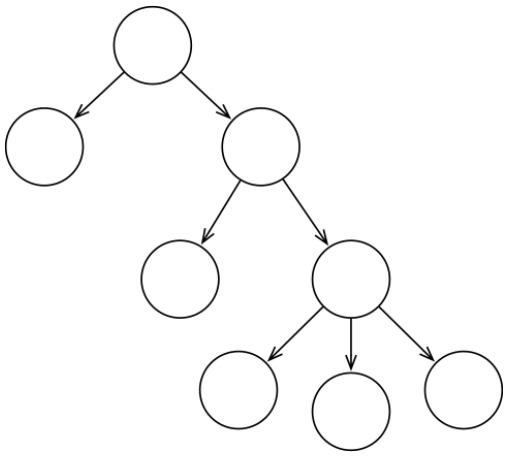
Árvores

Árvores são estruturas de dados hierárquicas. Basicamente, árvores são formadas por um conjunto de elementos, os quais chamamos nodos (ou vértices) conectados de forma específica por um conjunto de arestas. Um dos nodos, que dizemos estar no nível 0, é a raiz da árvore, e está no topo da hierarquia. A raiz está conectada a outros nodos, que estão no nível 1, que por sua vez estão conectados a outros nodos, no nível 2, e assim por diante.

As conexões entre os nodos de uma árvore seguem uma nomenclatura genealógica. Um nodo em um dado nível está conectado a seus filhos (no nível abaixo) e a seu pai (no nível acima). A raiz da árvore, que está no nível 0, possui filhos mas não possui pai.

A figura abaixo ilustra a estrutura de uma árvore.



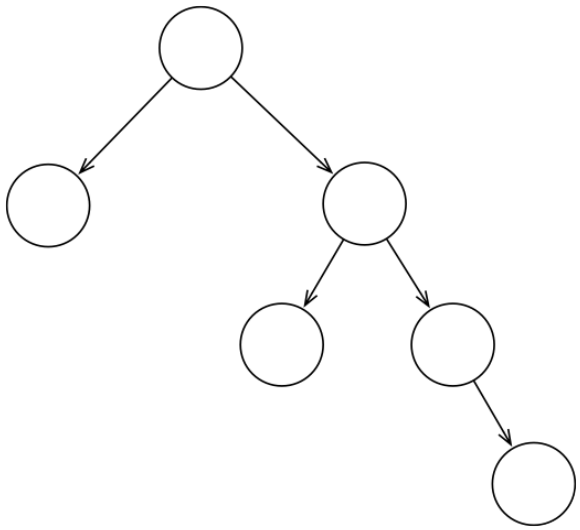
Exemplo de árvore.

Árvores podem ser desenhadas de muitas formas, mas a convenção em Computação é desenhá-las com a raíz no topo, apesar de isso ser um pouco

contra-intuitivo de acordo com nossa noção de árvore do cotidiano.

Árvores Binárias

Árvores binárias são árvores nas quais cada nodo pode ter no máximo dois filhos, conforme mostrado na figura abaixo.



Exemplo de árvore binária.

Uma árvore binária pode ser definida de forma recursiva, de acordo com o raciocínio a seguir. A raiz da árvore possui dois filhos, um à direita e outro à esquerda, que por sua vez são raizes de duas sub-árvores. Cada uma dessas sub-árvores possui uma sub-árvore esquerda e uma sub-árvore direita, seguindo esse mesmo raciocínio.

Representação de Árvores Binárias

Na prática, os nodos de uma árvore binária possuem um valor (chamado de chave) e dois apontadores, um para o filho da esquerda e outro para o filho da direita. Esses apontadores representam as ligações (arestas) de uma árvore. Veja abaixo uma implementação de árvore binária.

