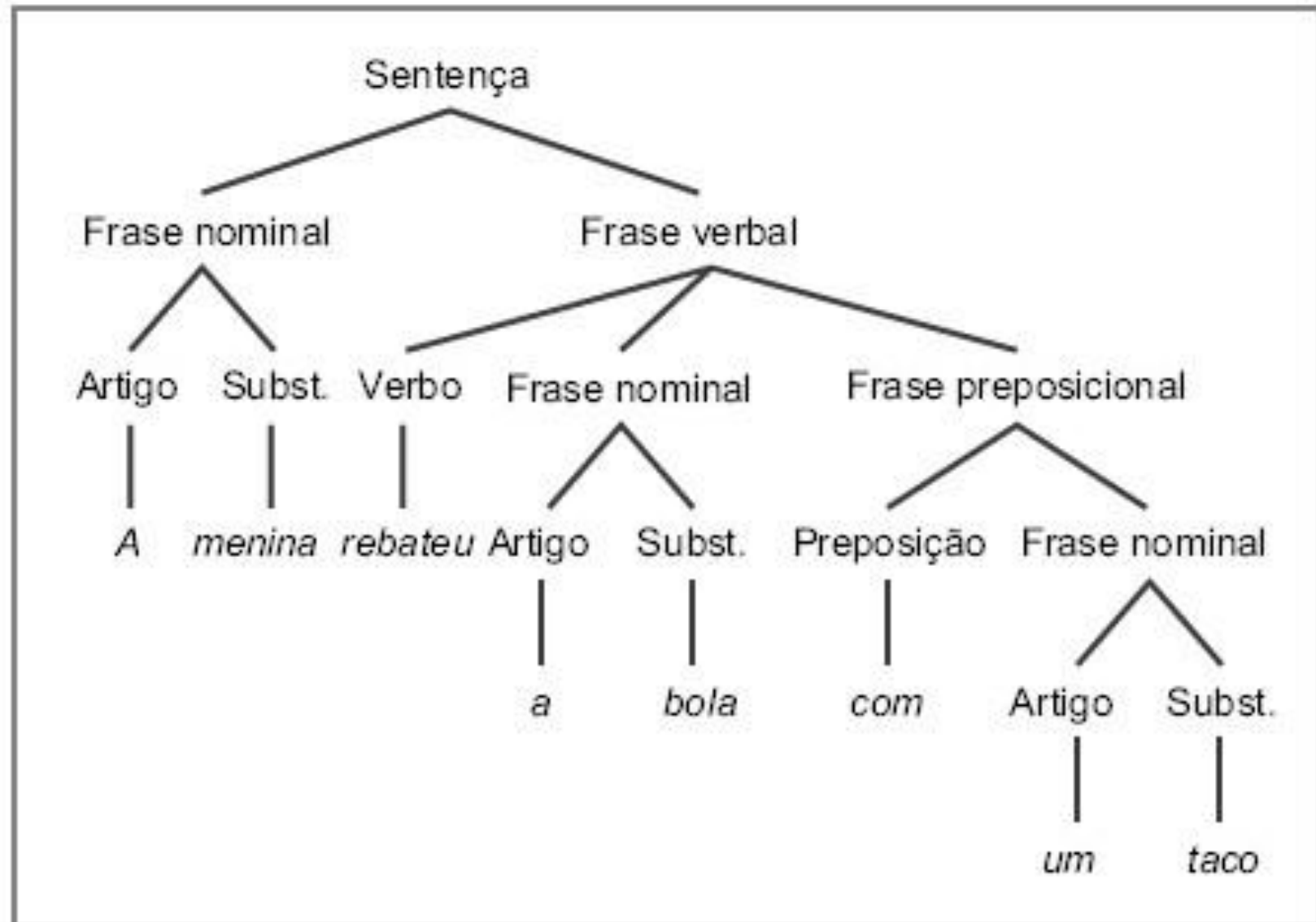
O que essas figuras tem em comum?

# Árvore de Relacionamentos



# O que essas figuras tem em comum?

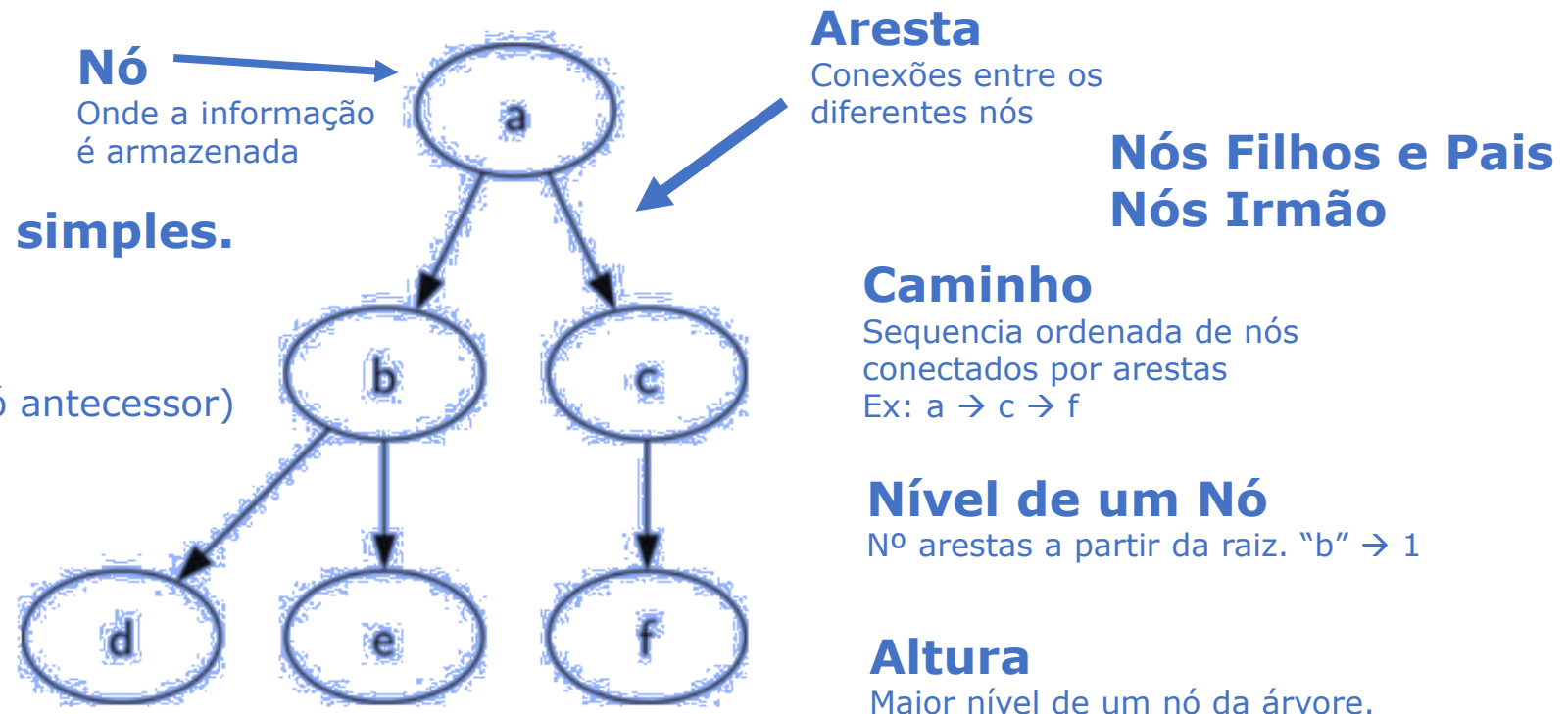
## Árvore de Análise



# Árvores

Árvores são estruturas de dados muito utilizadas em diversas áreas da ciência da computação, como sistemas operacionais, bancos de dados e redes de computadores.

Uma estrutura de dados do tipo árvore possui uma raiz a partir da qual diferentes ramos conectam um conjunto de nós intermediários até as folhas da árvore.



**Essa figura ilustra uma árvore simples.**  
**É composta por 6 nós.**

**"a" é a raiz** (único nó que não possui nó antecessor)

**"d", "e" e "f" são as folhas.**  
(nós que não possuem filhos)

# Árvores

Árvores são estruturas de dados muito utilizadas em diversas áreas da ciência da computação, como sistemas operacionais, bancos de dados e redes de computadores.

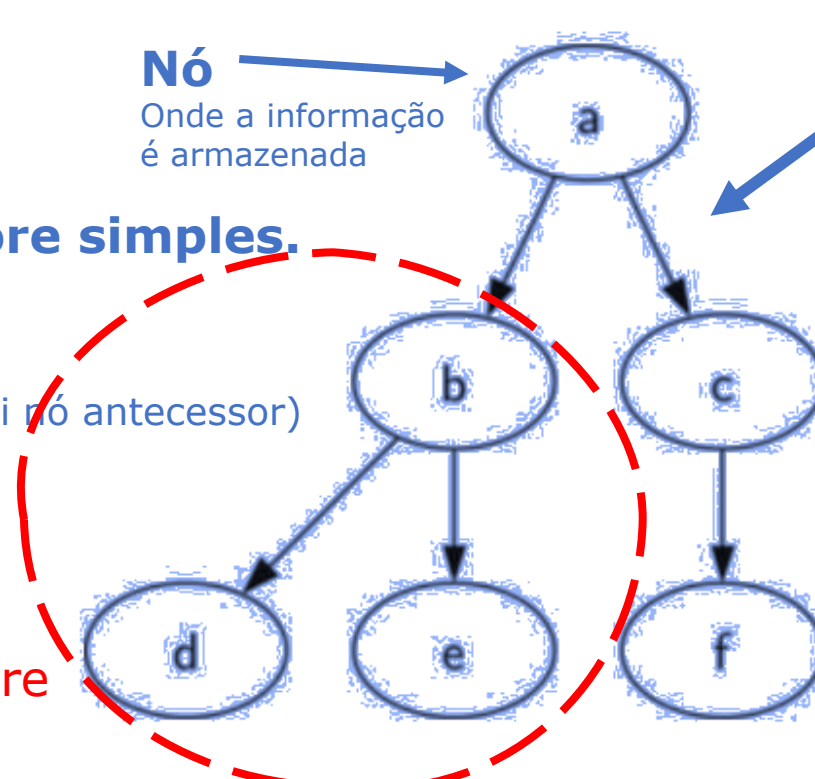
Uma estrutura de dados do tipo árvore possui uma raiz a partir da qual diferentes ramos conectam um conjunto de nós intermediários até as folhas da árvore.

**Essa figura ilustra uma árvore simples.**  
**É composta por 6 nós.**

**"a" é a raiz** (único nó que não possui nó antecessor)

**"d", "e" e "f" são as folhas.**  
(nós que não possuem filhos)

Subárvore



**Nó**  
Onde a informação  
é armazenada

**Aresta**  
Conexões entre os  
diferentes nós

**Nós Filhos e Pais**  
**Nós Irmão**

**Caminho**  
Sequencia ordenada de nós  
conectados por arestas  
Ex:  $a \rightarrow c \rightarrow f$

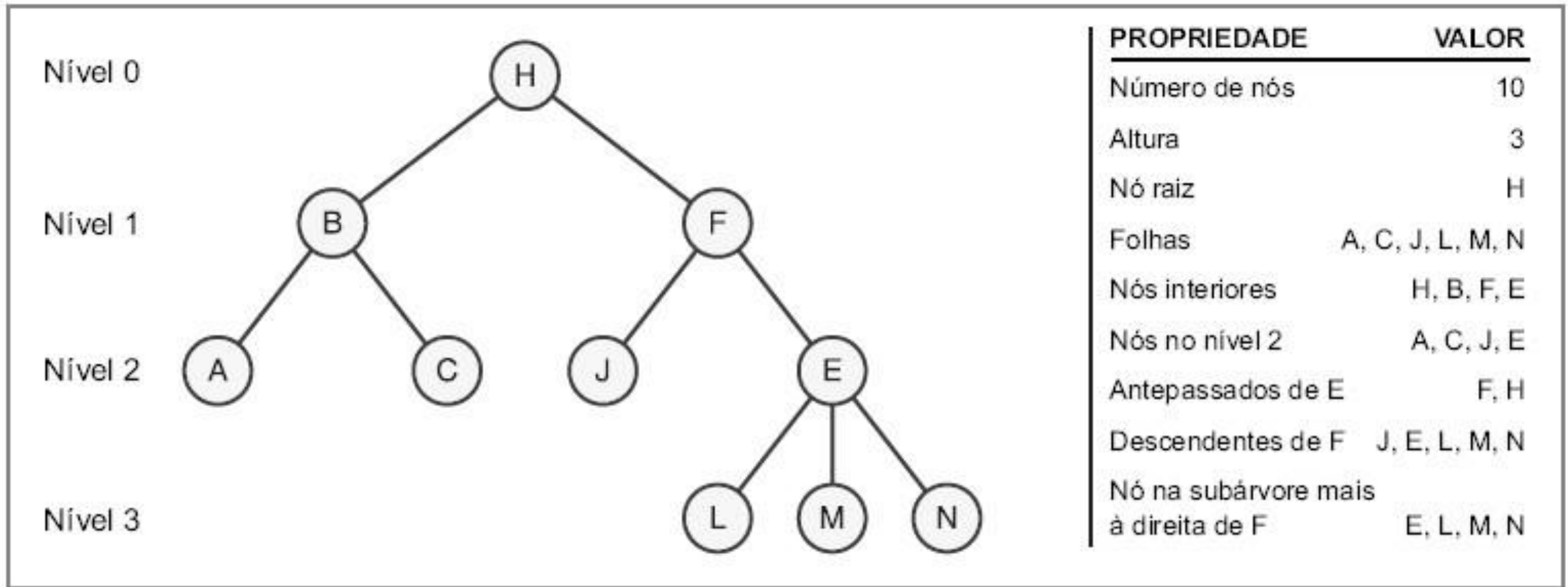
**Nível de um Nó**  
Nº arestas a partir da raiz. "b"  $\rightarrow$  1

**Altura**  
Maior nível de um nó da árvore.



# Árvores

Recapitulando...



# Árvores (definição formal 1)

Uma árvore consiste em um conjunto de nós e um conjunto de arestas que conectam pares de nós. Uma árvore possui as seguintes propriedades:

**i) Toda árvore tem um nó designado de **raiz**, por onde a busca, a inserção e a remoção de elementos deve iniciar. Em outras palavras, é a porta de entrada para o conjunto de dados.**

ii) Todo nó  $n$ , com exceção da raiz, é conectado por uma aresta a exatamente um nó pai  $p$ . Ou seja, cada nó da árvore tem precisamente um único pai.

iii) Existe um único caminho saindo da raiz e chegando em um nó arbitrário da árvore. Ou seja, sempre que desejarmos acessar um determinado nó, iremos sempre pelo mesmo caminho.

iv) Se cada nó da árvore possui no máximo dois nós filhos, dizemos que a árvore é uma árvore binária.

# Árvores (definição formal 1)

Uma árvore consiste em um conjunto de nós e um conjunto de arestas que conectam pares de nós. Uma árvore possui as seguintes propriedades:

i) Toda árvore tem um nó designado de raiz, por onde a busca, a inserção e a remoção de elementos deve iniciar. Em outras palavras, é a porta de entrada para o conjunto de dados.

**ii) Todo nó  $n$ , com exceção da raiz, é conectado por uma aresta  $a$  exatamente um nó pai  $p$ . Ou seja, cada nó da árvore tem precisamente um único pai.**

iii) Existe um único caminho saindo da raiz e chegando em um nó arbitrário da árvore. Ou seja, sempre que desejarmos acessar um determinado nó, iremos sempre pelo mesmo caminho.

iv) Se cada nó da árvore possui no máximo dois nós filhos, dizemos que a árvore é uma árvore binária.



# Árvores (definição formal 1)

Uma árvore consiste em um conjunto de nós e um conjunto de arestas que conectam pares de nós. Uma árvore possui as seguintes propriedades:

i) Toda árvore tem um nó designado de raiz, por onde a busca, a inserção e a remoção de elementos deve iniciar. Em outras palavras, é a porta de entrada para o conjunto de dados.

ii) Todo nó  $n$ , com exceção da raiz, é conectado por uma aresta a exatamente um nó pai  $p$ . Ou seja, cada nó da árvore tem precisamente um único pai.

iii) Existe um **único caminho** saindo da raiz e chegando em um nó arbitrário da árvore. Ou seja, sempre que desejarmos acessar um determinado nó, iremos sempre pelo mesmo caminho.

iv) Se cada nó da árvore possui no máximo dois nós filhos, dizemos que a árvore é uma árvore binária.

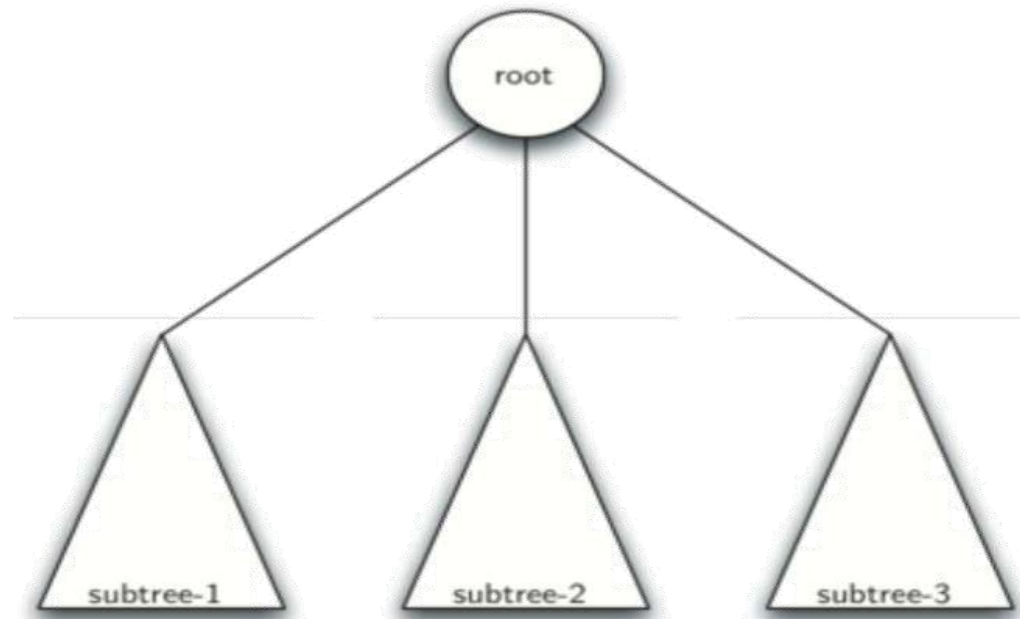
# Árvores (definição formal 1)

Uma árvore consiste em um conjunto de nós e um conjunto de arestas que conectam pares de nós. Uma árvore possui as seguintes propriedades:

- i) Toda árvore tem um nó designado de raiz, por onde a busca, a inserção e a remoção de elementos deve iniciar. Em outras palavras, é a porta de entrada para o conjunto de dados.
- ii) Todo nó  $n$ , com exceção da raiz, é conectado por uma aresta a exatamente um nó pai  $p$ . Ou seja, cada nó da árvore tem precisamente um único pai.
- iii) Existe um único caminho saindo da raiz e chegando em um nó arbitrário da árvore. Ou seja, sempre que desejarmos acessar um determinado nó, iremos sempre pelo mesmo caminho.
- iv) Se cada nó da árvore possui no máximo dois nós filhos, dizemos que a árvore é uma árvore binária.**

# Árvores (definição formal 2)

Uma árvore ou é vazia ou consiste de uma raiz com zero ou mais subárvores, cada uma sendo uma árvore. A raiz de cada subárvore é conectada a raiz da árvore pai por uma aresta.



