UNISINOS

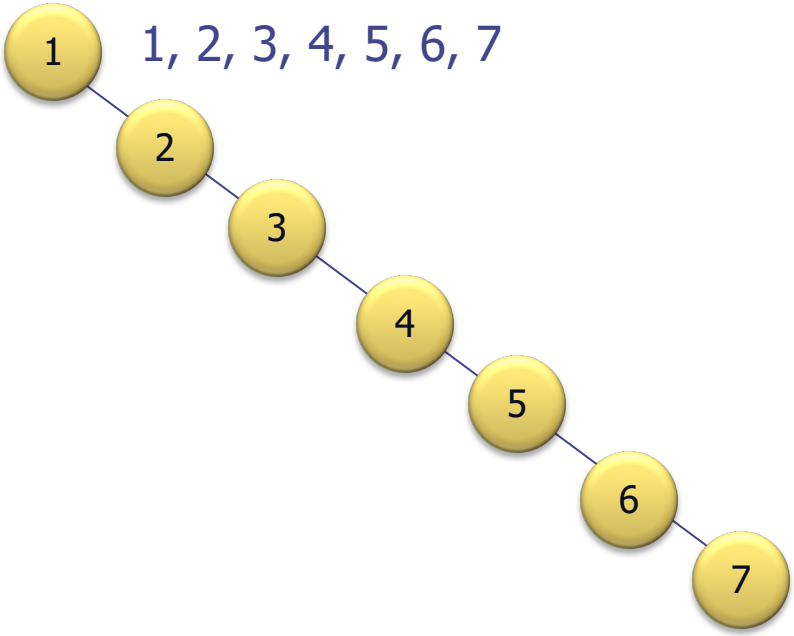
Estruturas Avançadas de Dados I
(Árvore AVL)

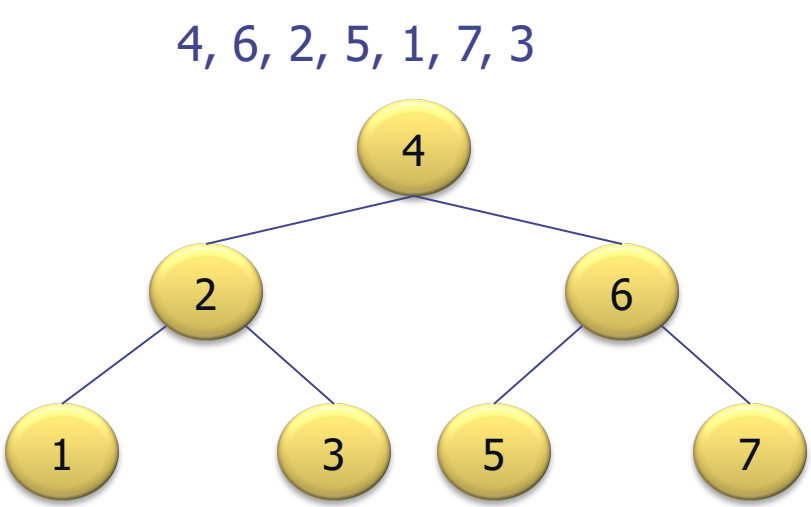
Prof. Gilberto Irajá Müller

Última atualização 12/03/2018

Introdução

- As Árvores Binárias de Pesquisa (ABP) estudadas têm uma **séria desvantagem** que pode afetar o tempo necessário para recuperar um item armazenado;
- A desvantagem é que o desempenho da ABP **depende da ordem em que os elementos são inseridos**.





Introdução

- Idealmente, deseja-se que a árvore esteja **balanceada** para qualquer nó **p** da árvore;
- Como saber se a árvore está **balanceada**?
- Para cada nó **p** da árvore a altura da sua **subárvore à esquerda** é **aproximadamente igual** à altura da sua **subárvore à direita**.

Definição

- O nome AVL vem de seus criadores Adelson Velsky e Landis (1962);

- Uma ABP **T** é denominada **AVL** se:
  - Para todos nós de **T**, as alturas de suas duas subárvores diferem **no máximo em uma unidade;**

- Uma ABP é **completamente balanceada** quando a distância média dos nós até a raiz for mínima.

Definição

- **Balanceamento Estático**

  Este balanceamento consiste em, depois de um certo tempo de uso da árvore, destruir sua estrutura, guardando suas informações em uma lista ordenada e reconstruí-la de forma balanceada.