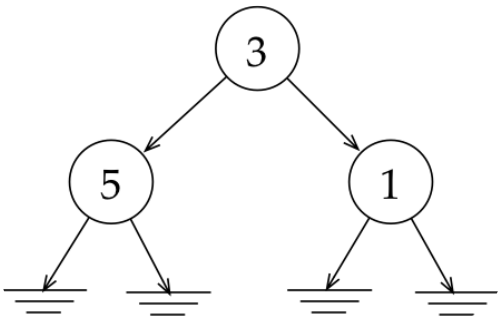
```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

    def __repr__(self):
        return '%s <- %s -> %s' % (self.esquerda and self.esquerda.chave,
                                   self.chave,
                                   self.direita and self.direita.chave)
```

Vejamos como criar nodos de uma árvore usando o código acima.

```
raiz = NodoArvore(3)
raiz.esquerda = NodoArvore(5)
raiz.direita  = NodoArvore(1)
print("Árvore: ", raiz)
```

```
Árvore:  5 <- 3 -> 1
```

A figura abaixo ilustra a árvore binária implementada acima. Note que o nodo raíz (nodo com o valor 3), possui dois filhos, um à esquerda (com o valor 5) e outro à direita (com o valor 1). Note também que os nodos cujos valores são 5 e 1 não possuem filhos (seus apontadores `esquerda` e `direita` são `None`, ou seja, não apontam para nenhum outro nodo).



Representacao de uma árvore binária.

Árvores Binárias de Pesquisa

Árvores binárias de pesquisa (ou Binary Search Tress - BSTs, do Inglês) são árvores cujos nodos são organizados de acordo com algumas propriedades. Mais formalmente, podemos definir árvores binárias de pesquisa como abaixo:

Definição de Árvore Binária de Pesquisa: *Seja x um nodo em uma árvore binária de pesquisa. Se y é um nodo na sub-árvore esquerda de x, então $y.chave \leq x.chave$. Se y é um nodo na sub-árvore direita de x, então $y.chave \geq x.chave$.*

Em outras palavras, árvores binárias de pesquisa são árvores que obedecem às seguintes propriedades:

1. Dado um nodo qualquer da árvore, todos os nodos à **esqueda** dele são **menores** ou iguais a ele.
2. Dado um nodo qualquer da árvore, todos os nodos à **direita** dele são **maiores** ou iguais a ele.

Para simplificar as coisas, não permitiremos elementos repetidos em nossas implementações de BSTs, portanto, nodos à esquerda de um nodo sempre serão menores que ele, e nodos à direita de um nodo serão sempre maiores que ele.

Caminhamentos em Árvore

Caminhamentos em árvore são formas de visitarmos todos os nodos de uma árvore em uma ordem pré-definida. Existem três tipos de caminhamentos básicos: pré-ordem, em ordem, e pós-ordem. Esses três tipos de caminhamentos são bem parecidos, como veremos abaixo.

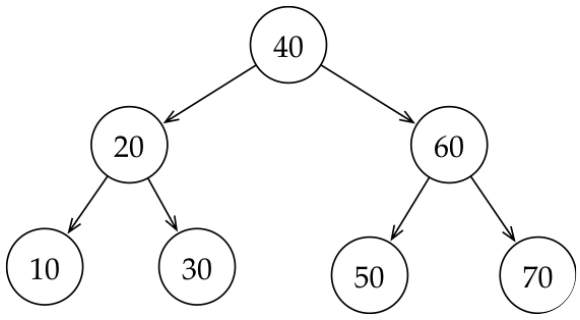
Começaremos nossa explicação com o caminhamento em ordem. Nesse tipo de caminhamento, visitamos recursivamente o nodo da esquerda, visitamos o nodo corrente, e visitamos recursivamente o nodo da direita. Assim, dadas as restrições de uma árvore binária de pesquisa, ao realizarmos o caminhamento em ordem, estaremos de fato visitando os nodos em ordem crescente de chaves. Entretanto, os três tipos de caminhamentos explicados aqui podem ser usados para qualquer tipo de árvore. A única diferença é que ordem em que os nodos serão impressos. Por exemplo, o caminhamento em ordem em uma BST imprime os nodos em ordem crescente, mas em uma árvore binária qualquer a ordem pode não ser essa.

Para ilustrar o caminhamento em ordem, primeiro construiremos uma árvore binária de pesquisa:

```
raiz = NodoArvore(40)

raiz.esquerda = NodoArvore(20)
raiz.direita  = NodoArvore(60)

raiz.direita.esquerda  = NodoArvore(50)
raiz.direita.direita   = NodoArvore(70)
raiz.esquerda.esquerda = NodoArvore(10)
raiz.esquerda.direita  = NodoArvore(30)
```

Note que obedecemos as propriedades de árvores binárias de pesquisa ao inserirmos os nodos na árvore acima.

A figura abaixo mostra uma representação visual da árvore construída acima.



Exemplo de BST.