Pela **definição recursiva**, sabemos que a árvore acima possui **pelo menos 4 nós**, uma vez que cada triângulo representando uma subárvore deve possuir uma raiz. Na verdade, ela pode ter muito mais nós do que isso, mas não sabemos pois não conhecemos a estrutura interna de cada subárvore.



# Árvores binárias

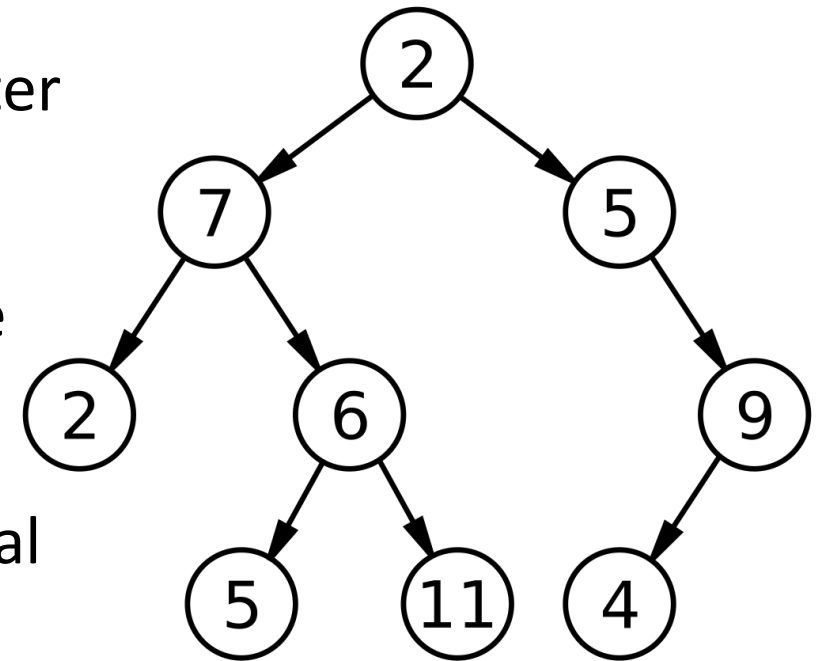
Os nós de uma árvore binária possuem **graus zero, um ou dois**. Um nó de grau zero é denominado folha.

Em uma árvore binária, por definição, cada nó poderá ter **até duas folhas**.

A **profundidade** de um nó é a distância deste nó até a raiz. Um conjunto de nós com a mesma profundidade é denominado nível da árvore. **A maior profundidade de um nó, é a altura da árvore.**

Uma árvore "estritamente binária" é uma árvore na qual todo nó tem zero ou duas folhas.

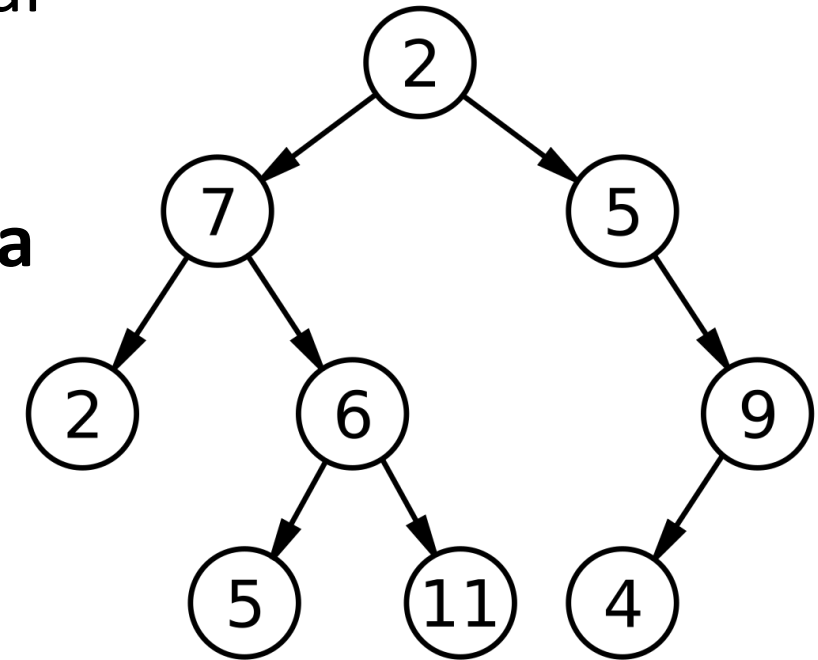
Existem autores, porém, que adotam essa definição para o termo quase completa, e utilizam o termo completa apenas para árvores em que todos os níveis têm o máximo número de elementos.



# Árvores binárias

Existem basicamente duas formas de implementar árvores binárias em Python:

- 1) utilizando uma **lista de sublistas**, ou
- 2) utilizando encadeamento lógico (como na **lista encadeada**)



## Principais Métodos:

### Métodos

BinaryTree()  
get\_left\_child()  
get\_right\_child()  
set\_root\_val(val)  
get\_root\_val()  
insert\_left(val)  
insert\_right(val)

### Descrição

cria uma nova instância da árvore binária  
retorna a subárvore a esquerda do nó corrente  
retorna a subárvore a direita do nó corrente  
armazena um valor no nó corrente  
retorna o valor armazenado no nó corrente  
cria uma nova árvore binária a esquerda do nó corrente  
cria uma nova árvore binária a direita do nó corrente

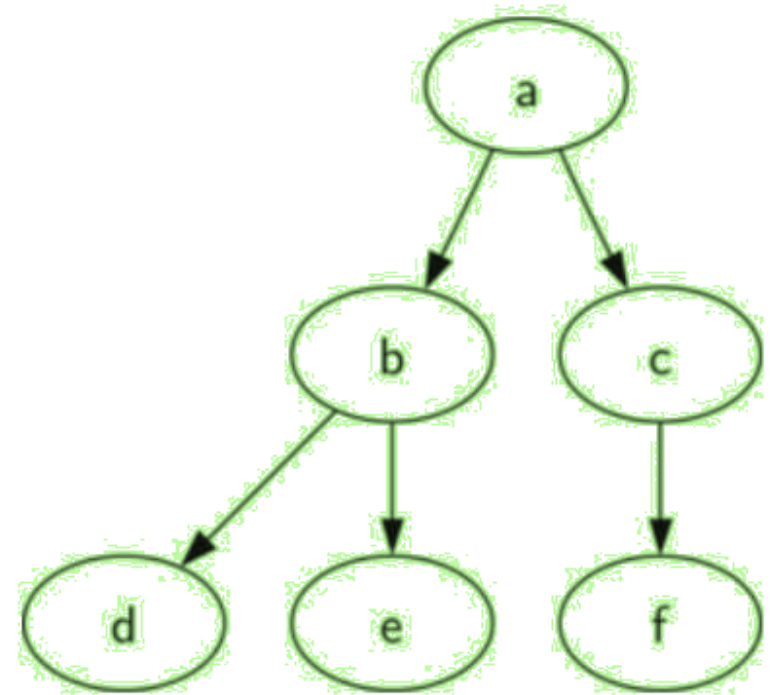
# Árvores Binária: LISTAS DE LISTAS

Em uma representação de lista de sublistas, o valor armazenado na raiz é sempre o primeiro elemento da lista. O segundo elemento da lista será a sublista que representa a subárvore a esquerda. Analogamente, o terceiro elemento da lista será a sublista que representa a subárvore a direita.

Para ilustrar como fica a representação da árvore da figura ao lado, apresentamos o código a seguir.

# Representação de árvore como lista de sublistas

```
my_tree = ['a', # raiz
           ['b', # subárvore esquerda
            ['d', [], []],
            ['e', [], []]
           ],
           ['c', # subárvore direita
            ['f', [], []],
            []
           ]
          ]
```

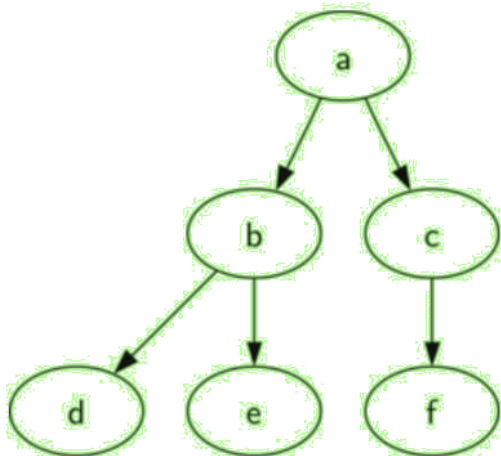




# Árvores Binária: LISTAS DE LISTAS

# Representação de árvore como lista de sublistas

```
my_tree = ['a', # raiz
           ['b', # subárvore esquerda
            ['d', [], []],
            ['e', [], []]
           ],
           ['c', # subárvore direita
            ['f', [], []],
            []
           ]
          ]
```



Uma propriedade muito interessante desta representação é que a estrutura de uma lista representando uma subárvore possui a mesma estrutura de uma árvore (lista), de modo que define uma representação recursiva.

```
print(my_tree)
print('Subárvore esquerda = ', my_tree[1])
print('Raiz = ', my_tree[0])
print('Subárvore direita = ', my_tree[2])
```

# Árvores Binária: LISTAS DE LISTAS (IMPLEMENTAÇÃO)

```
# Cria árvore binária com raiz r
def binary_tree(r):
    return [r, [], []]

# Insere novo ramo a esquerda da raiz
def insert_left(root, new_branch):
    # Analisa a subárvore a esquerda
    t = root.pop(1)
    # Se a subárvore a esquerda não é vazia
    if len(t) > 1:
        # Insere na posição 1 da raiz (esquerda)
        # Novo ramo será a raiz da subárvore a esquerda
        # Adiciona t na esquerda do novo ramo
        root.insert(1, [new_branch, t, []])
    else:
        # Se t for vazia, não há subárvore a esquerda
        root.insert(1, [new_branch, [], []])
    return root
```

# Árvores Binária: LISTAS DE LISTAS (IMPLEMENTAÇÃO)

```
# Insere novo ramo a direita da raiz
def insert_right(root, new_branch):
    # Analisa a subárvore a direita
    t = root.pop(2)
    # Se a subárvore a direita não é vazia
    if len(t) > 1:
        # Insere na posição 2 da raiz (direita)
        # Novo ramo será a raiz da subárvore a direita
        # Adiciona t na direita do novo ramo
        root.insert(2, [new_branch, [], t])
    else:
        # Se t for vazia, não há subárvore a direita
        root.insert(2, [new_branch, [], []])
    return root
```

# Árvores Binária: LISTAS DE LISTAS (IMPLEMENTAÇÃO)

```
def get_root_val(root):  
    return root[0]
```

```
def set_root_val(root, new_val):  
    root[0] = new_val
```

```
def get_left_child(root):  
    return root[1]
```

```
def get_right_child(root):  
    return root[2]
```



# Árvores Binária: LISTAS DE LISTAS (TESTANDO A IMPL.)

```
# Cria árvore binária
r = binary_tree(3)
# Adiciona subárvore a esquerda
insert_left(r, 4)
# Adiciona subárvore a esquerda
insert_left(r, 5)
# Adiciona subárvore a direita
insert_right(r, 6)
# Adiciona subárvore a direita
insert_right(r, 7)
print(r)

# Obtém subárvore a esquerda da raiz
l = get_left_child(r)
print(l)
# Muda a raiz da subárvore a esquerda
set_root_val(l, 9)
print(r)
# Insere a esquerda da subárvore a esquerda
insert_left(l, 11)
print(r)
```

# VAMOS PARA A PRÁTICA ?!!!



# Árvores Binária: LISTAS DE LISTAS (EXERCÍCIO)

Considere o seguinte código em Python.

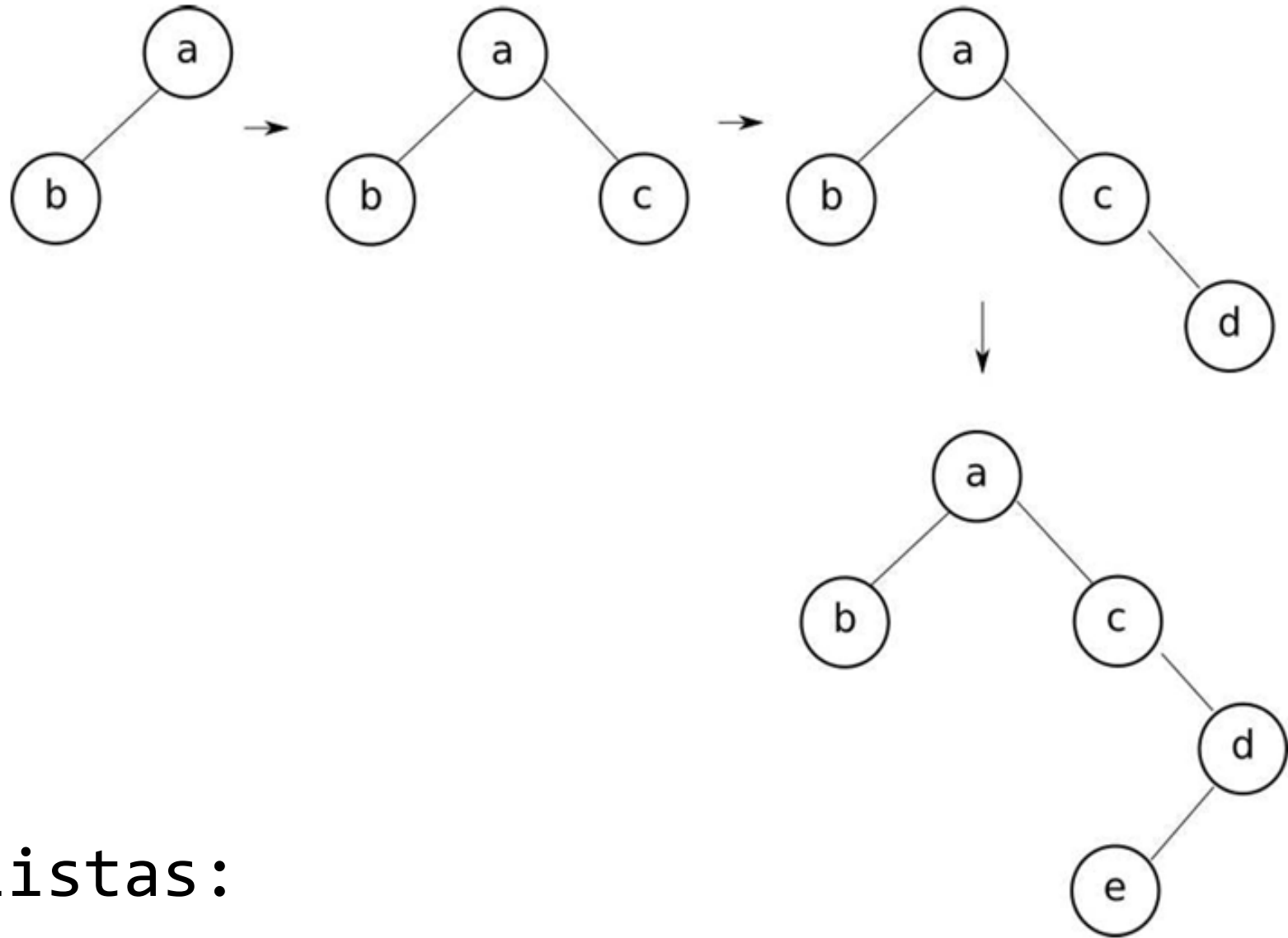
```
x = binary_tree('a')  
insert_left(x, 'b')  
insert_right(x, 'c')  
insert_right(get_right_child(x), 'd')  
insert_left(get_right_child(get_right_child(x)), 'e')
```

Desenhe a árvore resultante e forneça a representação usando lista de listas

**AGORA É COM  
VOCÊ !!!!**



# Árvore Resultante:



Representação com listas:

`['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []], []]]`

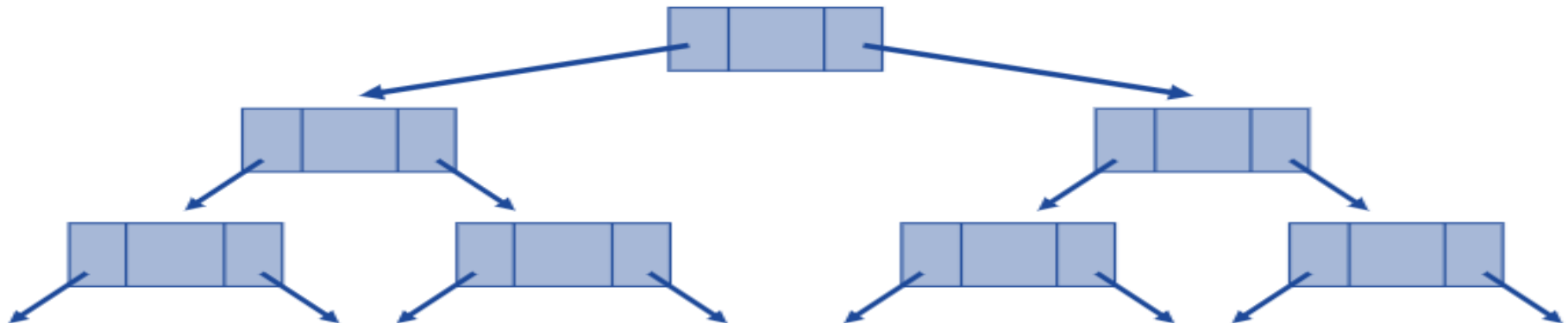


# Árvores Binária: LISTAS LIGADAS (Representação com referências)

Apesar de interessante, a representação de árvores binárias utilizando lista de listas não é muito intuitiva, principalmente quando o número de nós da árvore cresce.

O número de sublistas fica tão elevado que é muito fácil cometer erros e equívocos durante a manipulação da estrutura.

Sendo assim, veremos agora, como representar árvores binárias utilizando referências e um encadeamento lógico similar ao adotado nas listas duplamente encadeadas, em que cada nó possui duas referências: uma para o nó filho a esquerda e outra para o nó filho a direita.