- **Balanceamento Dinâmico**

  Tem por objetivo reajustar os nós de uma árvore sempre que uma inserção ou remoção provocar desbalanceamento.

Definição

- Como saber se a árvore está **desbalanceada**?
  - Verificando se existe algum nó "desregulado".
- Como saber se um nodo está **desregulado**?
  - Subtraindo-se as alturas das suas subárvores.
- Por questões de **eficiência**, a altura é armazenada nos nós correspondentes, sendo atualizadas durante as operações.

AVL – Fator de Balanceamento

- Possíveis valores de diferença para cada nó em uma árvore balanceada: **-1**, **0**, **1**.

- Fator de Balanceamento (FB) de cada nó da árvore
  - **FB(p)** = h(sae(p)) – h(sad(p)) :

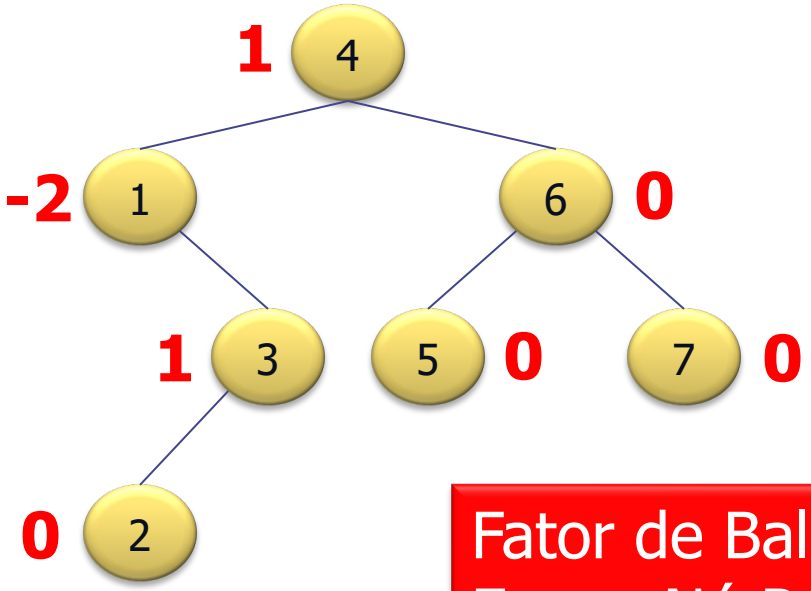
    h = altura
    p = nó
    sae = subárvore à esquerda
    sad = subárvore à direita

- Em uma árvore binária balanceada todos os FBs de todos os nós estão no intervalo $-1 \leq FB \leq 1$.

AVL - Exemplo

- Exemplo de uma ABP não AVL

  Inserção: **4, 1, 3, 6, 5, 2 e 7**



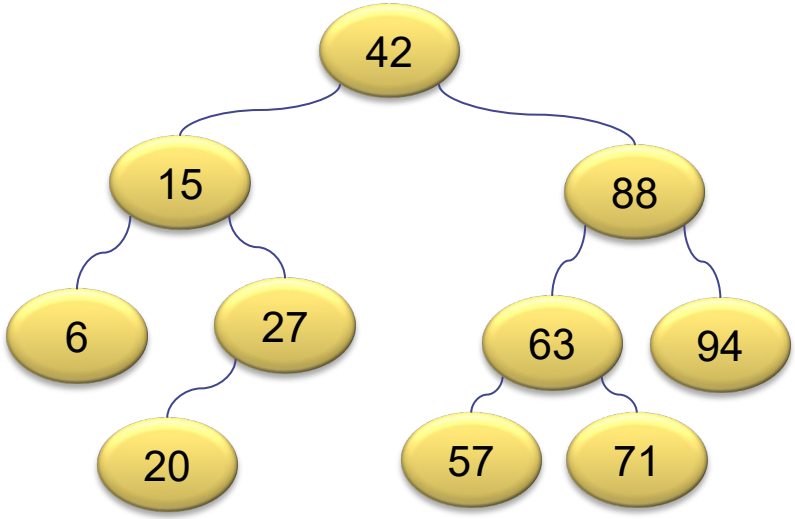
Fator de Balanceamento
Ex.: Nó Raiz
FB("4") = 3 – 2
FB("4") = **1**