Agora, vejamos como implementar o caminhamento em ordem.

```
def em_ordem(raiz):
    if not raiz:
        return

    # Visita filho da esquerda.
    em_ordem(raiz.esquerda)

    # Visita nodo corrente.
    print(raiz.chave),

    # Visita filho da direita.
    em_ordem(raiz.direita)
```

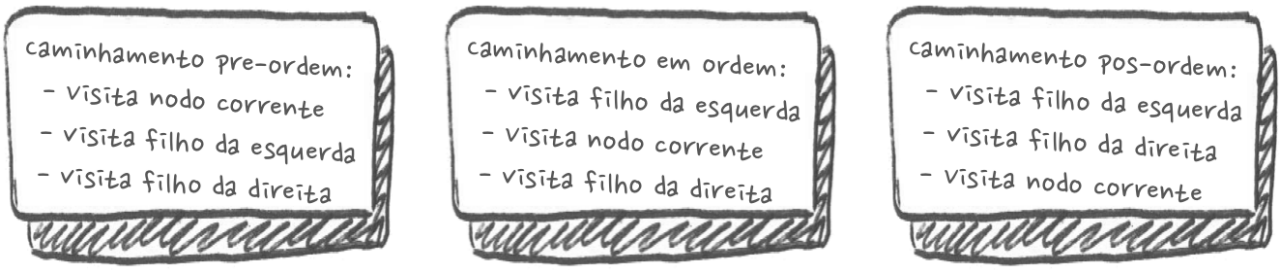
O código acima é uma implementação direta da ideia de caminhamento em ordem. Vejamos o que acontece ao executarmos o código acima na árvore criada anteriormente:

```
em_ordem(raiz)
```

```
10
20
30
40
50
60
70
```

Neste momento, é importante fazermos uma ressalva: ao explicar o caminhamento em ordem dizemos que *visitamos o nodo corrente*, mas na implementação acima simplesmente imprimimos o valor do campo `chave` de cada nodo. Isso ocorre porque o termo *visitar* pode significar tanto algo simples como imprimir o valor da chave, quanto realizar alguma operação mais complexa. A ideia geral do caminhamento em ordem não muda, independente das operações que são realizadas quando visitamos um nodo.

Os outros dois tipos de caminhamento seguem o mesmo raciocínio do caminhamento em ordem. A única diferença é a ordem em que visitamos o nodo corrente. No caminhamento pré-ordem, visitamos o nodo corrente antes de visitarmos recursivamente os nodos da esquerda e direita. No caminhamento pós-ordem visitamos o nodo corrente depois de visitarmos recursivamente os nodos da esquerda e direita. A figura abaixo resume os três tipos de caminhamentos em árvores.



Caminhamentos em árvores.

Inserção em Árvores Binárias de Pesquisa

Na seção anterior, inserimos os nodos na árvore binária de pesquisa um a um, e para garantir que as propriedades de BSTs fossem preservadas, tivemos que indicar explicitamente onde cada nodo deveria ser inserido. Nesta seção, veremos como codificar as regras de BSTs em um procedimento de inserção de modo que, ao chamarmos esse procedimento, as regras de BSTs sejam aplicadas a cada nodo inserido, sem que tenhamos que indicar explicitamente onde os nodos devem ficar.

O maior desafio ao se construir uma função para inserir nodos em uma árvore binária de pesquisa é encontrar o ponto onde cada nodo deve ser inserido. Uma vez encontrado esse ponto, podemos simplesmente ajustar os apontadores `esquerda` ou `direita` para que o nodo seja inserido na árvore.

Para encontrar o ponto de inserção de um nodo em uma árvore binária de pesquisa, precisamos observar as propriedades dessas árvores: dado um nodo qualquer, nodos menores do que ele são inseridos à sua esquerda, e nodos maiores do que ele são inseridos à sua direita. Vejamos como transformar essas ideias em código.

```
def insere(raiz, nodo):
    """Insere um nodo em uma árvore binária de pesquisa."""
    # Nodo deve ser inserido na raiz.
    if raiz is None:
        raiz = nodo

    # Nodo deve ser inserido na subárvore direita.
    elif raiz.chave < nodo.chave:
        if raiz.direita is None:
            raiz.direita = nodo
        else:
            insere(raiz.direita, nodo)

    # Nodo deve ser inserido na subárvore esquerda.
    else:
        if raiz.esquerda is None:
            raiz.esquerda = nodo
        else:
            insere(raiz.esquerda, nodo)
```