# Árvores Binárias...

## *Uma primeira implementação*

```
class BinaryTree:
    # Construtor
    def __init__(self, valor):
        self.key = valor
        self.left_child = None
        self.right_child = None

    # Insere nó a esquerda
    def insert_left(self, valor):
        # Se nó corrente não tem filho a esquerda, OK
        if self.left_child == None:
            self.left_child = BinaryTree(valor)
        else:
            # Se tem filho a esquerda, pendura subárvore
            # a esquerda do nó corrente na esquerda do
            # novo nó recém criado
            temp = BinaryTree(valor)
            temp.left_child = self.left_child
            self.left_child = temp
```

```
# Insere nó a direita
def insert_right(self, valor):
    # Se nó corrente não tem filho a direita, OK
    if self.right_child == None:
        self.right_child = BinaryTree(valor)
    else:
        # Se tem filho a direita, pendura subárvore
        # a direita do nó corrente na direita do
        # novo nó recém criado
        temp = BinaryTree(valor)
        temp.right_child = self.right_child
        self.right_child = temp

    # Obtém filho a direita
    def get_right_child(self):
        return self.right_child

    # Obtém filho a esquerda
    def get_left_child(self):
        return self.left_child

    # Atualiza valor do nó corrente
    def set_root_val(self, valor):
        self.key = valor

    # Obtém valor do nó corrente
    def get_root_val(self):
        return self.key
```

# Árvores Binárias...

## *Testando...*

```
# Cria nó com valor 'a'
```

```
r = BinaryTree('a')
```

```
print(r.get_root_val())
```

```
print(r.get_left_child())
```

```
print(r.get_right_child())
```

```
# Insere nó com valor 'b' a esquerda da raiz
```

```
r.insert_left('b')
```

```
print(r.get_left_child().get_root_val())
```

```
# Insere nó com valor 'c' na direita da raiz
```

```
r.insert_right('c')
```

```
print(r.get_right_child().get_root_val())
```

```
# Insere nó com valor 'd' a esquerda no filho a esquerda da raiz
```

```
r.get_left_child().insert_left('d')
```

```
print(r.get_left_child().get_left_child().get_root_val())
```

```
# Insere nó com valor 'e' a direita no filho a esquerda da raiz
```

```
r.get_left_child().insert_right('e')
```

```
print(r.get_left_child().get_right_child().get_root_val())
```

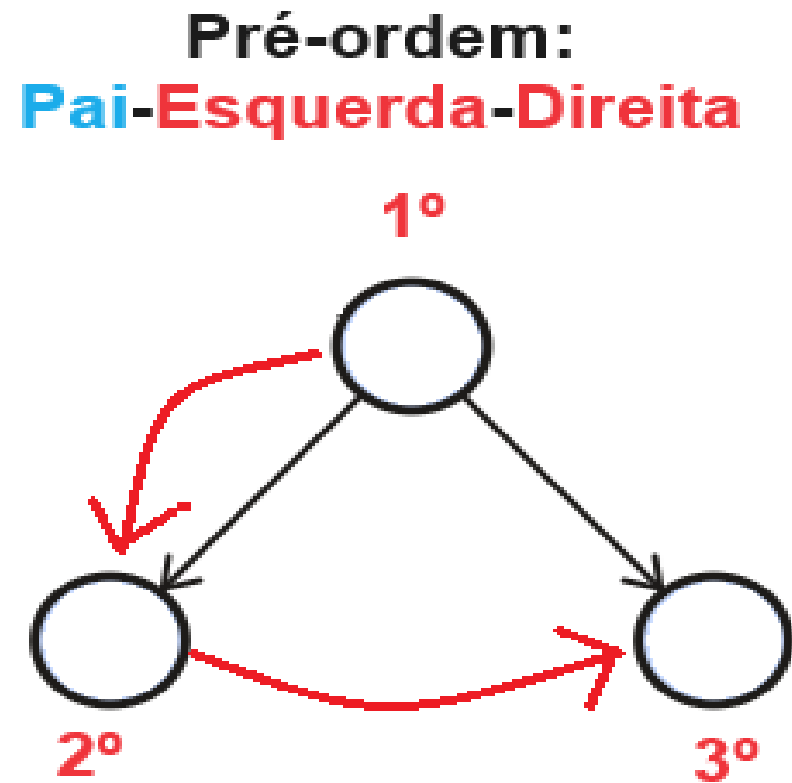
# Árvores Binária: PERCORRENDO UMA ÁRVORE

Há basicamente 3 formas de navegar por uma árvore binária: pre-order, in-order e post-order.

- **Pre-order:** visitamos primeiramente a raiz da árvore, depois recursivamente a subárvore a esquerda e finalmente recursivamente a subárvore a direita.
- **In-order:** visitamos recursivamente a subárvore a esquerda, depois passamos pela raiz da árvore e por fim visitamos recursivamente a subárvore a direita.
- **Post-order:** visitamos recursivamente a subárvore a esquerda, depois visitamos recursivamente a subárvore a direita e por fim passamos pela raiz

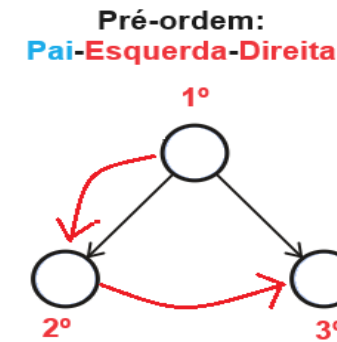
# Árvores Binária: PERCORRENDO UMA ÁRVORE

**Pré-ordem:** No percurso pré-ordem, ou “pre order”, os nós são visitados na seguinte ordem: primeiro o nó pai, em seguida o filho esquerdo e, por último, o filho direito.





# Árvores Binária: PRÉ-ORDEM

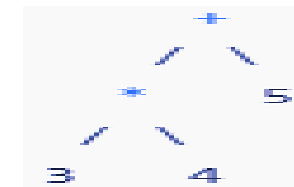


## Possibilidades:

**Construção de uma cópia da árvore:** O percurso pré-ordem é útil para construir uma cópia exata da árvore ou para clonar a estrutura da árvore. Isso ocorre porque ele começa pela raiz, permitindo que a cópia seja construída com o mesmo formato.

**Serialização de árvores para armazenamento ou transmissão:** Ao armazenar ou transmitir a estrutura de uma árvore binária (por exemplo, em arquivos ou para transmitir via rede), o percurso pré-ordem pode ser usado para serializar a árvore de forma compacta. Ele permite reconstruir a árvore na mesma ordem em que foi percorrida.

**Avaliação de expressões prefixas (Polonesa):** No processamento de árvores de expressão matemática, em que os operadores estão nos nós internos e os operandos nas folhas, o percurso pré-ordem é utilizado para avaliar expressões no formato prefixo (notação polonesa). Por exemplo, a expressão **+ \* 3 4 5** seria percorrida em pré-ordem para calcular o resultado diretamente.

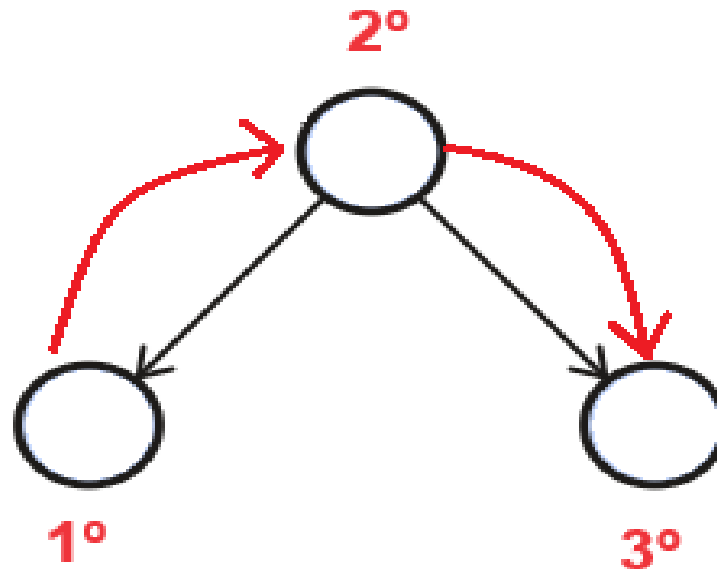


**Implementação de sistemas de arquivos:** Em alguns sistemas de arquivos organizados em árvores, o percurso pré-ordem permite acessar o diretório pai antes de acessar os subdiretórios e arquivos, útil para manipulações que necessitam de referência ao nó pai antes de processar os filhos.

# Árvores Binária: PERCORRENDO UMA ÁRVORE

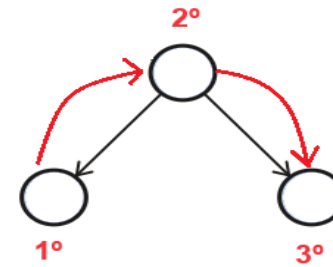
**Em ordem (Simétrico):** No percurso em ordem, também conhecido como “in order”, a árvore é percorrida de forma que os nós sejam visitados na seguinte ordem: primeiro o filho esquerdo, depois o nó pai e, por fim, o filho direito.

**Em ordem:**  
**Esquerda-Pai-Direita**



# Árvores Binária: EM-ORDEM

Em ordem:  
Esquerda-Pai-Direita



## Possibilidades:

**Árvore Binária de Pesquisa (BST) / Extração de Dados Ordenados:** Visitar os nós em ordem crescente dos seus valores.

**Impressão de Dados para Relatório (ordenado):** Muitas aplicações requerem a geração de relatórios onde os dados precisam estar ordenados. O percurso em ordem facilita a obtenção desses dados de forma estruturada diretamente da árvore (extração de dados em ordem cronológica)

**Conversão de Árvore em Lista Ordenada:** O percurso em ordem pode ser usado para converter uma árvore em uma lista ou array ordenado. Isso é útil quando se precisa manipular os dados em estruturas que requerem ordenação ou acesso sequencial.

**Recuperação de Dados em Sistemas de Arquivos:** Em alguns sistemas de arquivos que utilizam árvores binárias para armazenar referências a arquivos e diretórios, o percurso em ordem pode ser usado para listar os arquivos em ordem alfabética ou de acordo com algum atributo.

**Implementação de Algoritmos em Compiladores:** Análise Sintática e Semântica: Em compiladores, árvores de sintaxe abstrata (AST) podem ser percorridas em ordem para realizar análises que dependem da ordem das operações ou declarações no código fonte.

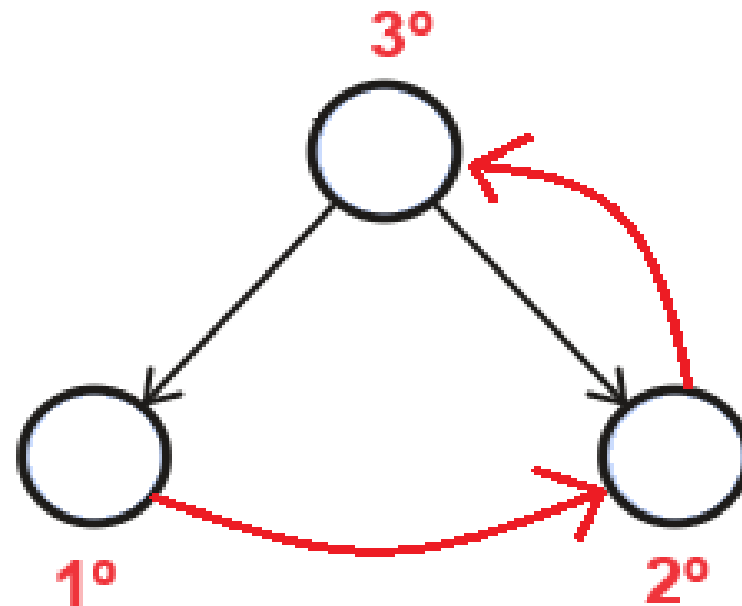
**Expressões Matemáticas Infixas:** O percurso em ordem é análogo à leitura de expressões matemáticas na notação infixa, que é a forma como normalmente escrevemos expressões (por exemplo,  $a + b$ ). Isso é útil em interpretadores ou calculadoras que precisam converter árvores de expressão em notação infixa legível.

**Inteligência Artificial e Jogos:** Árvores de Decisão: Em algumas aplicações de inteligência artificial, árvores de decisão podem ser percorridas em ordem para avaliar condições ou estados que dependem de uma sequência específica. Ex: Xadrez

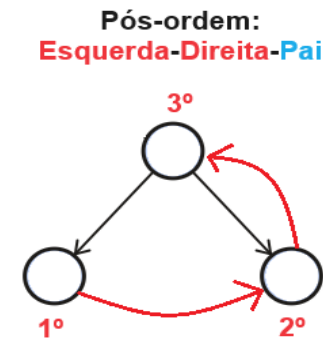
# Árvores Binária: PERCORRENDO UMA ÁRVORE

**Pós-ordem:** O percurso pós-ordem, ou “post order”, envolve a visita aos nós na seguinte ordem: primeiro o filho esquerdo, depois o filho direito e, por último, o nó pai.

**Pós-ordem:**  
**Esquerda-Direita-Pai**



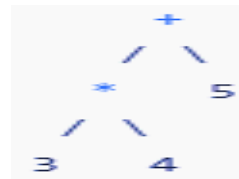
# Árvores Binária: PÓS-ORDEM



## Possibilidades:

**Apagar ou liberar memória da árvore:** Em linguagens que exigem o gerenciamento manual de memória, o percurso pós-ordem é ideal para liberar memória. Ao visitar primeiro as subárvores antes de visitar o nó pai, esse percurso garante que todos os nós filhos sejam deletados antes do pai, evitando problemas de referência pendente.

**Avaliação de expressões infixas (Posfixa):** Em árvores de expressão matemática, o percurso pós-ordem é usado para calcular o valor de expressões posfixas (**notação polonesa reversa**). Por exemplo, para a expressão **3 4 \* 5 +**, o percurso pós-ordem ajuda a avaliar a expressão sem necessidade de parênteses, calculando as operações na ordem correta.



**Cálculo de tamanhos de diretórios ou subárvores:** Para obter o tamanho de diretórios e subdiretórios, o percurso pós-ordem é útil, pois permite somar o tamanho de cada subdiretório antes de somar o diretório pai. Isso é especialmente útil em sistemas de arquivos ou em estruturas onde o cálculo cumulativo precisa ser feito de baixo para cima.

**Geração de código em compiladores:** No contexto de compiladores, o percurso pós-ordem é usado para gerar código a partir de uma árvore de sintaxe abstrata (AST). Nesse caso, o compilador percorre a árvore de expressão, processando primeiro os operandos e, em seguida, os operadores, facilitando a geração de código na ordem correta.



# Árvores Binária: PERCORRENDO UMA ÁRVORE

A dica pra entender rapidamente esses métodos de percursos em árvores binárias é seguir essas duas regras:

- O nó filho da esquerda é **sempre** visitado antes que o da direita:
- O nome do método se refere ao momento em que o nó pai é visitado, ou seja:
  - **Pré-ordem**: o pai é visitado antes dos filhos, ou seja, Pai-Esquerda-Direita;
  - **Em ordem**: o pai é visitado entre os filhos, ou seja, Esquerda-Pai-Direita;
  - **Pós-ordem**: o pai é visitado após os filhos, ou seja, Esquerda-Direita-Pai;