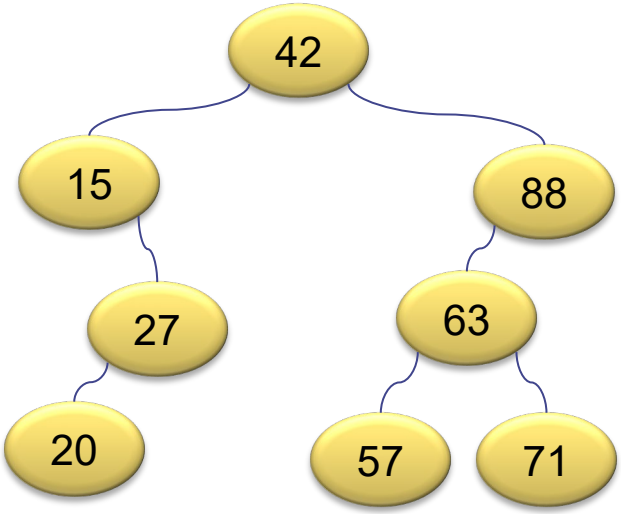
AVL – Vantagem

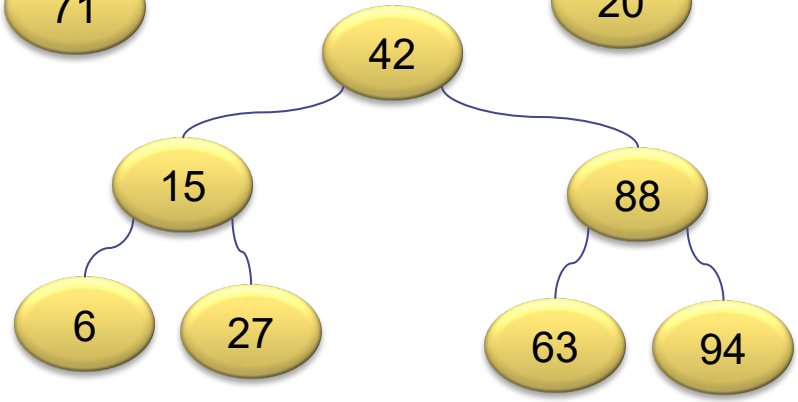
- A vantagem de uma árvore balanceada com relação a uma degenerada está em sua eficiência;
- Ex.: numa árvore binária degenerada de 10.000 nós são necessárias, em média, 5.000 comparações (semelhança com arrays ordenados e listas encadeadas);
- Numa árvore balanceada com o mesmo número de nós essa média reduz-se a 14 comparações.

AVL – Exercício

- **Exercício 1:** Identifique quais árvores BSTs abaixo são AVL.







AVL – Como garantir o balanceamento?

- A inserção ou remoção de um nó em uma árvore AVL **pode** (ou **não)** provocar seu desbalanceamento;

- Se a árvore AVL ficar desbalanceada, a restauração do seu balanceamento é realizada através de **ROTAÇÕES**.

Rotação Simples:
- Rotação à Esquerda
- Rotação à Direita

Rotação Dupla:
- Rotação à Esquerda
- Rotação à Direita

AVL – Classe Nodo

```
protected class Node {
    private K key;
    private V value;
    private int height;
    private Node left, right;

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public Node next(K other) {
        return other.compareTo(key) < 0 ? left : right;
    }

    public boolean isLeaf() {
        return left == null && right == null;
    }

    @Override
    public String toString() {
        return "" + key;
    }
}
```