Agora criaremos a mesma árvore criada anteriormente, mas desta vez usando a função de inserção que acabamos de desenvolver.

```
# Cria uma árvore binária de pesquisa.
raiz = NodoArvore(40)
for chave in [20, 60, 50, 70, 10, 30]:
    nodo = NodoArvore(chave)
    insere(raiz, nodo)
# Imprime o caminhamento em ordem da árvore.
em_ordem(raiz)
```

```
10
20
30
40
50
60
70
```

Apesar de alguns conceitos de árvores parecerem complexos, temos sido capazes de implementá-los com relativa facilidade. Isso se deve ao fato de que a maioria dos conceitos de árvores possuirem uma natureza inerentemente recursiva. Assim, a implementação deles usando recursividade acaba sendo natural e simples. Sempre que você se deparar com um problema envolvendo árvores, tente pensar em uma solução recursiva antes de mais nada.

Busca em Árvores Binárias de Pesquisa

Assim como a partir das propriedades de árvores binárias de pesquisa fomos capazes de criar um algoritmo para inserir nodos nessas árvores, faremos também para procurar nodos nelas. O algoritmo de busca em árvores binárias de pesquisa pode ser dividido em três casos:

1. A chave procurada está na raiz da árvore. Nesse caso, simplesmente retornamos a raiz da árvore como resultado da busca.
2. A chave procurada é menor que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore esquerda.
3. A chave procurada é maior que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore direita.

A implementação da ideia acima, assim como o tratamento do caso em que o nodo não está presente na árvore, são mostrados no código abaixo.

```
def busca(raiz, chave):
    """Procura por uma chave em uma árvore binária de pesquisa."""
    # Trata o caso em que a chave procurada não está presente.
    if raiz is None:
        return None

    # A chave procurada está na raiz da árvore.
    if raiz.chave == chave:
        return raiz

    # A chave procurada é maior que a da raiz.
```

```
    if raiz.chave < chave:
        return busca(raiz.direita, chave)

    # A chave procurada é menor que a da raiz.
    return busca(raiz.esquerda, chave)
```

O código abaixo testa nossa implementação de busca em árvores binárias de pesquisa.

```
# Cria uma árvore binária de pesquisa.
raiz = NodoArvore(40)
for chave in [20, 60, 50, 70, 10, 30]:
    nodo = NodoArvore(chave)
    insere(raiz, nodo)

# Procura por valores na árvore.
for chave in [-50, 10, 30, 70, 100]:
    resultado = busca(raiz, chave)
    if resultado:
        print("Busca pela chave {}: Sucesso!".format(chave))
    else:
        print("Busca pela chave {}: Falha!".format(chave))
```

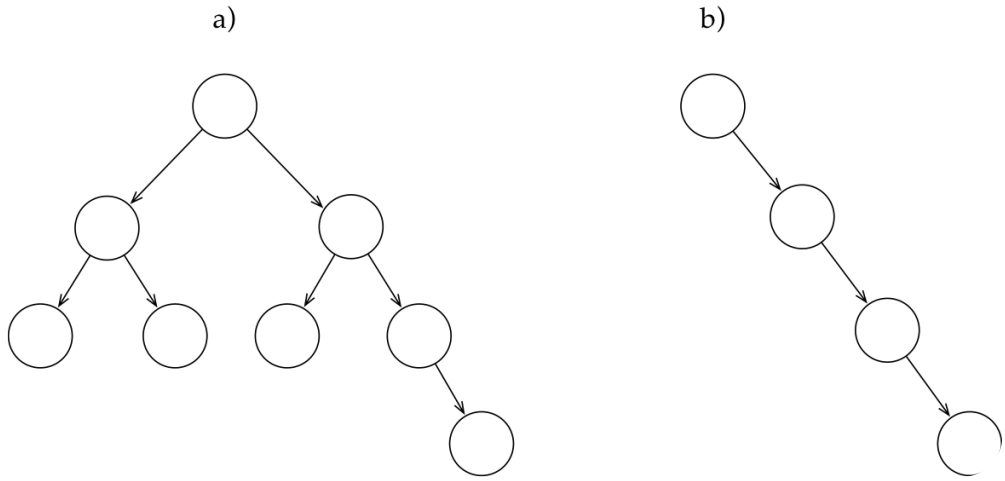
```
Busca pela chave -50: Falha!
Busca pela chave 10: Sucesso!
Busca pela chave 30: Sucesso!
Busca pela chave 70: Sucesso!
Busca pela chave 100: Falha!
```

Complexidades Assintóticas

Antes de discutirmos complexidade assintótica de procedimentos que operam sobre árvores binárias, precisamos definir o que é uma árvore balanceada. A necessidade dessa definição ficará clara abaixo.