**Pré-ordem:** Pai - Esquerda - Direita

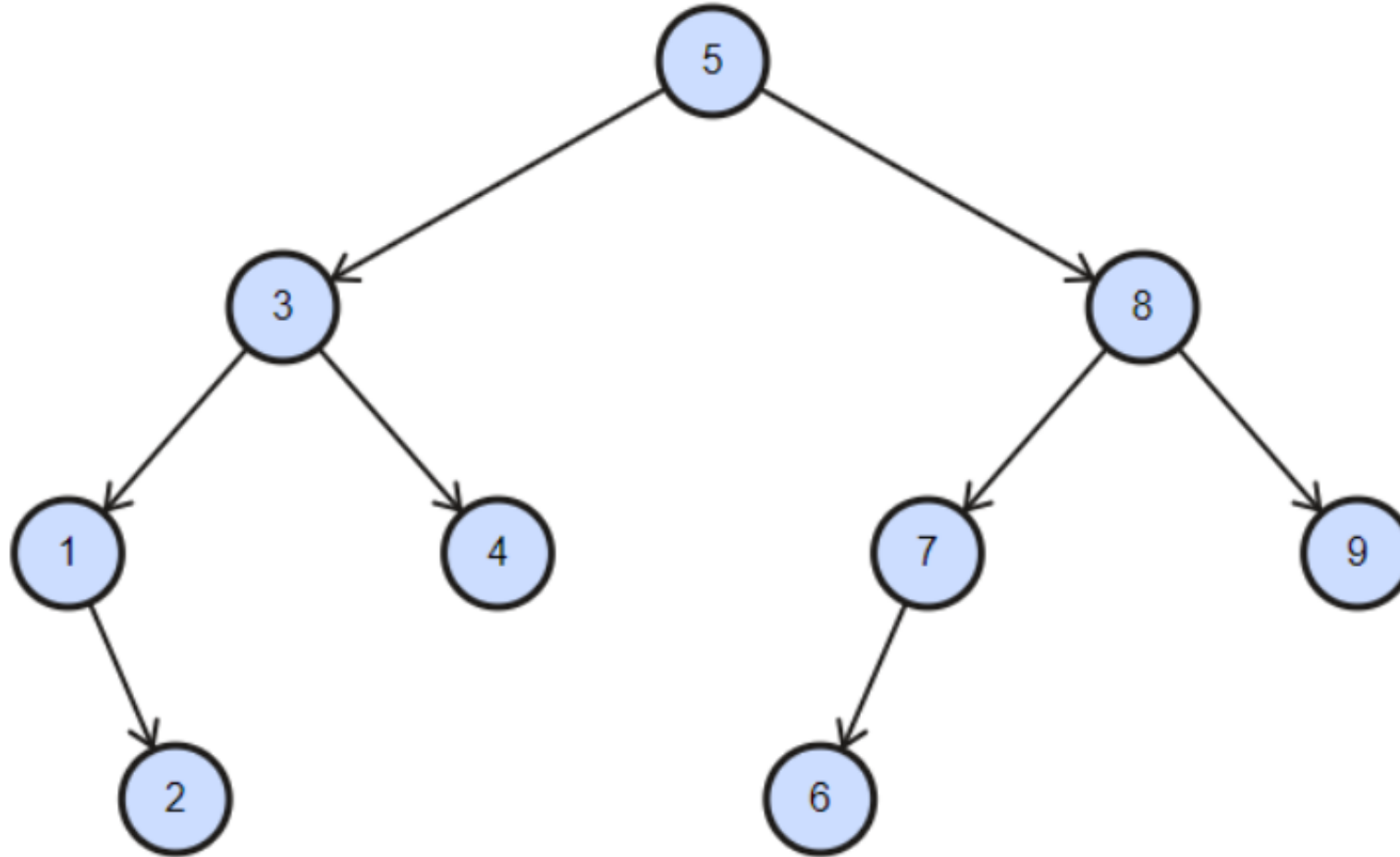
**Em ordem:** Esquerda - Pai - Direita

**Pós-ordem:** Esquerda - Direita - Pai

**Obs:** O nó **esquerdo** é visitado **antes** do nó **direito** em *qualquer método* de percurso em árvore binária.

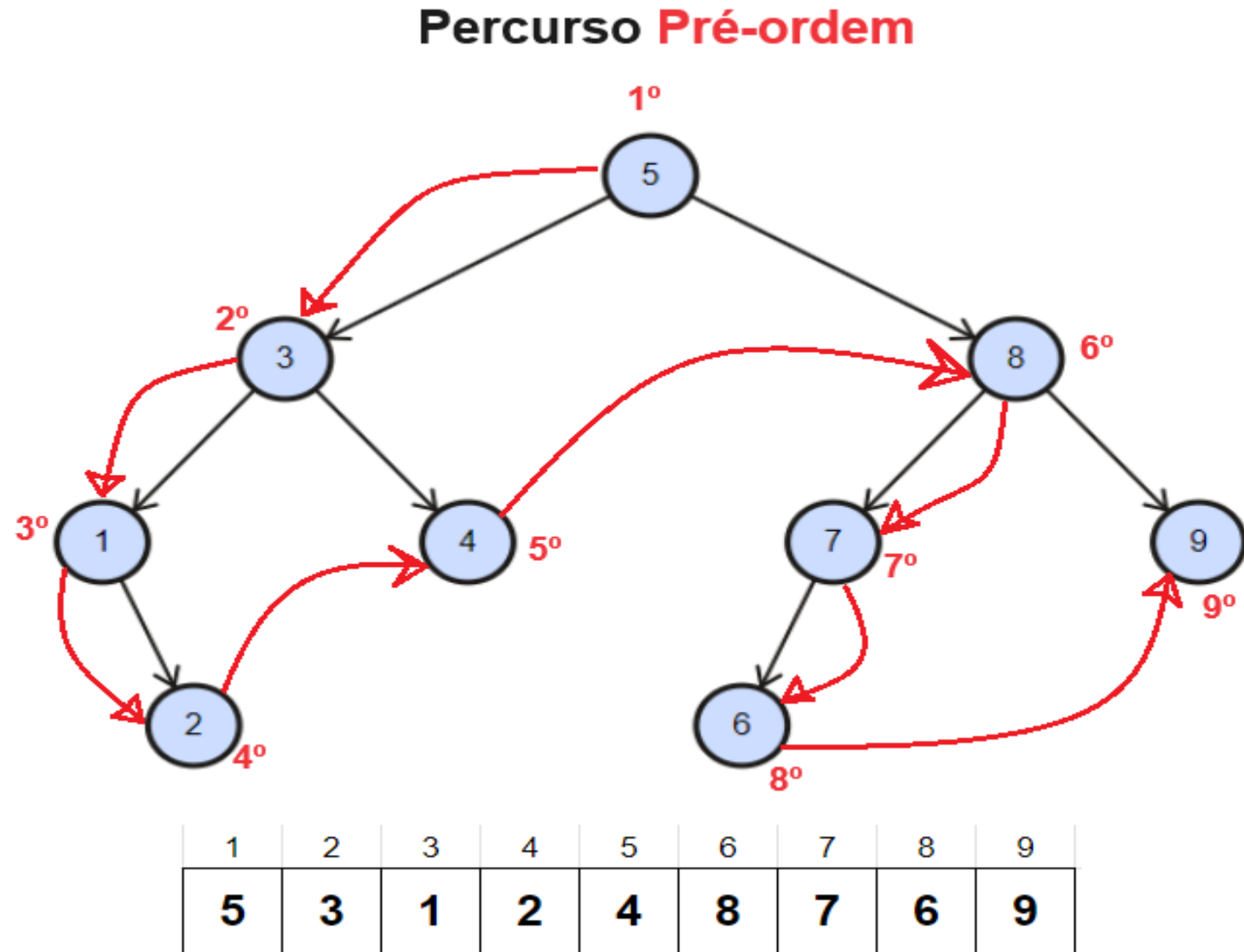
# Árvores Binária: PERCORRENDO UMA ÁRVORE

Exemplo:



# Árvores Binária: PERCORRENDO UMA ÁRVORE

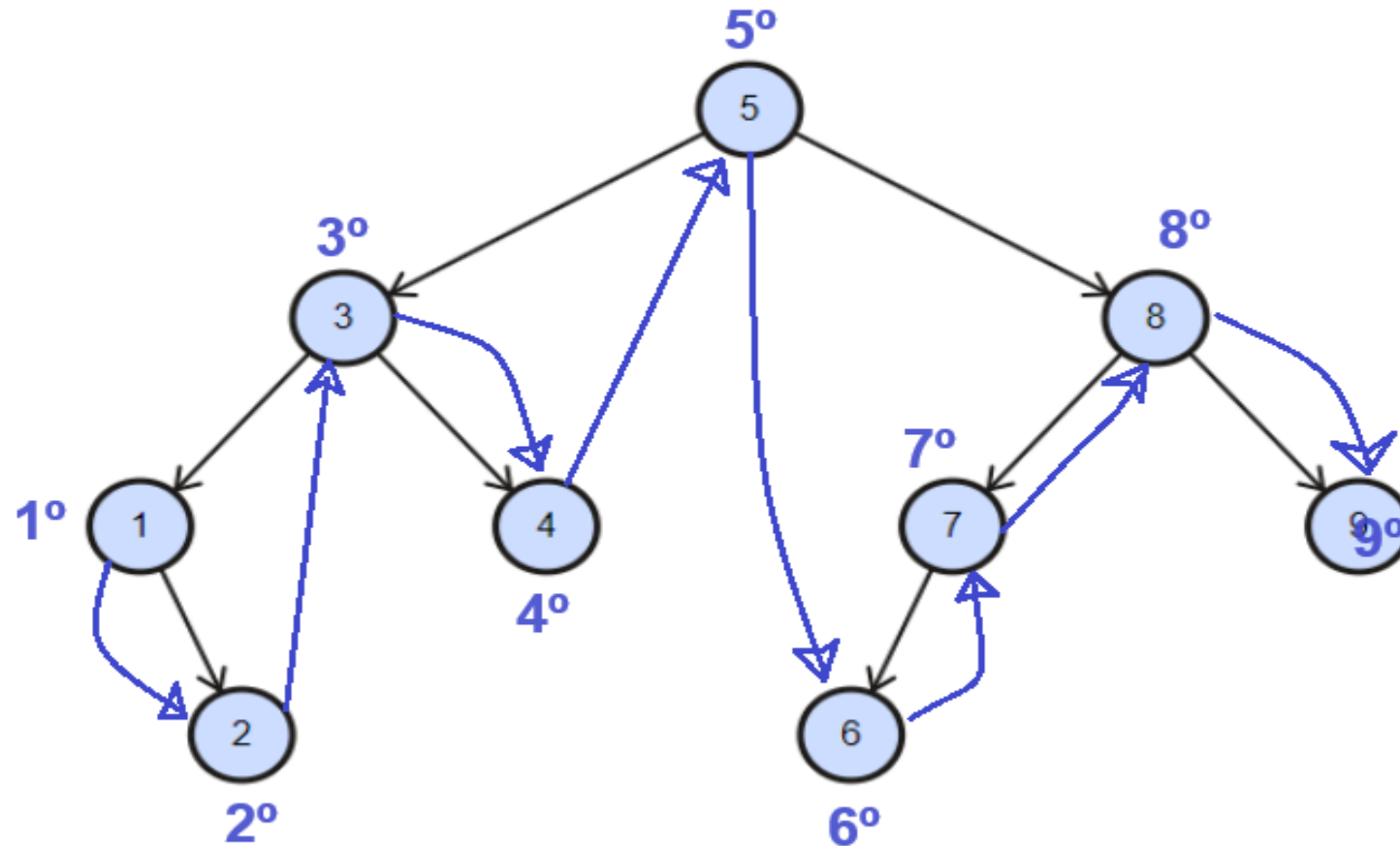
Exemplo:



# Árvores Binária: PERCORRENDO UMA ÁRVORE

Exemplo:

Percurso **Em Ordem**

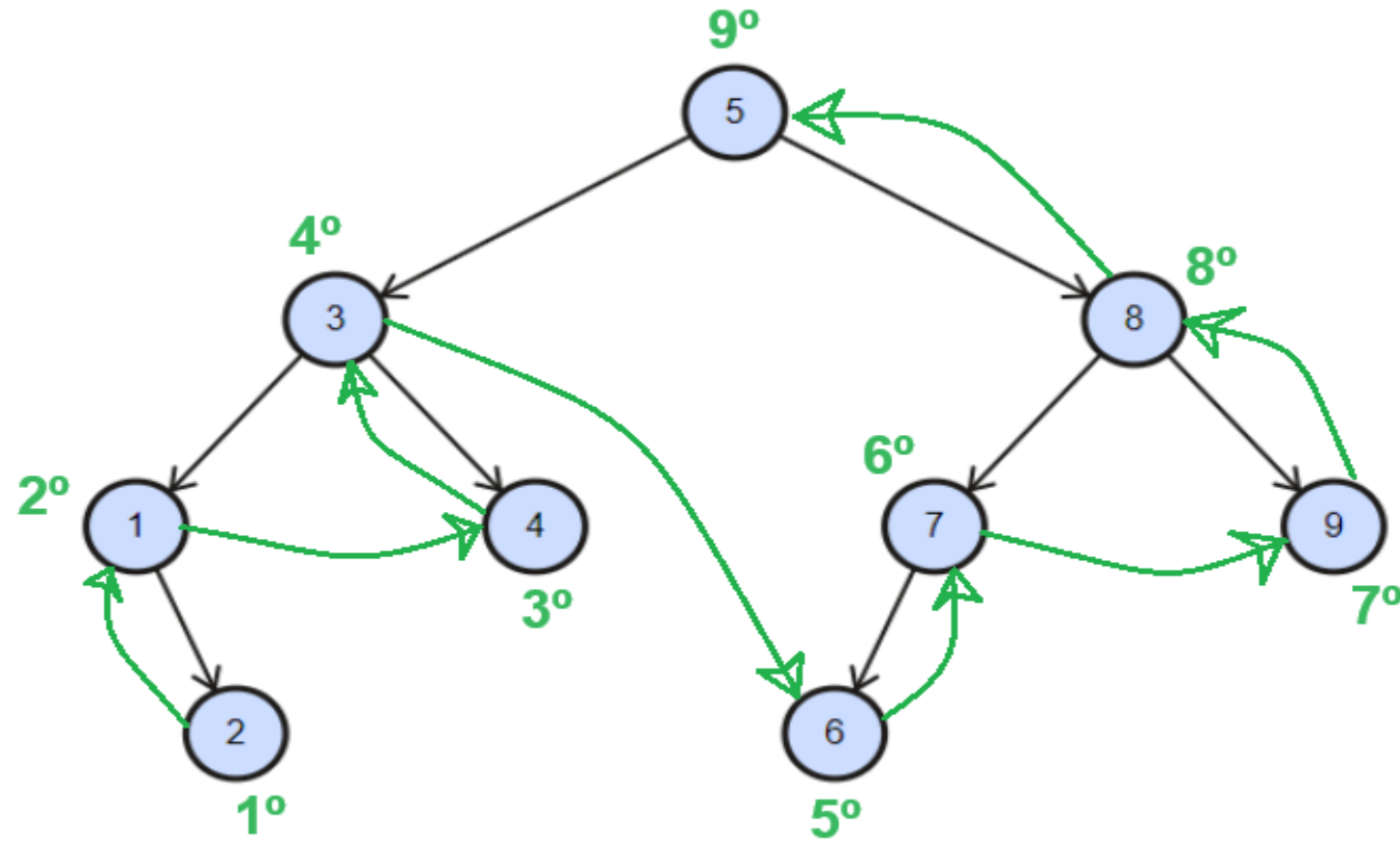


1	2	3	4	5	6	7	8	9
<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>

# Árvores Binária: PERCORRENDO UMA ÁRVORE

Exemplo:

Percurso **Pós-ordem**



1	2	3	4	5	6	7	8	9
<b>2</b>	<b>1</b>	<b>4</b>	<b>3</b>	<b>6</b>	<b>7</b>	<b>9</b>	<b>8</b>	<b>5</b>

# Árvores Binária: PERCORRENDO UMA ÁRVORE (implementação)

# Percorre uma árvore binária em Preorder

```
def preorder(self):  
    # Imprime valor da raiz  
    print(self.key)  
    # Visita subárvore a esquerda  
    if self.left_child:  
        self.left.preorder()  
    # Visita subárvore a direita  
    if self.right_child:  
        self.right.preorder()
```



# Árvores Binária: PERCORRENDO UMA ÁRVORE (implementação)

# Percorre uma árvore binária em Inorder

```
def inorder(self):  
    # Visita subárvore a esquerda  
    if self.left_child:  
        self.left.inorder()  
    # Imprime valor da raiz  
    print(self.key)  
    # Visita subárvore a direita  
    if self.right_child:  
        self.right.inorder()
```

# Árvores Binária: PERCORRENDO UMA ÁRVORE (implementação)

# Percorre uma árvore binária em Postorder

```
def postorder(self):  
    # Visita subárvore a esquerda  
    if self.left_child:  
        self.left.postorder()  
    # Visita subárvore a direita  
    if self.right_child:  
        self.right.postorder()  
    # Imprime valor da raiz  
    print(self.key)
```

# VAMOS PARA A PRÁTICA ?!!!



# Árvores Binária de Busca: LISTAS LIGADAS

Apesar de funcional, a estrutura de dados implementada pela classe **BinaryTree** não é otimizada para busca de elementos no conjunto.

Para essa finalidade, veremos que existe uma classe de árvores mais adequada: **as árvores binárias de busca** (BinarySearchTree).

Uma árvore binária de busca implementa um TDA Map (tipo de dados abstrado – Map), que é uma estrutura que mapeia uma chave a um valor. Neste tipo de estrutura de dados, nós não estamos interessados na localização exata dos elementos na árvore, mas sim em utilizar a estrutura da árvore binária para realizar busca de maneira eficiente.

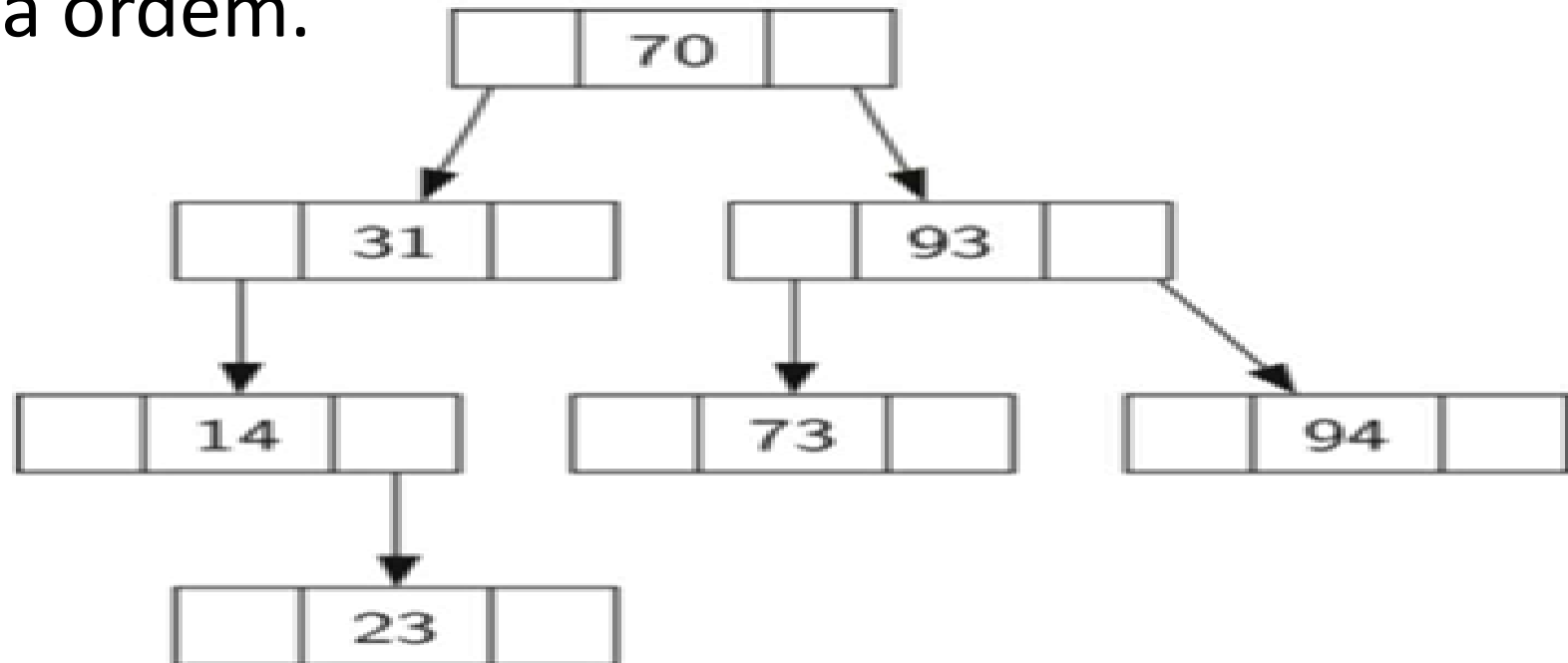
# Árvores Binária de Busca: LISTAS LIGADAS

A seguir apresentamos os principais métodos da classe **BinarySearchTree**.

Método	Operação
Map()	cria um mapeamento vazio
put(key, val)	adiciona um novo par chave-valor ao mapeamento (se chave já existe, atualiza o valor referente a ela)
get(key)	retorna o valor associado a chave
del map[key]	deleta o par chave-valor do mapeamento
len()	retorna o número de pares chave-valor no mapeamento
in	retorna True se chave pertence ao mapeamento (key in map)

# Árvores Binária de Busca: LISTAS LIGADAS

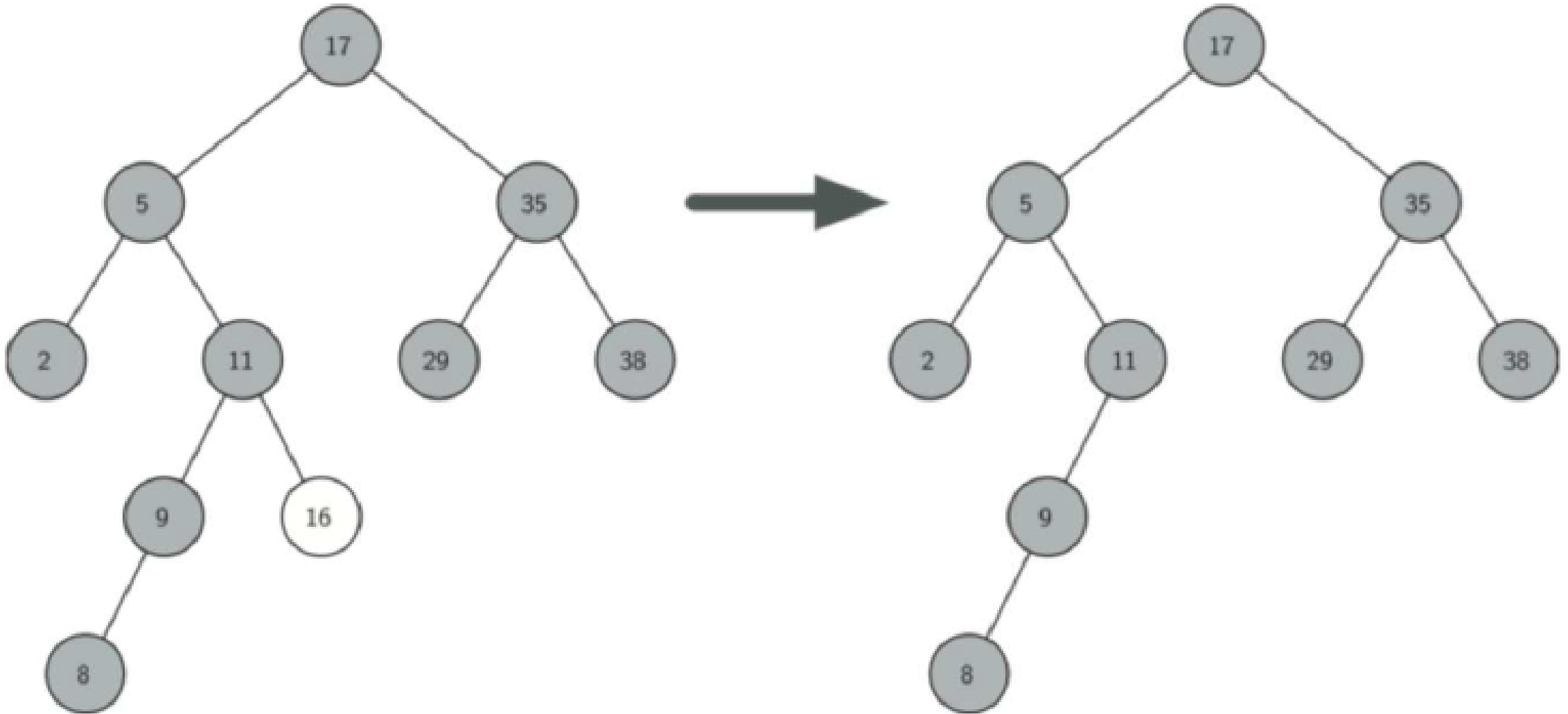
**Propriedade chave:** em uma árvore binária de busca, chaves menores que a chave do nó pai devem estar na subárvore a esquerda e chaves maiores que a chave do nó pai devem estar na subárvore a direita. Por exemplo, suponha que desejamos criar uma árvore binária de busca com os seguintes valores: 70, 31, 93, 94, 14, 23, 73, nessa ordem.