AVL – Classe

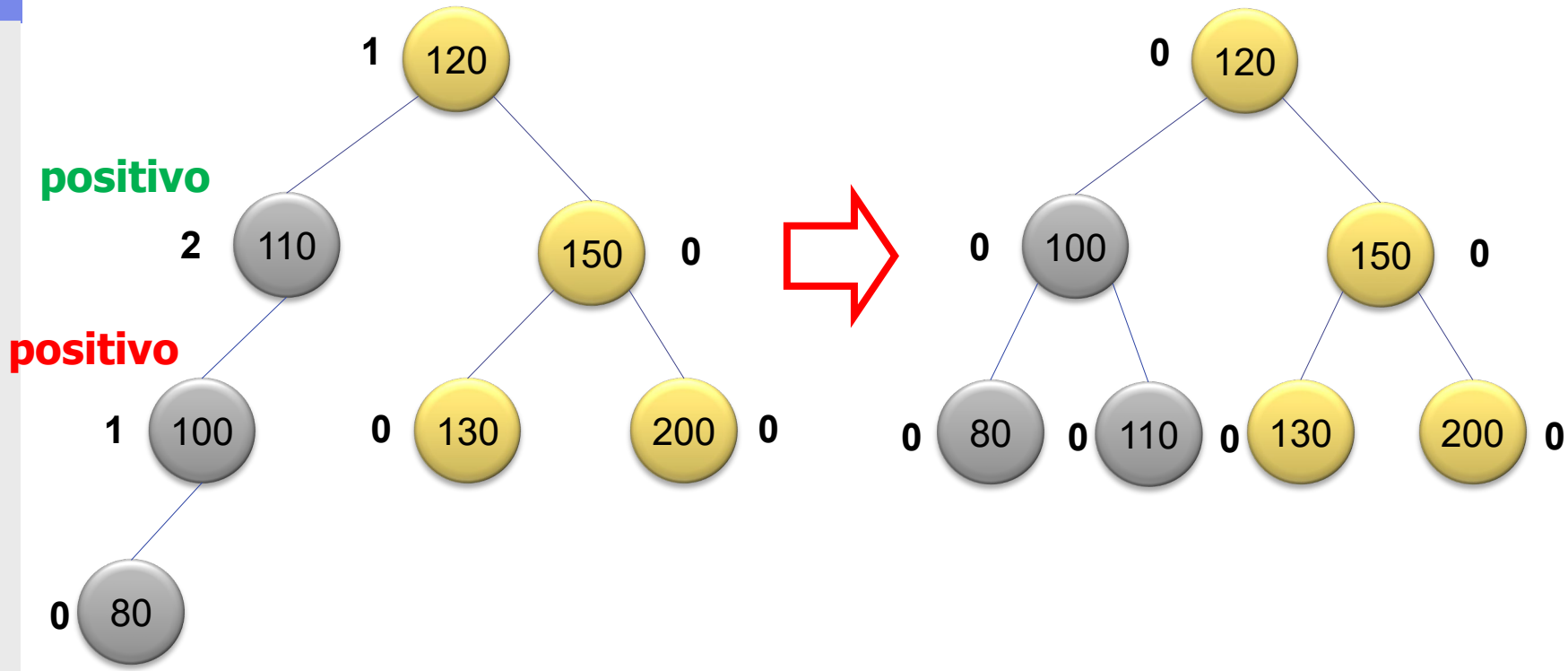
```
public class AvlBinarySearchTree<K extends Comparable<K>, V>
implements BinarySearchTreeADT<K, V> {
   protected Node root;

   protected class Node { ... } // Classe Anterior!

   @Override
   public void clear() {
      root = null;
   }

   @Override
   public boolean isEmpty() {
      return root == null;
   }

// Continua...
```

AVL – Interface

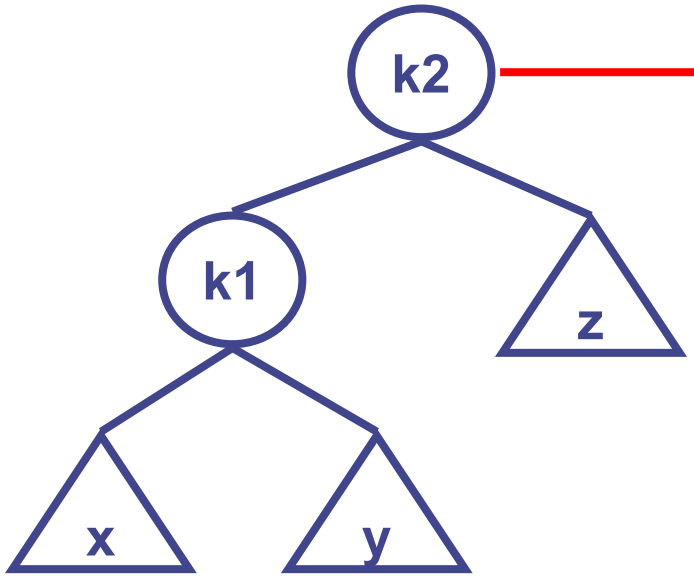
```
public interface BinarySearchTreeADT<K, V> {
    public void clear();
    public boolean isEmpty();
    public V search(K key);
    public void insert(K key, V value);
    public boolean delete(K key);
    public void preOrder();
    public void inOrder();
    public void postOrder();
    public void levelOrder();
}
```

AVL – Rotação Simples à Direita

- Toda vez que uma subárvore fica com um fator:
  - positivo e sua subárvore da esquerda também tem um fator positivo