Definição: *Uma árvore binária é balanceada se a diferença da profundidade de duas folhas quaisquer é no máximo 1. A profundidade de um nodo é o número de níveis da raiz até aquele nodo.*

Na figura abaixo, a árvore a) é balanceada, e a árvore b) não é balanceada.



Exemplo de árvore balanceada e não balanceada.

Se uma árvore é balanceada, tanto no caso da inserção quando no caso da busca, a cada chamada recursiva do algoritmo, descartamos metade da árvore original. Portanto, a complexidade assintótica desses dois procedimentos é logarítmica no tamanho (número de nodos) da árvore. Na verdade, muitos procedimentos que operam sobre árvores binárias de pesquisa funcionam com base nessa mesma ideia de eliminar metade da árvore a cada etapa do procedimento. Isso ocorre por causa da natureza recursiva das árvores binárias de pesquisa e pela forma como os nodos são inseridos nelas.

Entretanto, se a árvore não for balanceada, não descartaremos metade da árvore original a cada chamada recursiva. Em casos como o da árvore não balanceada mostrada acima, a complexidade dos procedimentos de inserção e busca será linear no tamanho da árvore, pois, na prática, a árvore mostrada funciona como se fosse uma lista encadeada.

No caso de árvores binárias, é importante sempre fazer a distinção da complexidade dos procedimentos caso a árvore seja balanceada e caso ela não seja. Além disso, é importante estar atento a procedimentos que parecem logarítmicos, mas que na verdade visitam todos os nodos da árvore. Para resumir, tenha em mente as seguintes situações:

- Se um procedimento visita todos os nodos de uma árvore, sua complexidade assintótica é linear no número de nodos da árvore. Como exemplo desse tipo de procedimento citamos os caminhamentos pré-ordem, em ordem, e pós-ordem.
- Se um procedimento descarta metade da árvore a cada iteração ou chamada e a árvore é balanceada, sua complexidade é logarítmica no tamanho da árvore. Como exemplo desse tipo de procedimento citamos a inserção e a busca em árvores binárias de pesquisa, quando executados em árvores balanceadas.
- Se um procedimento descarta metade da árvore a cada iteração ou chamada mas a árvore não é balanceada, sua complexidade é linear no tamanho da árvore. Como exemplo desse tipo de procedimento citamos a inserção e a busca em árvores binárias de pesquisa, quando executados em árvores não balanceadas.

Existe uma forma alternativa de se analisar a complexidade de operações em árvores e que dependem de a árvore ser ou não balanceada. Essa alternativa consiste de descrevermos a complexidade assintótica das operações não em termos do tamanho (número de nodos) da árvore, mas em termos da altura (número de níveis) da árvore. A altura de uma árvore balanceada, como a mostrada na parte a) da figura anterior, será $O(\log n)$, mas se a árvore não for balanceada, como a mostrada na parte b) da figura anterior, a altura será $O(n)$. A escolha de se analisar a complexidade em termos do tamanho da árvore ou de sua altura é uma escolha sua. Mas seja qual for a sua escolha, tenha em mente que, dependendo da forma como os nodos estiverem organizados na árvore, as operações podem ser logarítmicas ou lineares.

Exercícios

Exercício 1: Encontre o menor elemento em uma BST.

O menor elemento de uma BST é o nodo mais à esquerda na árvore.

```
def minimo(raiz):
    nodo = raiz
    while nodo.esquerda is not None:
        nodo = nodo.esquerda
    return nodo.chave
```

Exercício 2: Encontre o maior elemento em uma BST.

O maior elemento de uma BST é o nodo mais à direita na árvore.

```
def maximo(raiz):
    nodo = raiz
    while nodo.direita is not None:
        nodo = nodo.direita
    return nodo.chave
```

Exercício 3: Verifique se duas árvores binárias são idênticas.