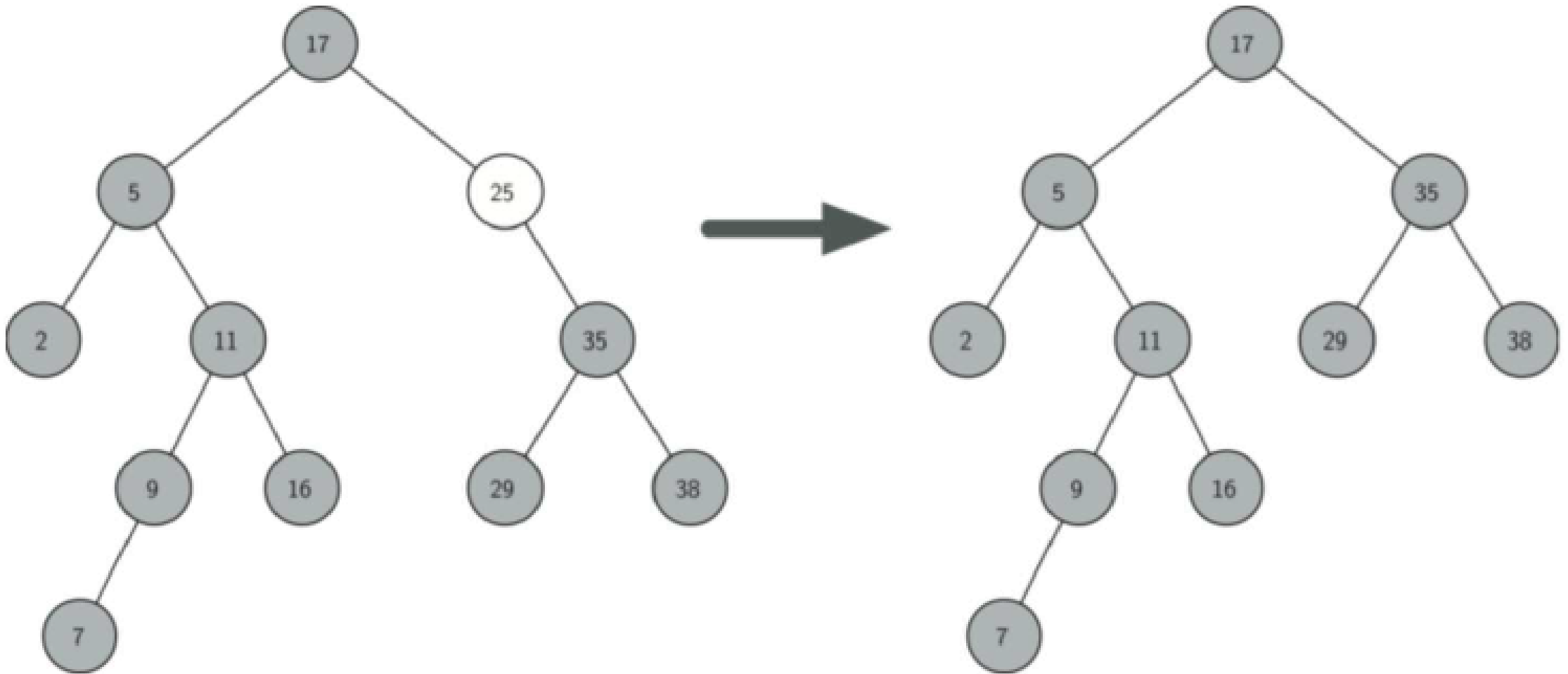
# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS TRIVIAL:



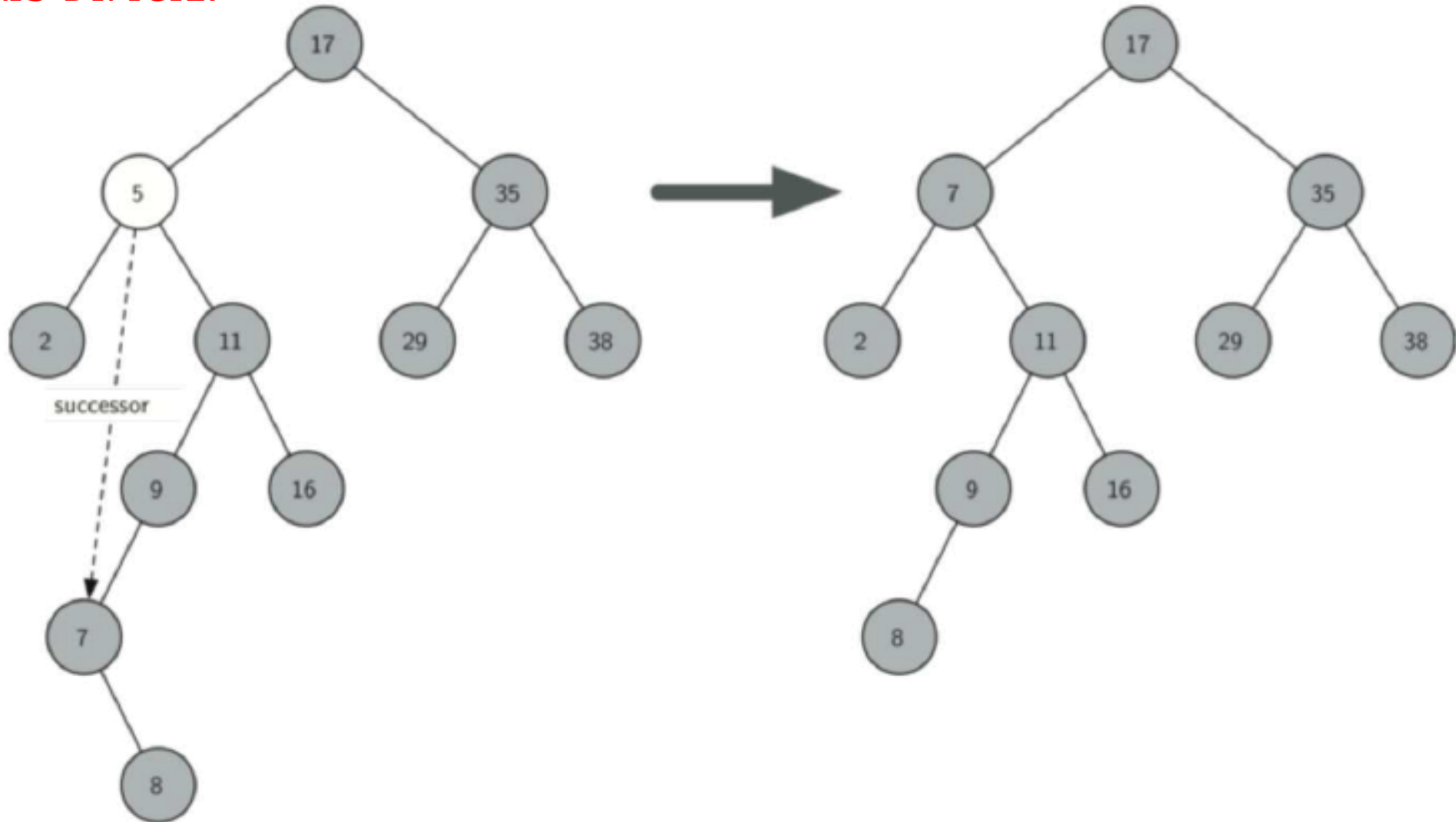
# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO INTERMEDIÁRIA:



# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

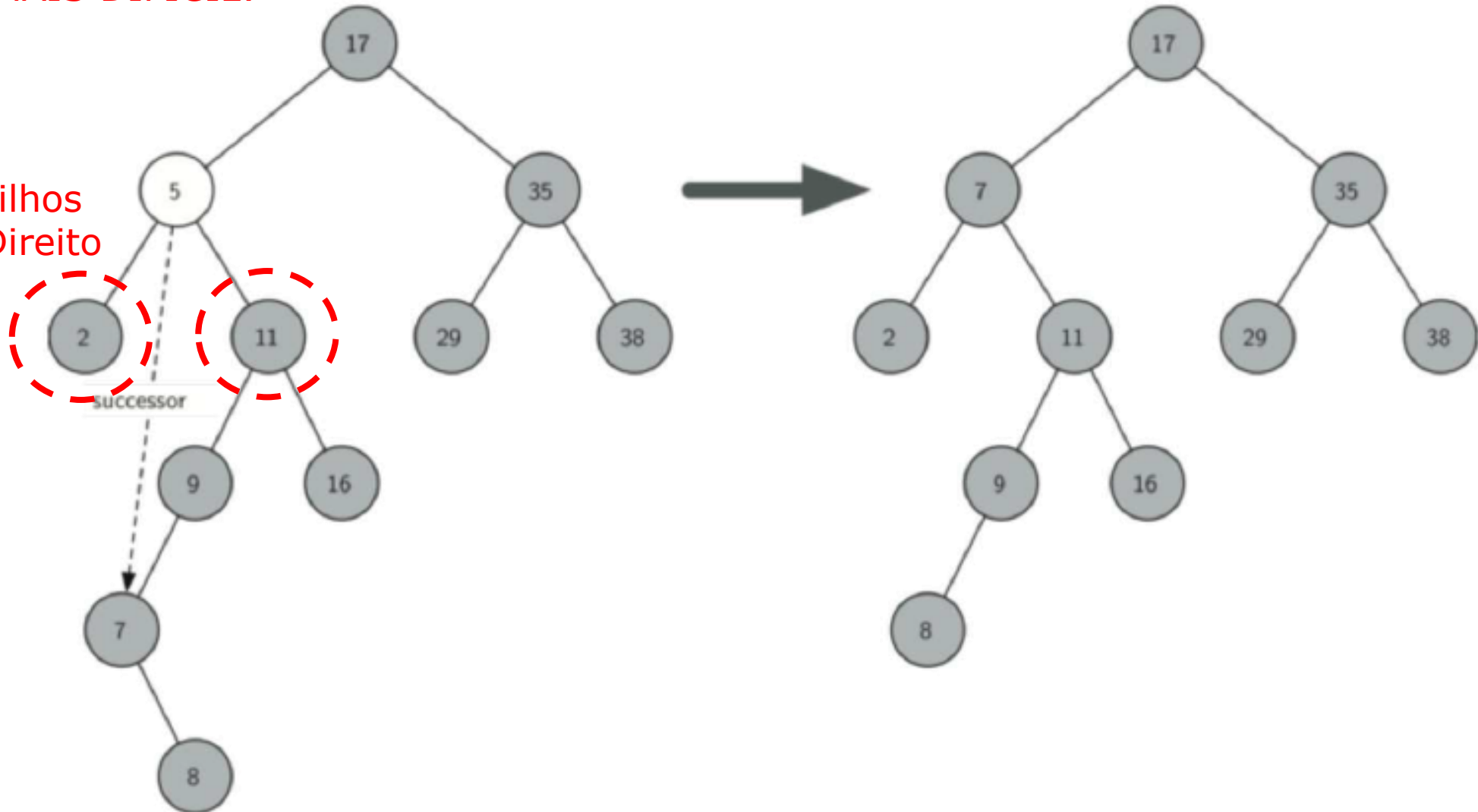


# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

1

Tem ambos filhos  
Esquerdo e Direito

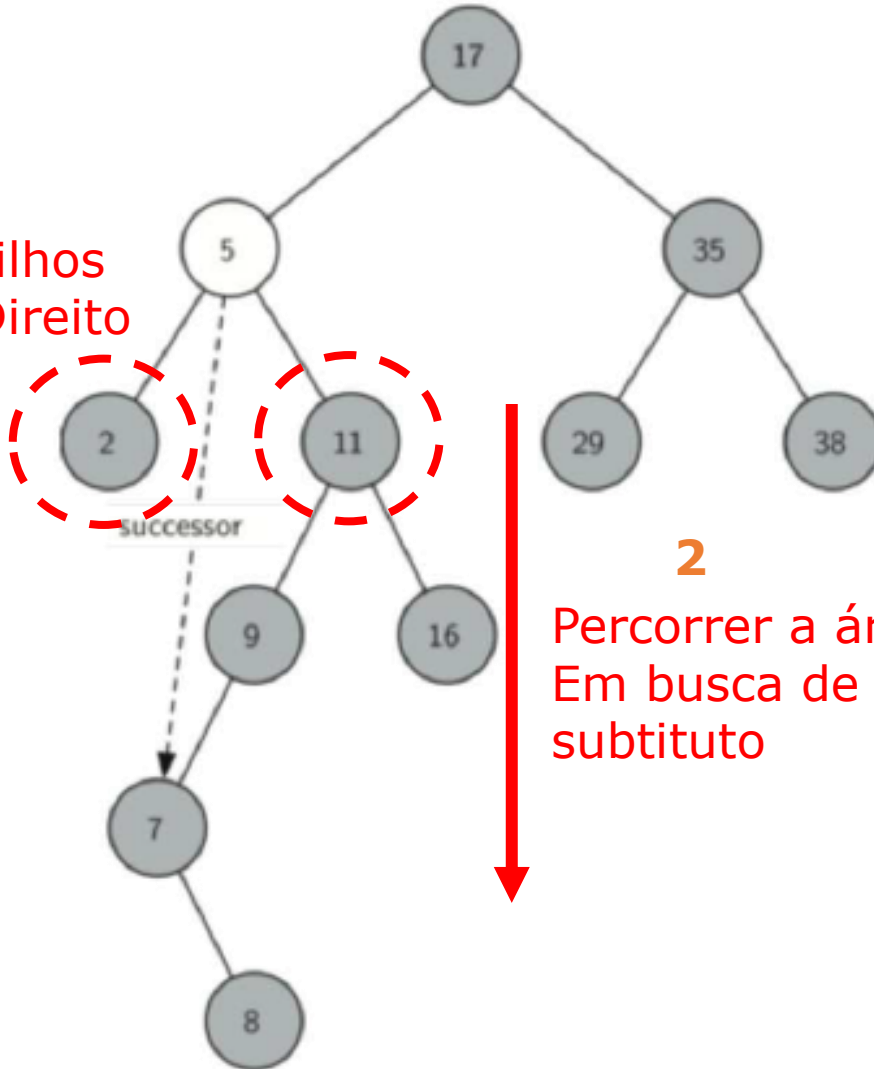


# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

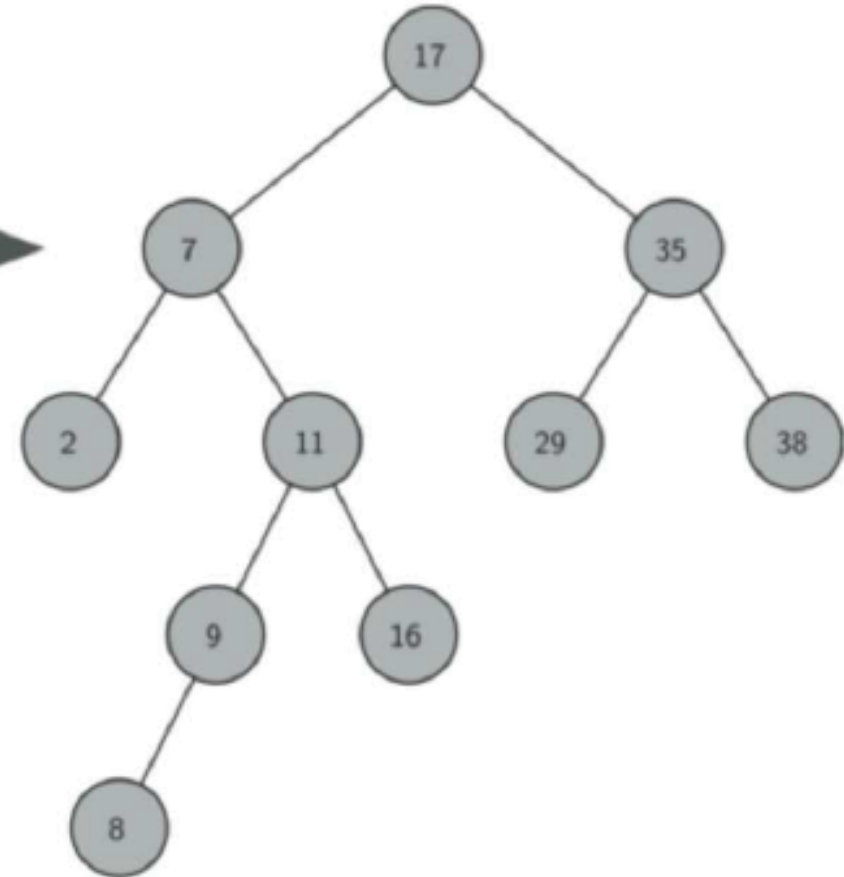
1

Tem ambos filhos  
Esquerdo e Direito



2

Percorrer a árvore  
Em busca de um nó  
substituto

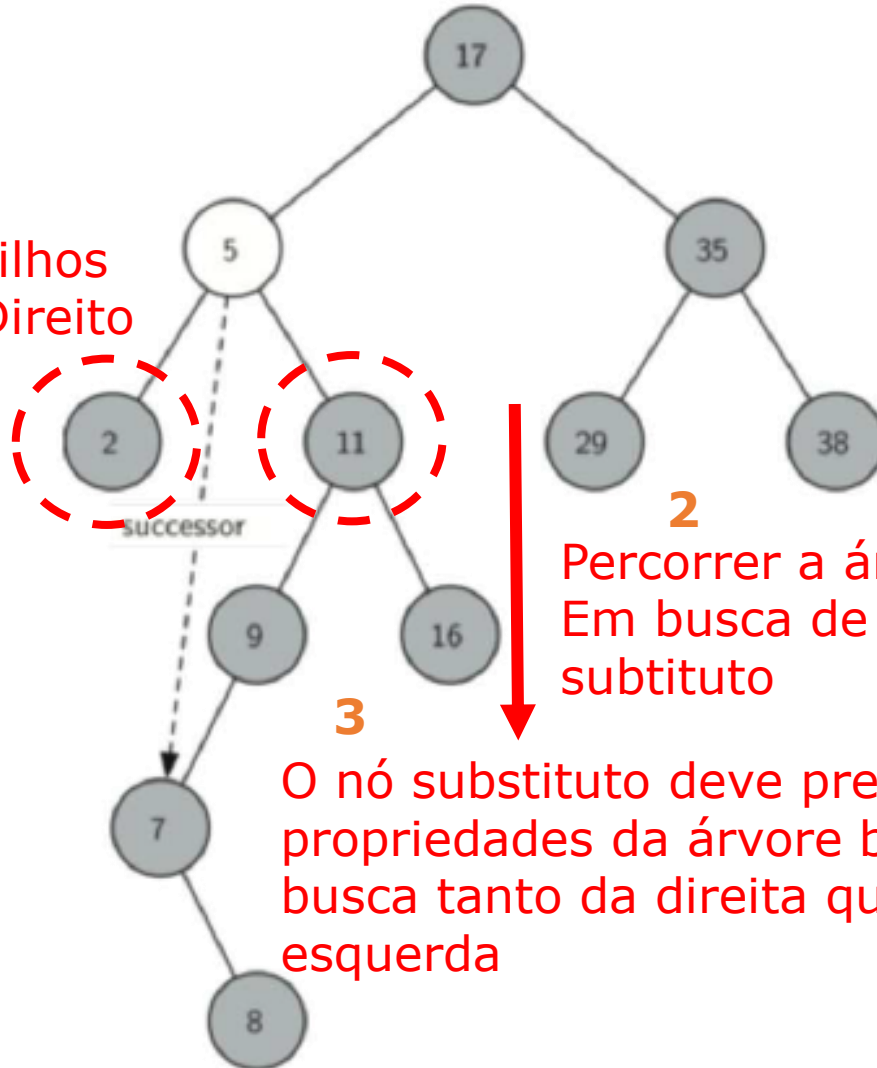


# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

1

Tem ambos filhos  
Esquerdo e Direito

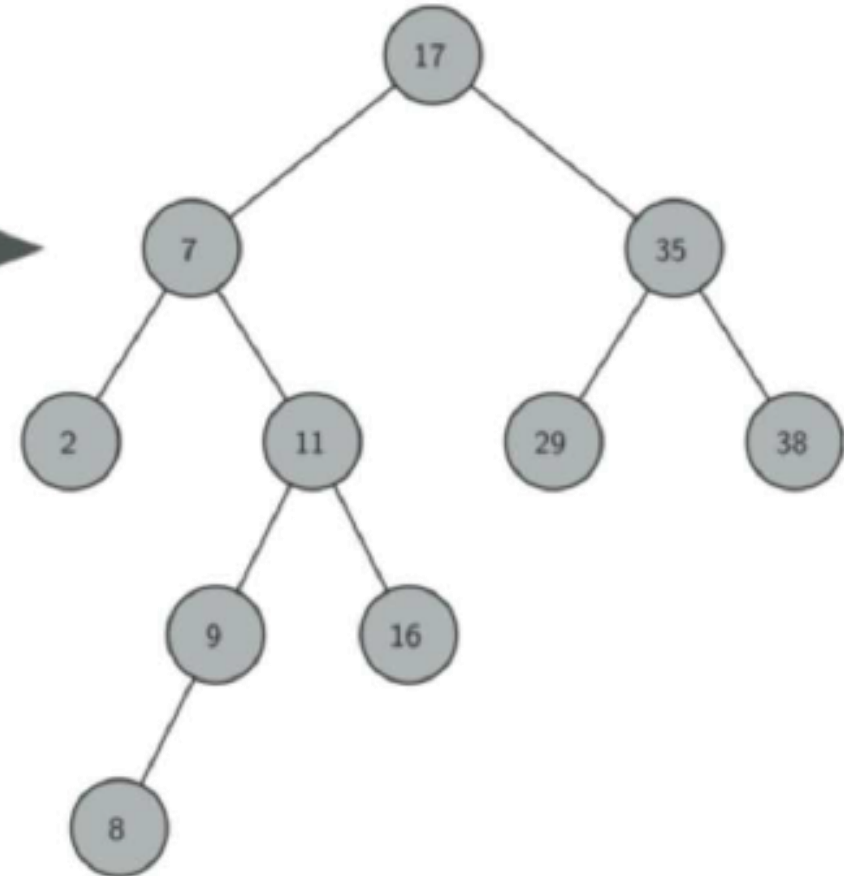


2

Percorrer a árvore  
Em busca de um nó  
substituto

3

O nó substituto deve preservar as  
propriedades da árvore binária de  
busca tanto da direita quanto da  
esquerda





# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

1

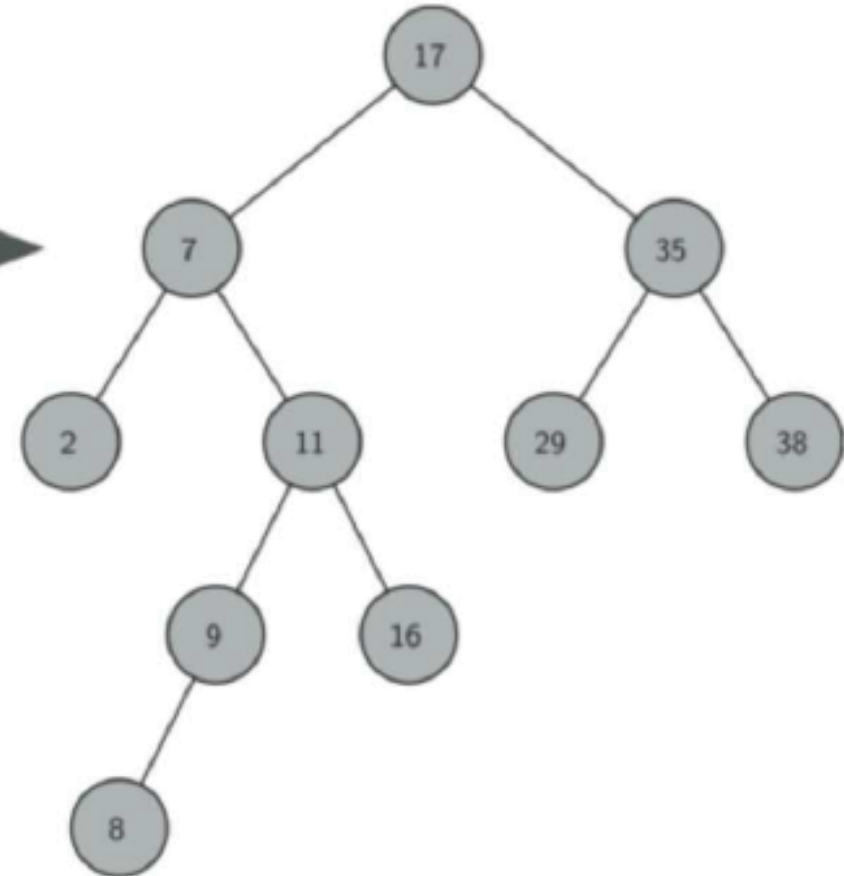
Tem ambos filhos  
Esquerdo e Direito



menor chave que seja maior  
que o nó a ser removido

3  
O nó substituto deve preservar as  
propriedades da árvore binária de  
busca tanto da direita quanto da  
esquerda

4  
O nó que permite preservar essa propriedade é o nó com a menor chave  
que seja maior que a chave do nó corrente (a ser removido).

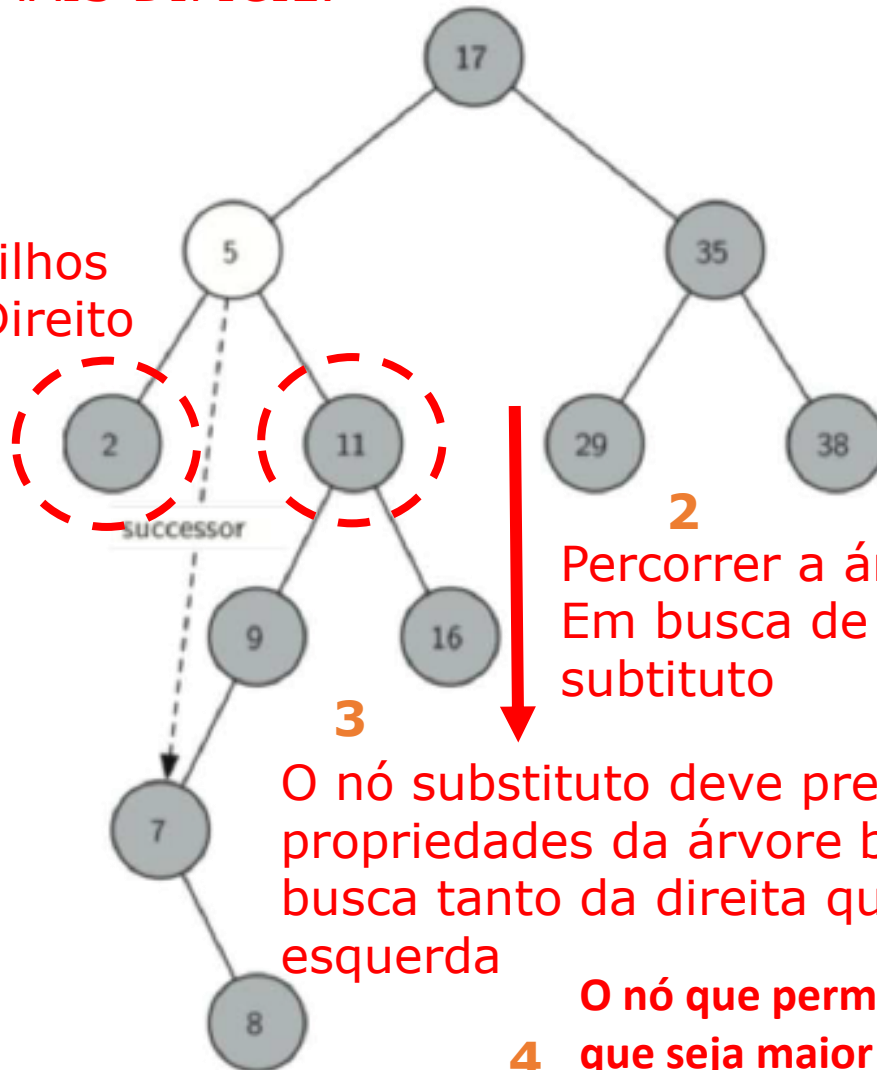


# Árvores Binária de Busca: REMOÇÃO DE NÓ

SITUAÇÃO MAIS DIFÍCIL:

1

Tem ambos filhos  
Esquerdo e Direito



2

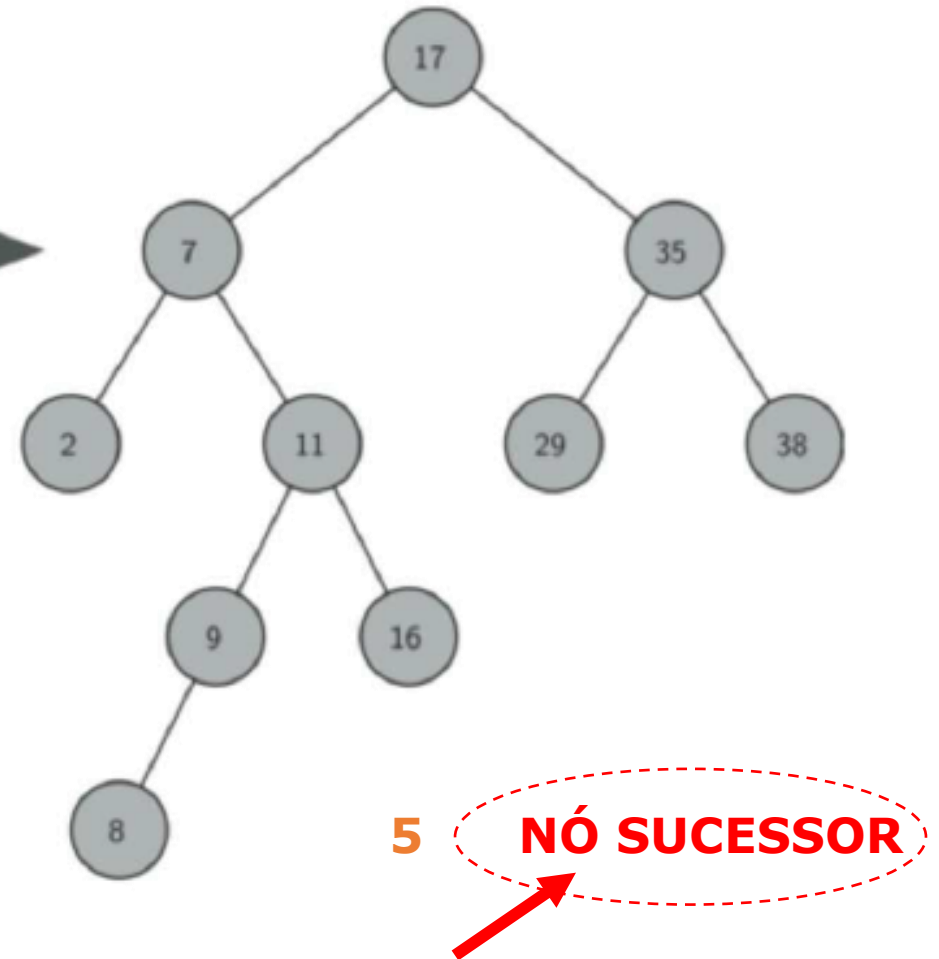
Percorrer a árvore  
Em busca de um nó  
substituto

3

O nó substituto deve preservar as  
propriedades da árvore binária de  
busca tanto da direita quanto da  
esquerda

4

O nó que permite preservar essa propriedade é o nó com a menor chave  
que seja maior que a chave do nó corrente (a ser removido).



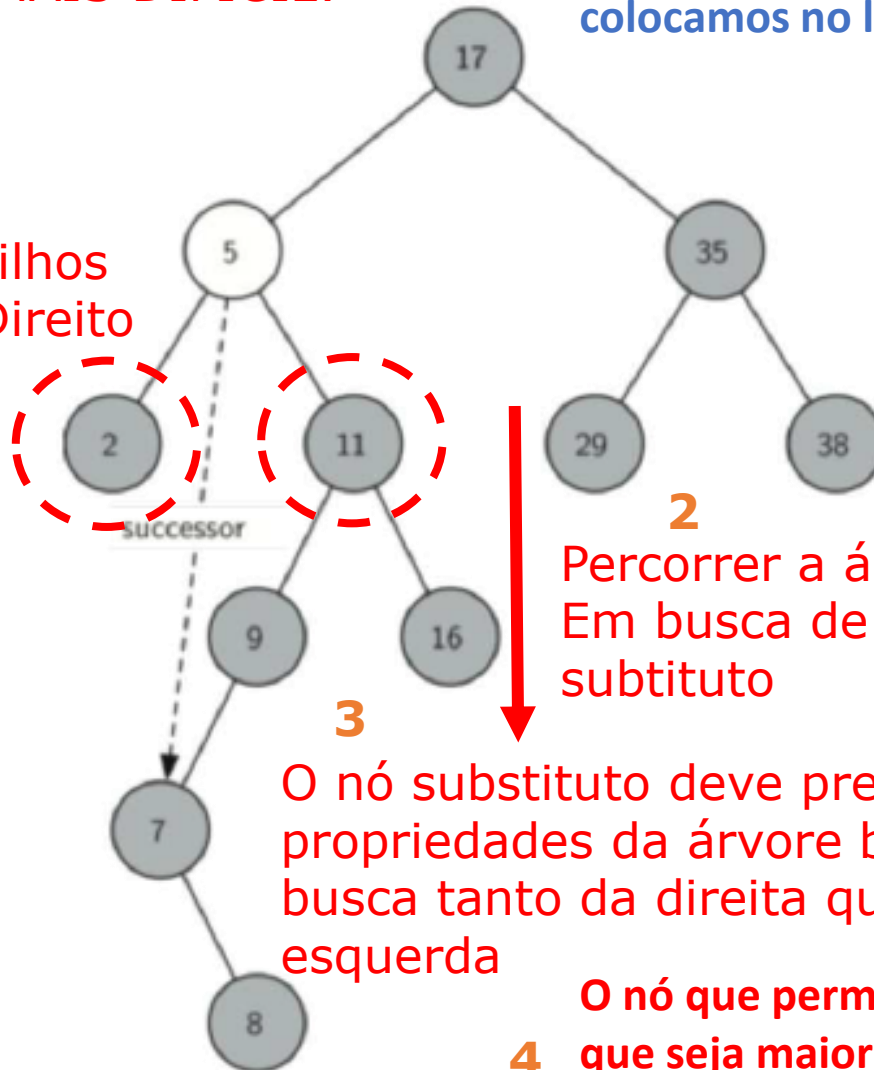
5

**NÓ SUCESSOR**

# Árvores Binária de Busca: REMOÇÃO DE NÓ

## SITUAÇÃO MAIS DIFÍCIL:

**1**  
Tem ambos filhos  
Esquerdo e Direito



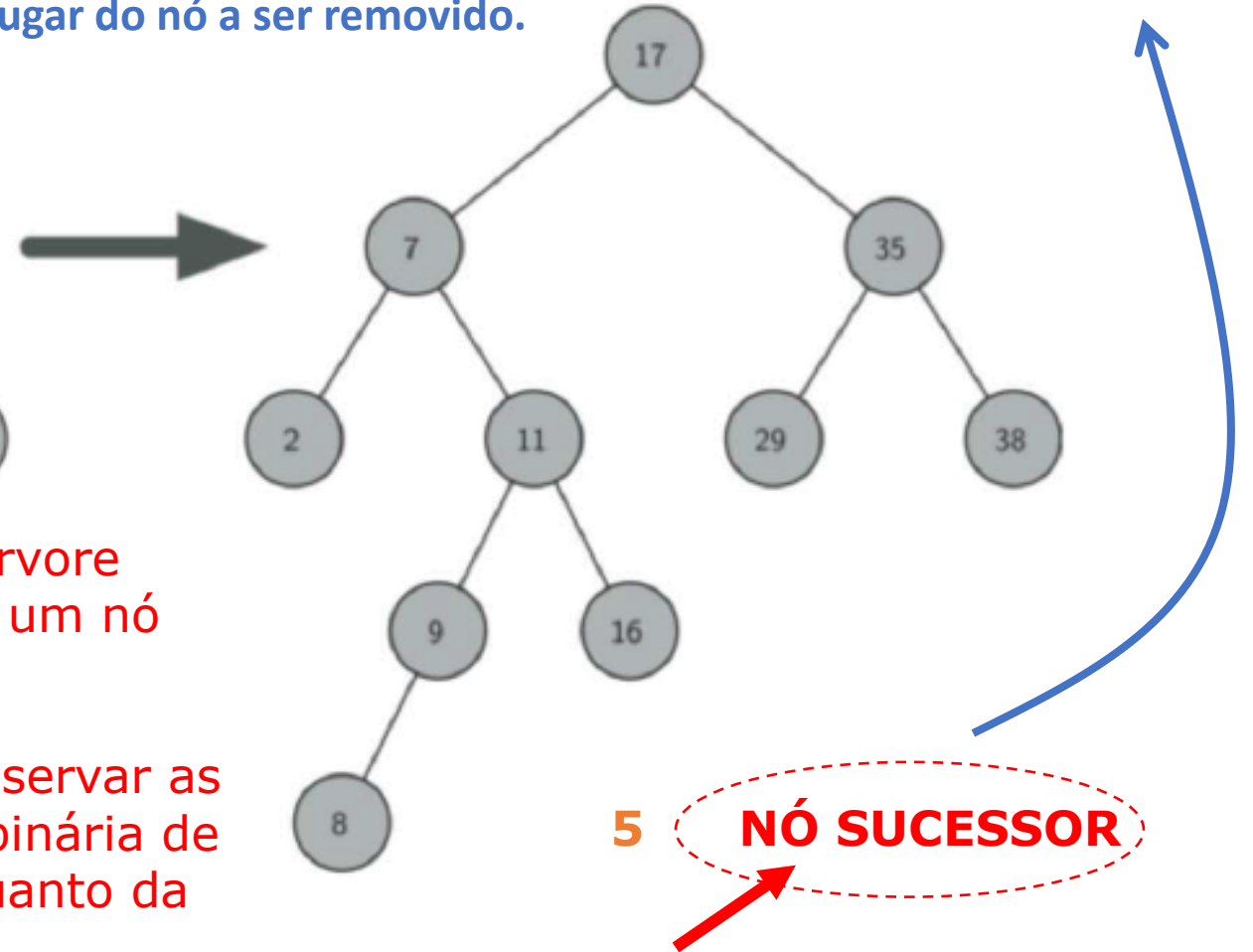
**6** Uma propriedade interessante é que o *successor* tem no máximo um único filho (0 ou 1 filho), de modo que já sabemos como remove-lo utilizando os dois casos discutidos previamente. Após a remoção do *successor*, nós simplesmente o colocamos no lugar do nó a ser removido.