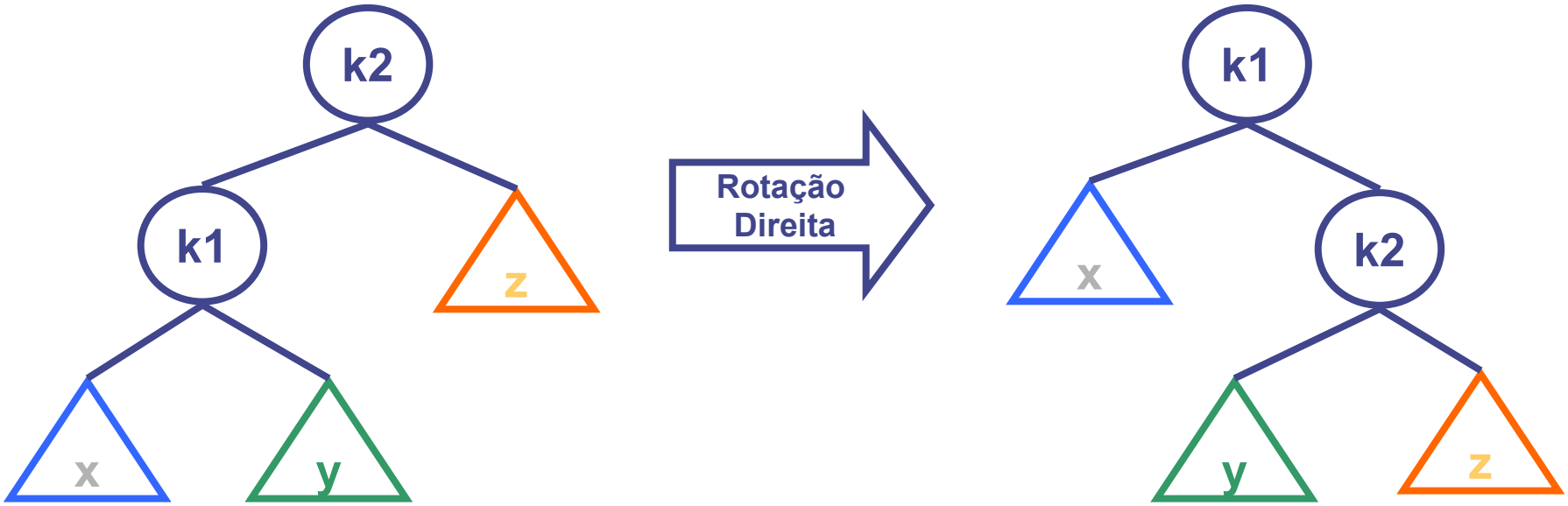
AVL – Rotação Simples à Direita



Nó k2 é a raiz de transformação

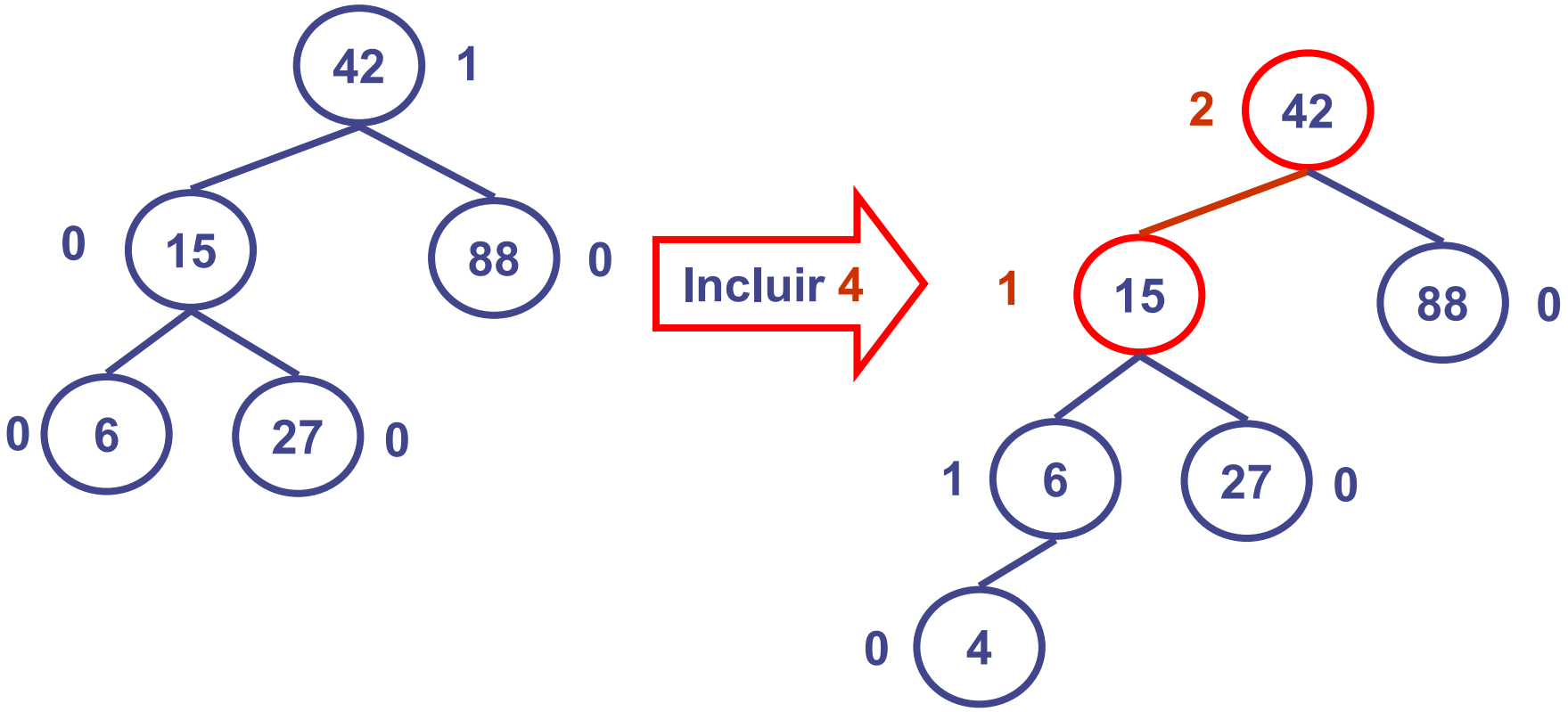
X, Y e Z são subárvores
(ou vazias)

AVL – Rotação Simples à Direita

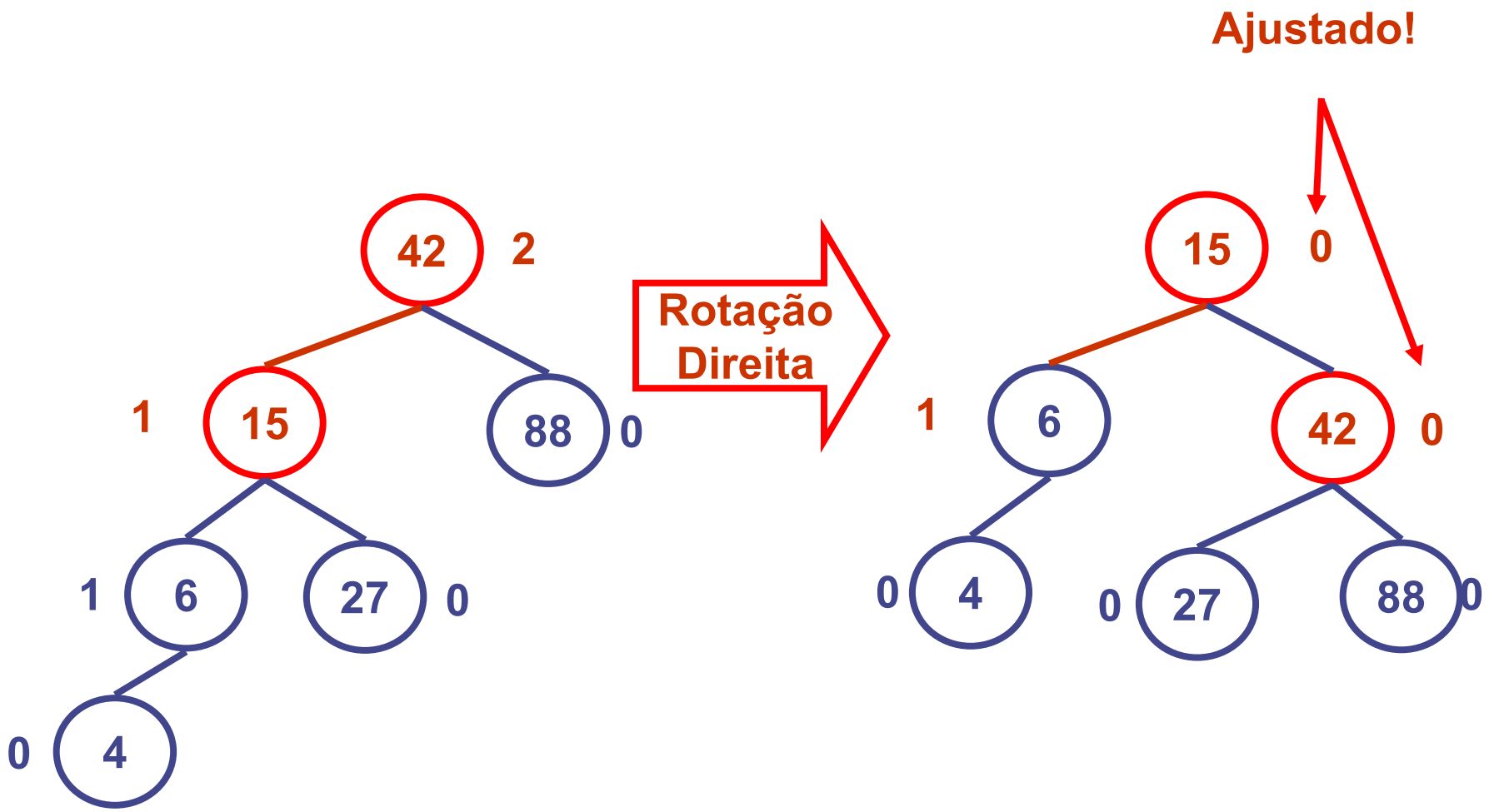



```
private Node doRightRotation(Node k2) {
    Node k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = 1 + Math.max(height(k2.left), height(k2.right));
    k1.height = 1 + Math.max(height(k1.left), height(k1.right));
    return k1;
}
```