As três situações a serem tratadas são resumidas no código abaixo.

```
def identicas(a, b):
    # 1. As duas árvores são vazias.
    if a is None and b is None:
        return True

    # 2. Nenhuma das árvores é vazia. Precisamos compará-las.
    if a is not None and b is not None:
        return ((a.chave == b.chave) and
                identicas(a.esquerda, b.esquerda) and
                identicas(a.direita, b.direita))

    # 3. Uma árvore é vazia mas a outra não.
    return False
```

Exercício 4: Calcule a altura de uma BST.

A altura de um nodo em uma árvore é o número de níveis desde o nodo em questão até a folha mais baixa da árvore. Quando falamos da altura de uma árvore, estamos nos refedindo à altura do nodo raíz.

Por definição, a altura de uma árvore é a altura da raiz (que é 1), mais a altura da maior subárvore. O código abaixo implementa essa ideia.

```
def altura(raiz):
    if raiz is None:
        return 0
    return max(altura(raiz.esquerda), altura(raiz.direita)) + 1
```

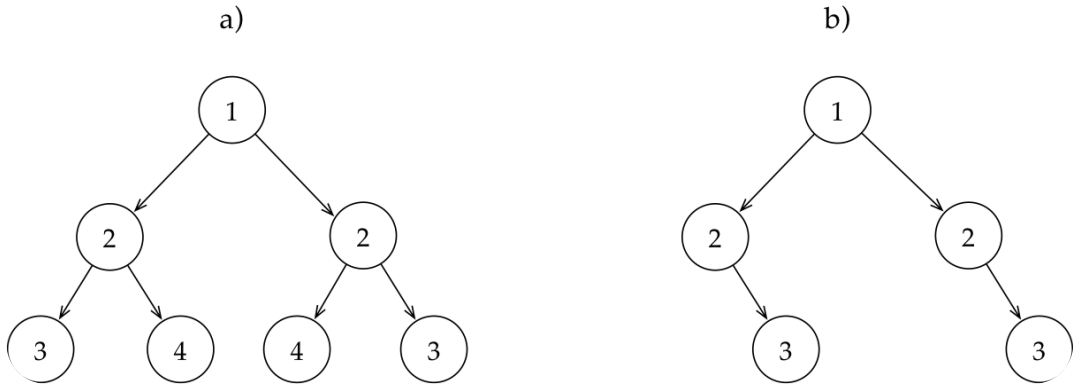
Exercício 5: Verifique se uma árvore binária é balanceada.

```
def balanceada(raiz):
    # Uma árvore binária vazia é balanceada.
    if raiz is None:
        return True

    altura_esq = altura(raiz.esquerda)
    altura_dir = altura(raiz.direita)
    # Alturas diferem em mais de uma unidade.
    if abs(altura_esq - altura_dir) > 1:
        return False

    return balanceada(raiz.esquerda) and balanceada(raiz.direita)
```

Exercício 6: Determine se uma árvore é simétrica.

Uma árvore (note que aqui não estamos falando de BST) é simétrica se a subárvore esquerda for um espelho da subárvore direita. A figura abaixo

mostra uma árvore simétrica, na parte a), e uma árvore não simétrica, na parte b).



Exemplo de árvore simétrica e não simétrica.

O código abaixo implementa a verificação se uma árvore é simétrica. A ideia dessa implementação pode ser resumida em três passos:

1. Dado um nodo, o filho à esquerda desse nodo precisa ser igual ao filho da direita.
2. O filho da esquerda da subárvore esquerda precisa ser igual ao filho da direita da subárvore direita.
3. O filho da direita da subárvore esquerda precisa ser igual ao filho da esquerda da subárvore direita.

```
def checa_simetrica(raiz):
    def simetricas(subarvore_esq, subarvore_dir):
        if not subarvore_esq and not subarvore_dir:
            return True
        elif subarvore_esq and subarvore_dir:
            return (subarvore_esq.chave == subarvore_dir.chave and
                    simetricas(subarvore_esq.esquerda, subarvore_dir.direita) and
                    simetricas(subarvore_esq.direita, subarvore_dir.esquerda))
        # Uma sub-árvore é vazia e a outra não.
        return False

    return not raiz or simetricas(raiz.esquerda, raiz.direita)
```

← Anterior Início Próximo →