**2**  
Percorrer a árvore  
Em busca de um nó  
substituto

**3**  
O nó substituto deve preservar as  
propriedades da árvore binária de  
busca tanto da direita quanto da  
esquerda

**4** O nó que permite preservar essa propriedade é o nó com a menor chave que seja maior que a chave do nó corrente (a ser removido).

**5** **NÓ SUCESSOR**

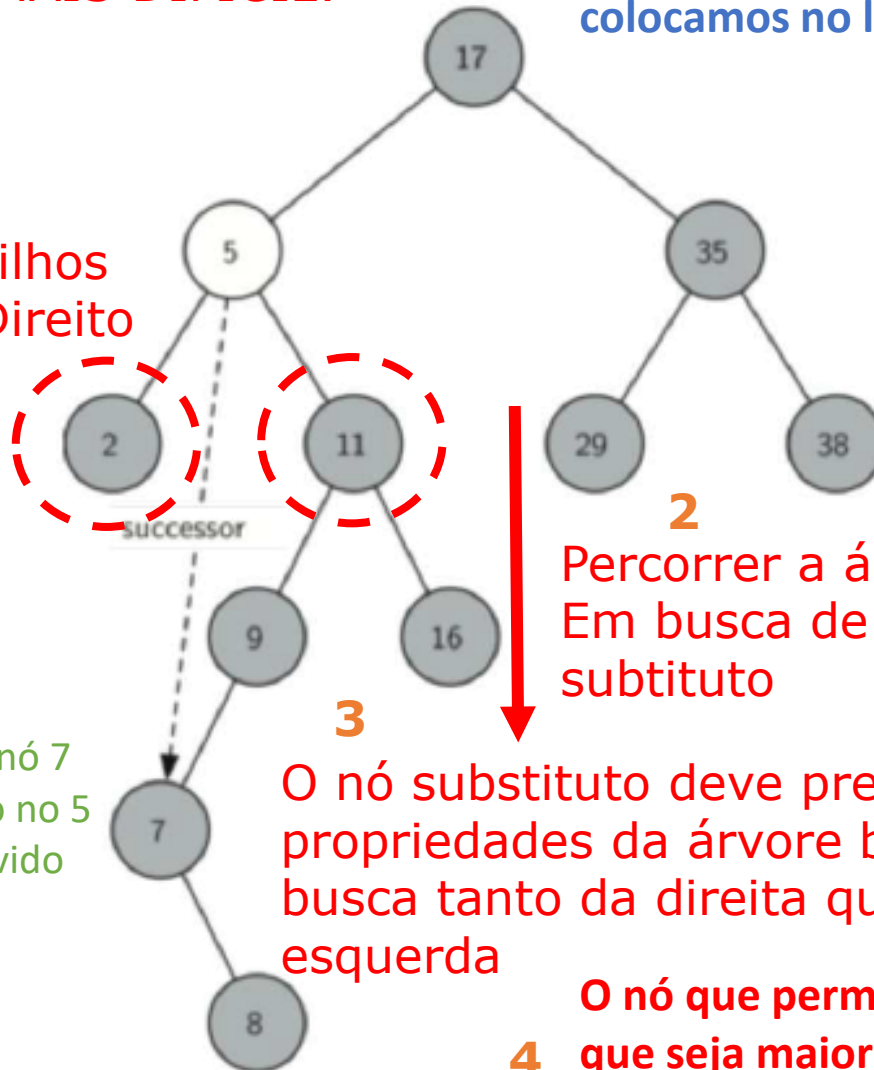


# Árvores Binária de Busca: REMOÇÃO DE NÓ

## SITUAÇÃO MAIS DIFÍCIL:

1

Tem ambos filhos  
Esquerdo e Direito



6

Uma propriedade interessante é que o *successor* tem no máximo um único filho (0 ou 1 filho), de modo que já sabemos como remove-lo utilizando os dois casos discutidos previamente. Após a remoção do *successor*, nós simplesmente o colocamos no lugar do nó a ser removido.

2

Percorrer a árvore  
Em busca de um nó  
substituto

3

O nó substituto deve preservar as  
propriedades da árvore binária de  
busca tanto da direita quanto da  
esquerda

4

O nó que permite preservar essa propriedade é o nó com a menor chave  
que seja maior que a chave do nó corrente (a ser removido).

5

**NÓ SUCESSOR**

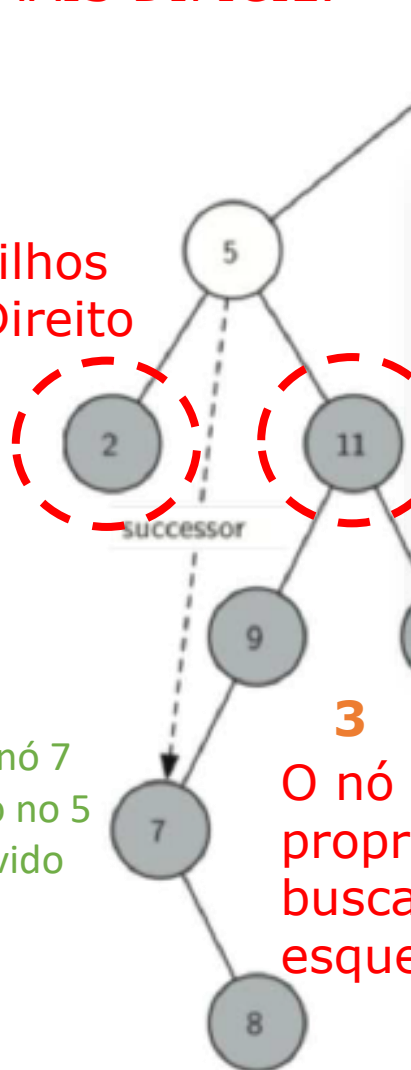
As informações do nó 7  
são copiadas para o nó 5  
e ele então é removido  
da árvore

# Árvores Binária de Busca: REMOÇÃO DE NÓ

## SITUAÇÃO MAIS DIFÍCIL:

1

Tem ambos filhos  
Esquerdo e Direito



6

Uma propriedade interessante é que o *sucessor* tem no máximo um único filho (0 ou 1 filho), de modo que já sabemos como remove-lo utilizando os dois casos discutidos previamente. Após a remoção do *sucessor*, nós simplesmente o colocamos no lugar do nó a ser removido.

```
if no_atual.tem_ambos_os_filhos():  
    sucessor = no_atual.encontrar_sucessor()  
    sucessor.remover_sucessor()  
    no_atual.chave = sucessor.chave  
    no_atual.valor = sucessor.valor
```

3

O nó substituto deve preservar as propriedades da árvore binária de busca tanto da direita quanto da esquerda

4

O nó que permite preservar essa propriedade é o nó com a menor chave que seja maior que a chave do nó corrente (a ser removido).

5

**NÓ SUCESSOR**

As informações do nó 7 são copiadas para o nó 5 e ele então é removido da árvore



# VAMOS PARA A PRÁTICA ?!!!





# DÚVIDAS ENVIADAS...

Árvores...

# Como percorrer uma árvore binária?

- Pra que utilizo cada um dos três tipos: pré-ordem, em-ordem e pós-ordem?

Mayumi

Peterson

Débora

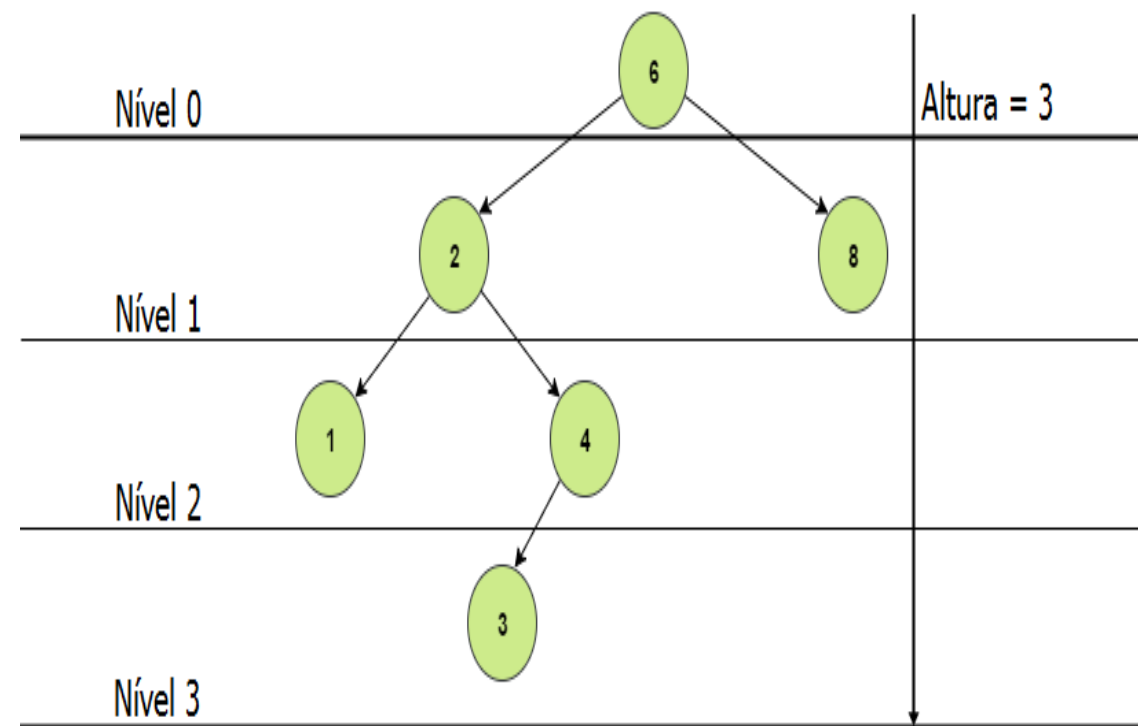
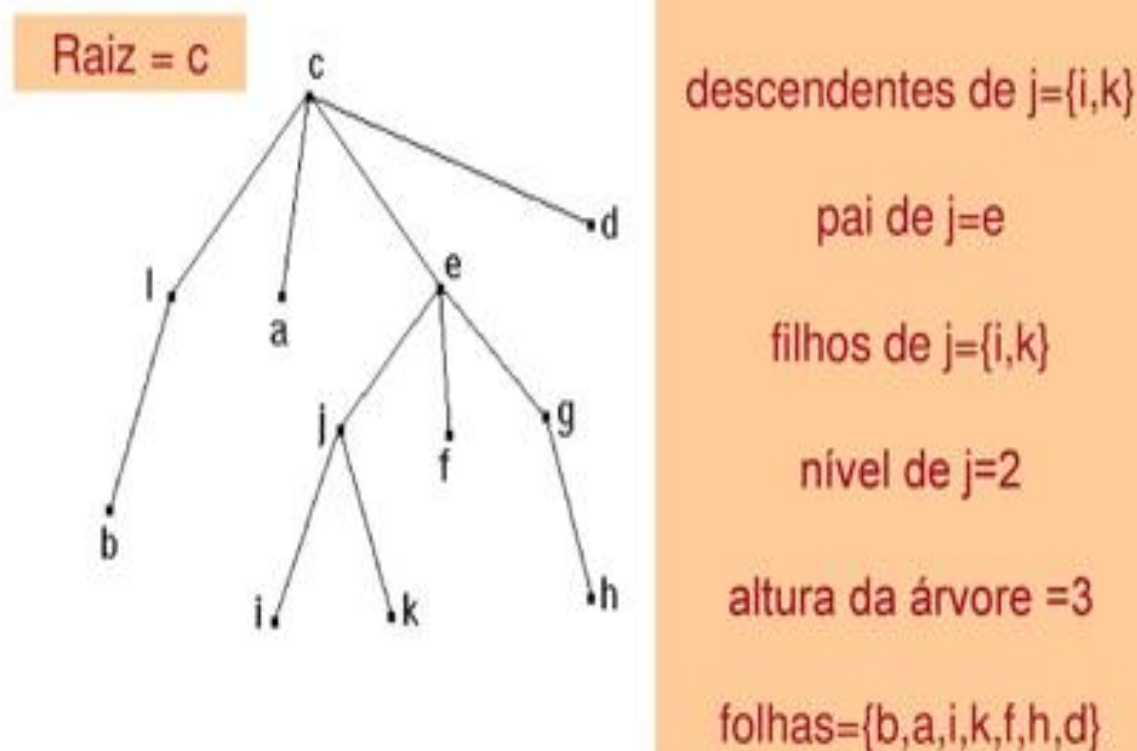
# Como funciona um BST e inserção de números ordenados?

- Como funciona uma BST em relação a organização dos nós?
- Quais as implicações de inserir uma sequência de números ordenados em uma BST?

Vitória

# Qual a diferença entre nível e altura?

Ana Elisa



Inclusão de Ponteiros também para os nós pais? É viável?  
*Quando fazemos chamadas recursivas, os nós anteriores já ficam armazenados em pilhas de execução*

Lucas Camelo

### **DESVANTAGENS DE INCLUSÃO DESSES PONTEIROS:**

**Aumento do Uso de Memória:** Cada nó exigiria um ponteiro adicional, o que aumenta o consumo de memória. Em árvores muito grandes, isso pode se tornar um problema.

**Complexidade na Manutenção:** Inserções, remoções e rotações (como em árvores AVL ou Red-Black) se tornariam mais complexas. Ao realizar essas operações, seria necessário garantir que o ponteiro para o nó pai seja atualizado corretamente, aumentando as chances de erros.

**Pouca Utilidade Prática:** A maioria das operações em árvores binárias (busca, inserção, remoção) não requer conhecimento do nó pai. A árvore é projetada para ser eficiente sem essa informação adicional.

*O uso de recursão (e da pilha de execução) é suficiente para operações que precisam "lembrar" os nós anteriores.*

# As árvores aceitam apenas números ou podemos armazenar outros valores (como strings)?

Eric Rodrigues

```
class BinaryTree:
    # Construtor
    def __init__(self, valor):
        self.key = valor
        self.left_child = None
        self.right_child = None
```

Como é o processo de rotação em uma árvore red/black em uma árvore desbalanceada?

Débora





red  
black  
tree

*Black-Red Trees*

Árvores Rubo-Negras

Árvores Vermelho-Preta

# O que são Árvores Vermelho-Preto?

Uma árvore de busca binária de altura  $h$  pode suportar qualquer das operações básicas de conjuntos dinâmicos — como Search, Predecessor, Successor, Minimum, Maximum, Insert e Delete — no tempo  $O(h)$ .

Assim, as operações de conjuntos são rápidas se a altura da árvore de busca é pequena.

Todavia, se a altura da árvore é grande, a execução dessas operações poderá ser mais lenta do que com uma lista ligada.

**Árvores vermelho-preto são um dos muitos esquemas de árvores de busca que são “balanceadas” de modo a garantir que operações básicas de conjuntos dinâmicos demorem o tempo  $O(\lg n)$  no pior caso.**

# O que são Árvores Vermelho-Preto?

Uma *árvore vermelho-preto* é uma árvore de busca binária com um bit extra de armazenamento por nó: sua *cor* — ela pode ser Vermelha ou Preta.

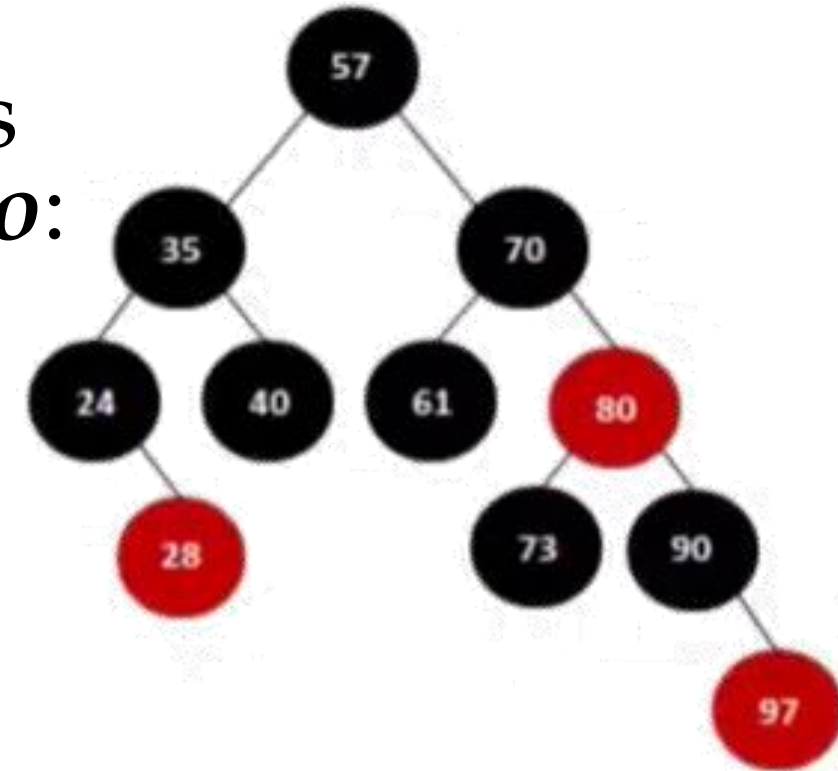
Restringindo as cores dos nós em qualquer caminho simples da raiz até uma folha, as árvores vermelho-preto asseguram que o comprimento de nenhum desses caminhos seja maior que duas vezes o de qualquer outro, de modo que a árvore é aproximadamente *balanceada*.

Cada nó da árvore contém agora os atributos *cor*, *chave*, *esquerda* e *direita*.

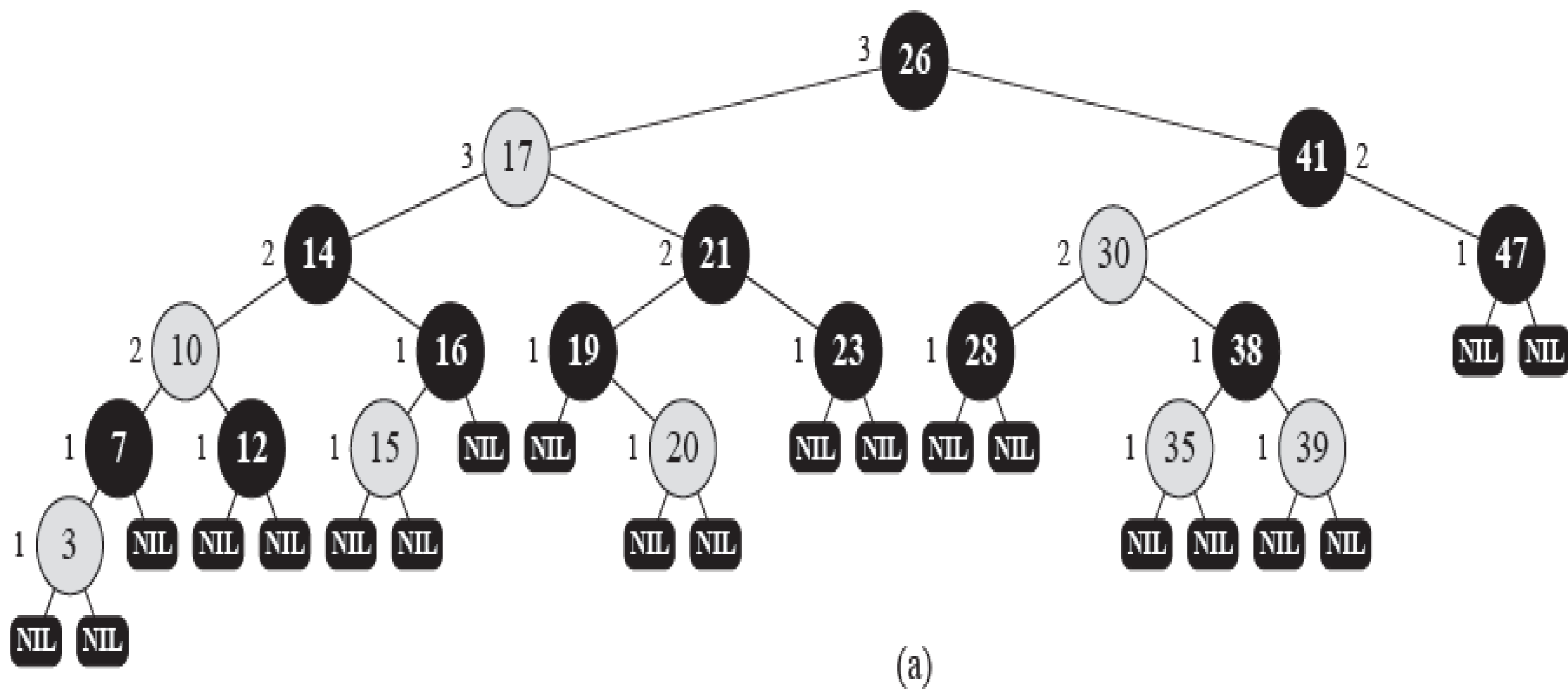
# O que são Árvores Vermelho-Preto?

Uma árvore vermelho-preto é uma árvore de busca binária que satisfaz as seguintes *propriedades vermelho-preto*:

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (nil) é preta.
4. Se um nó é vermelho, então os seus filhos são pretos (não existem nós consecutivos vermelhos)
5. Para cada nó, todos os caminhos simples do nó até folhas descendentes contêm o mesmo número de nós pretos.



# O que são Árvores Vermelho-Preto?



# INSERÇÃO em Árvores Vermelho-Preto?

Cada novo nó inserido, por definição é VERMELHO

A inserção acontece como em qualquer ABB

Após inserção, verifique se as propriedades da AV-P ainda se mantêm.

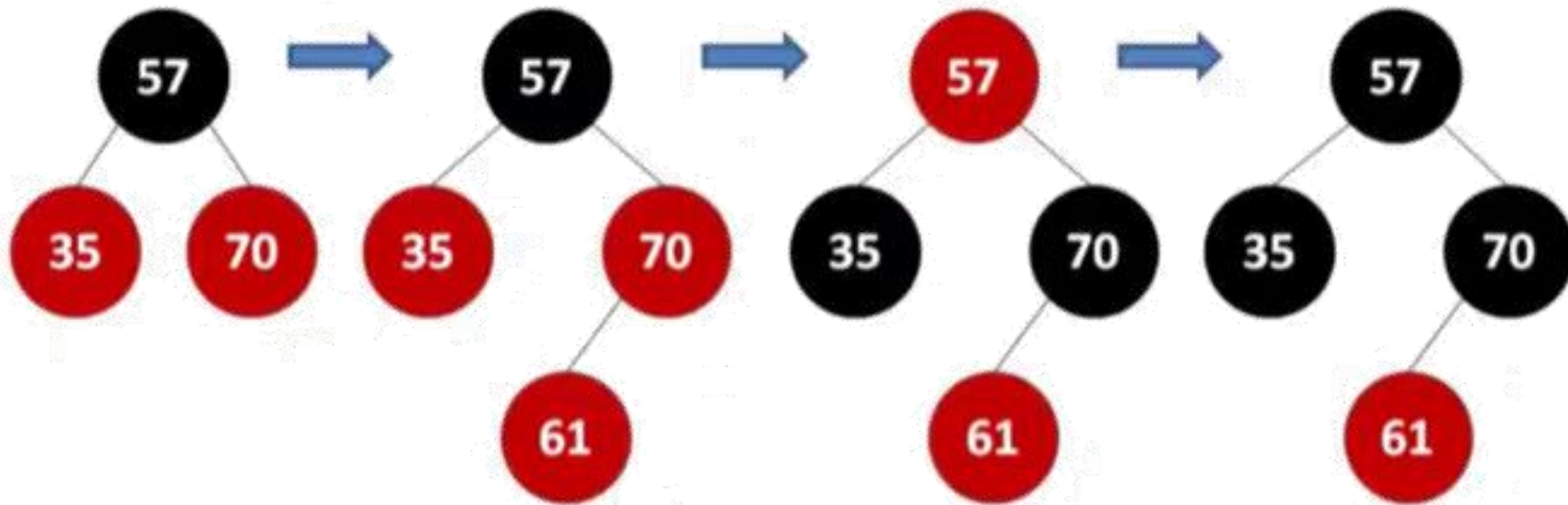
Lembre-se:

- A raiz da árvore é sempre PRETO
- Se o pai do novo nó inserido é PRETO, todas as propriedades se mantêm
- Se o pai do novo nó inserido for VERMELHO, rotações ou alterações de cor precisam ser feitas.

# INSERÇÃO em Árvores Vermelho-Preto

## Caso 1: O pai e o tio do novo nó são vermelhos

Observe a inserção do número 61.



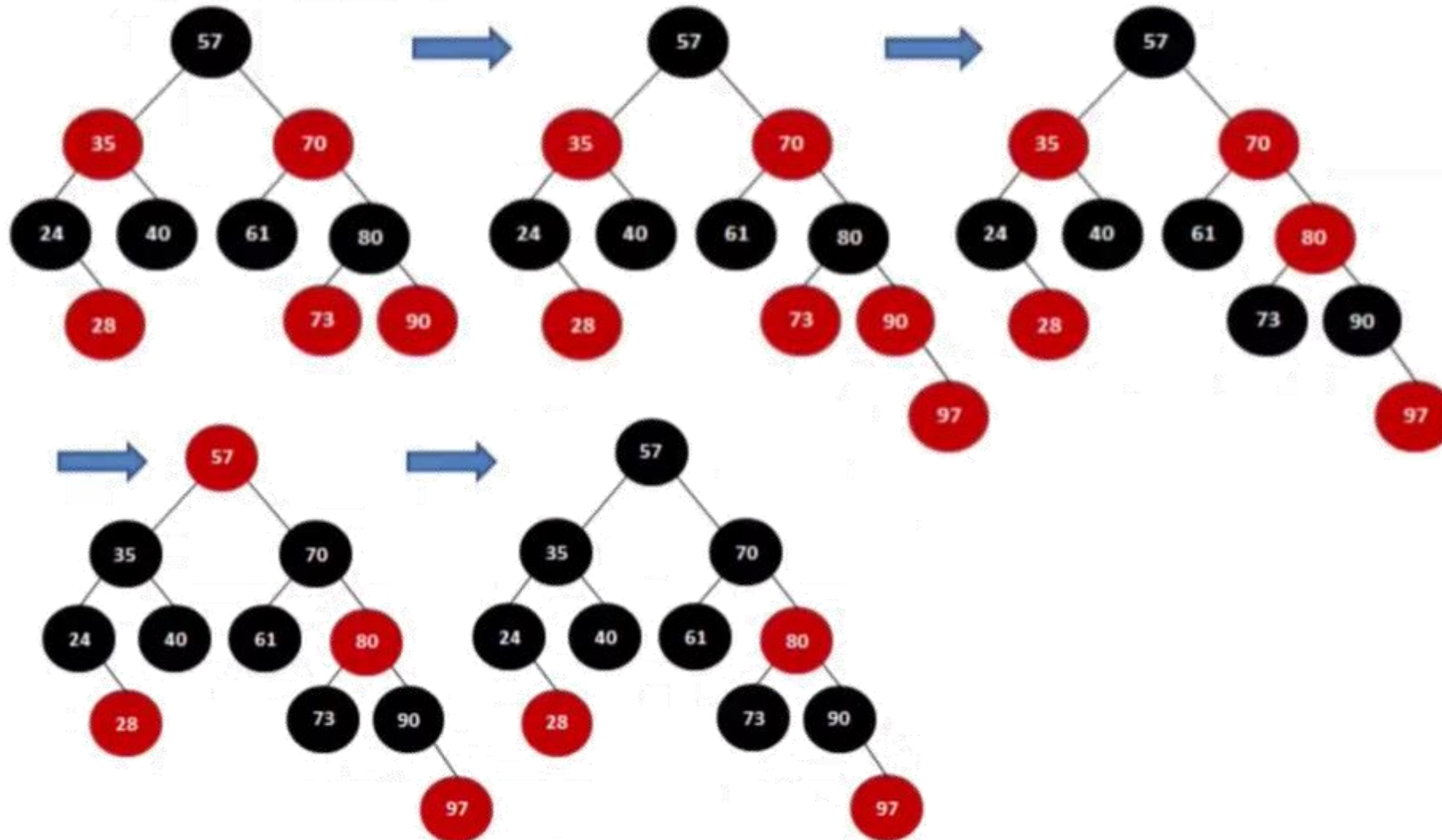
- Pai e tio ficam PRETOS
- Avô fica VERMELHO



# INSERÇÃO em Árvores Vermelho-Preto

## Caso 1: O pai e o tio do novo nó são vermelhos

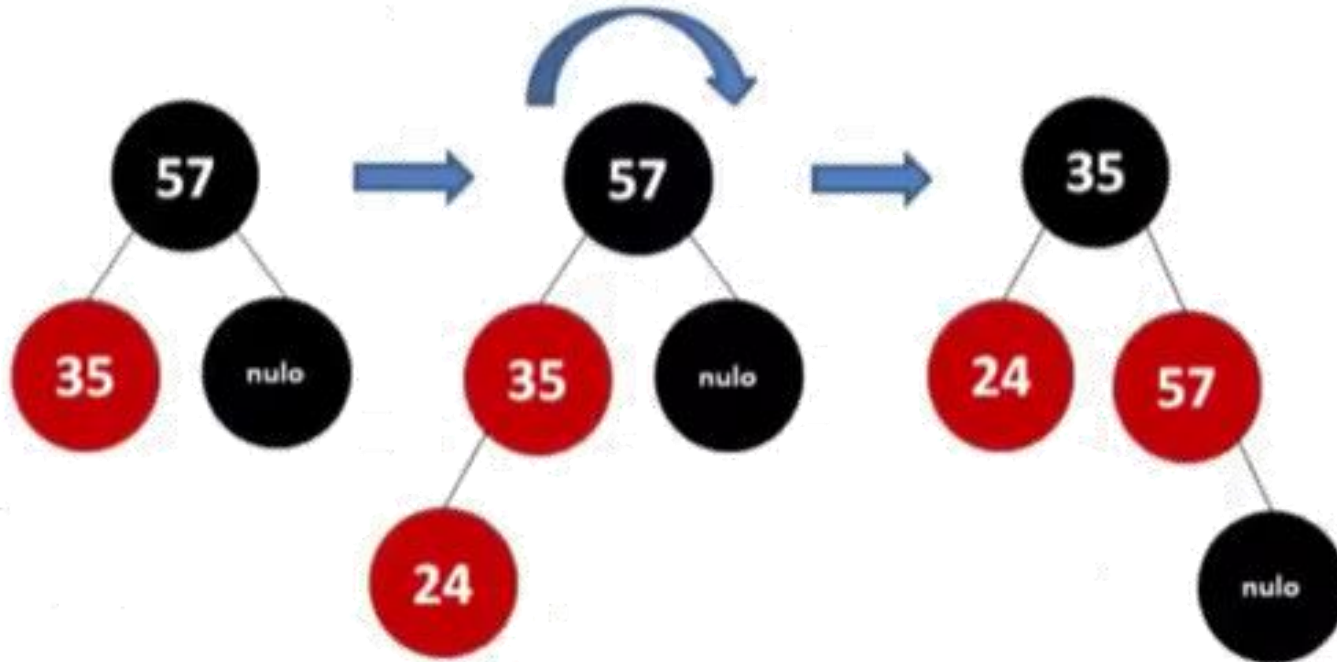
Observe a inserção do número 97.



# INSERÇÃO em Árvores Vermelho-Preto

## Caso 2: O pai é vermelho e o tio é preto

Observe a inserção do número 24.



Rotação simples para a Direita

Pai do novo nó é filho esquerdo e novo nó é filho esquerdo

- Pai fica PRETO;
- Avô fica VERVELHO;
- rotaciona o avô para a DIREITA

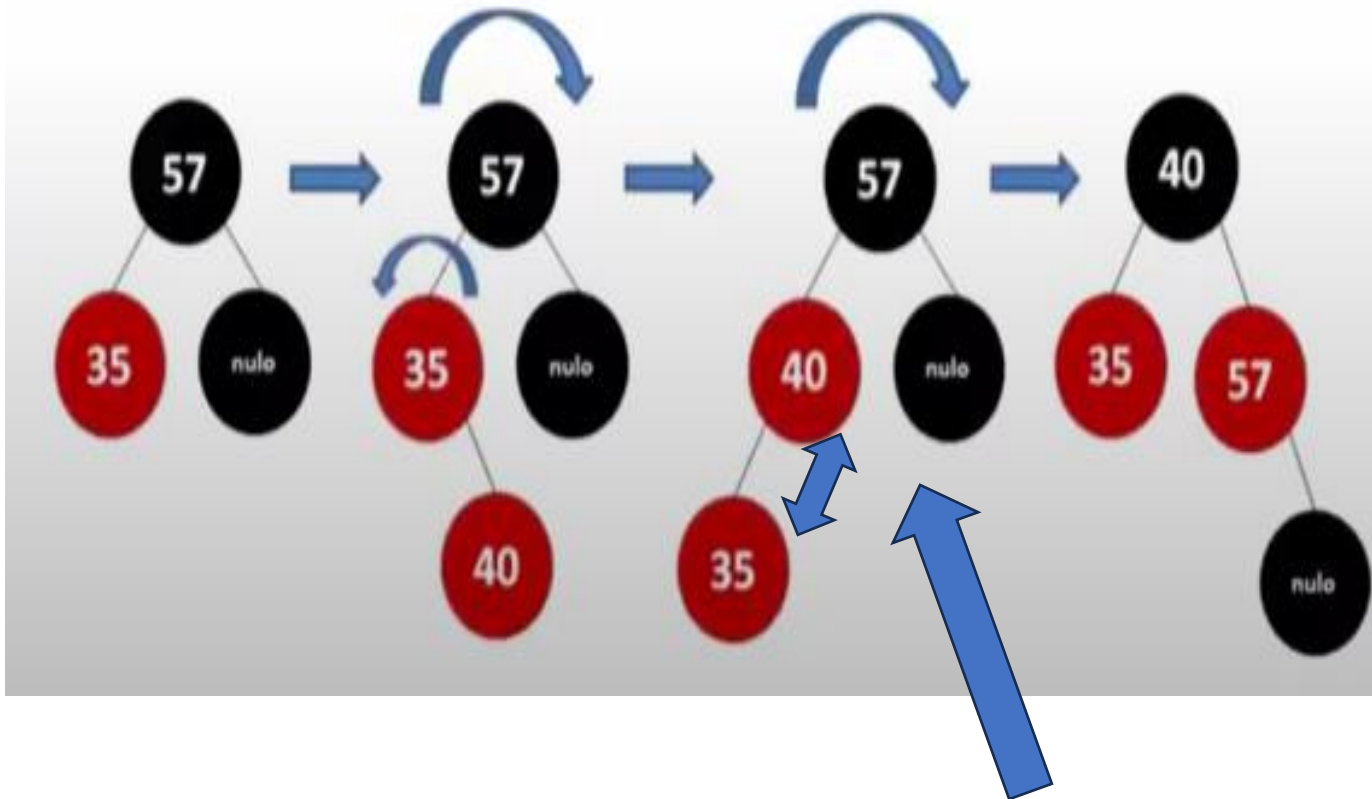
Nesse caso específico não haverá a necessidade de propagação da verificação  
Pois manteve o número de nós negros estável.

# INSERÇÃO em Árvores Vermelho-Preto

## Caso 2: O pai é vermelho e o tio é preto

### Rotação Dupla

Observe a inserção do número 40.



Pai do novo nó é filho esquerdo e novo nó é filho direito

- Rotaciona o pai para a esquerda;
- Novo nó = filho esquerdo do novo nó;
- Pai fica PRETO;
- Avô fica VERMELHO;
- Rotaciona o avô para a direita

Mesmo caso que o anterior