AVL – Rotação Simples à Direita

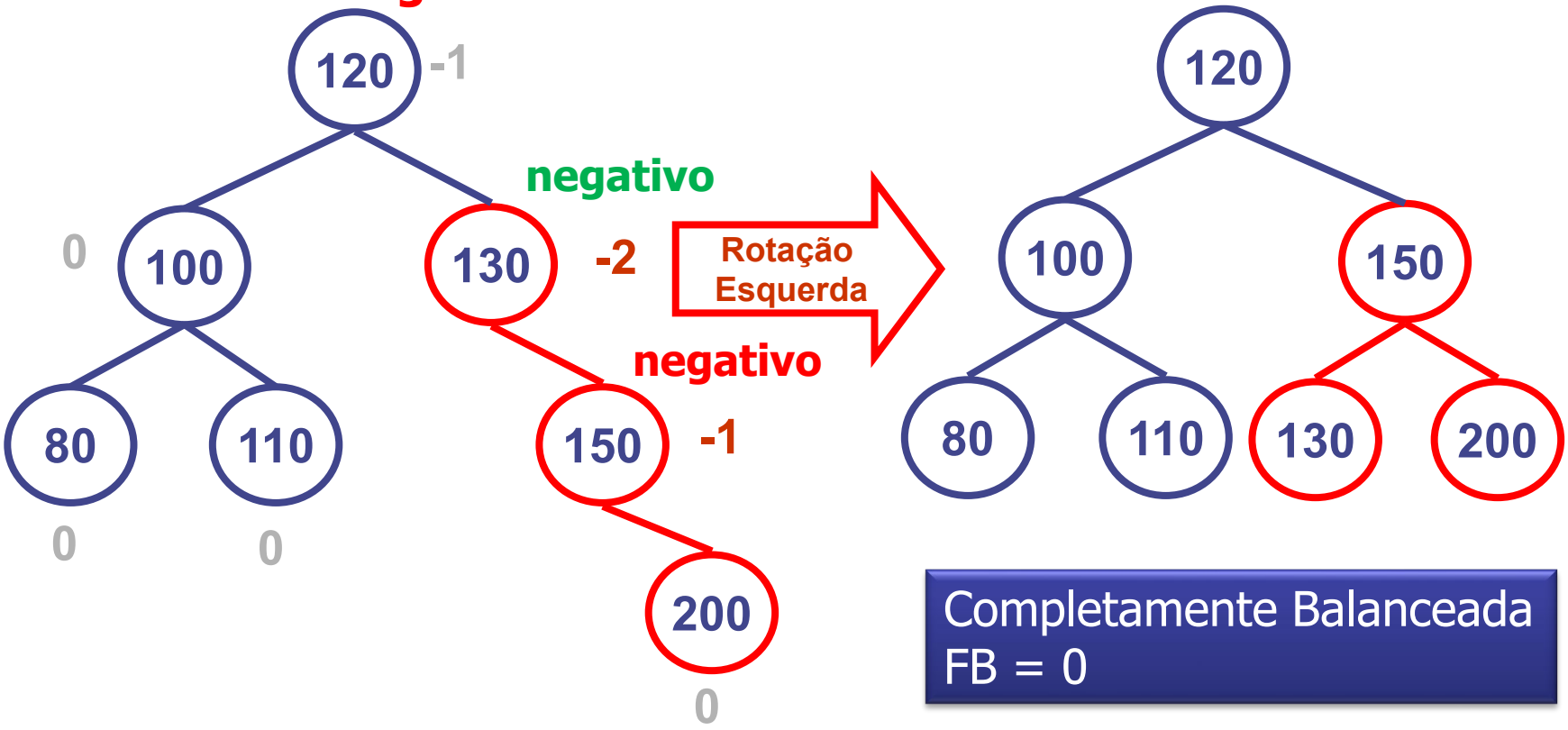


AVL – Rotação Simples à Direita



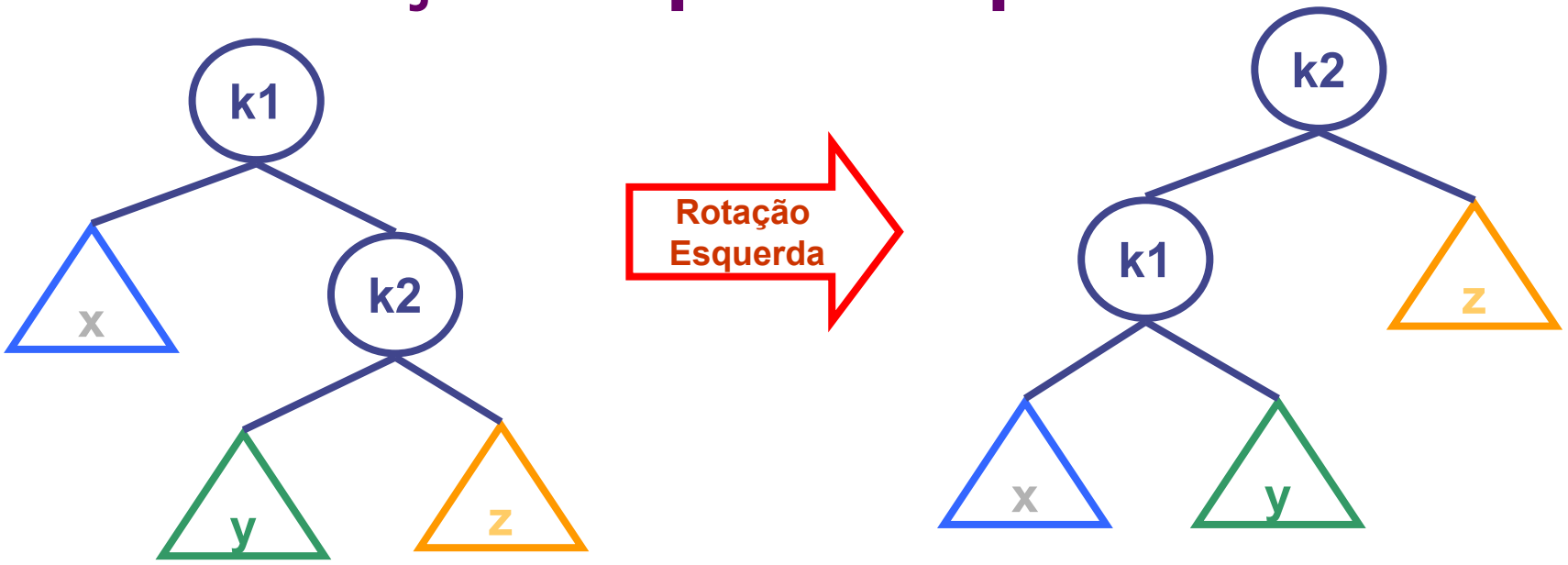
AVL – Rotação Simples à Esquerda

- Toda vez que uma subárvore fica com um fator:
  - negativo e sua subárvore da direita também tem um fator negativo



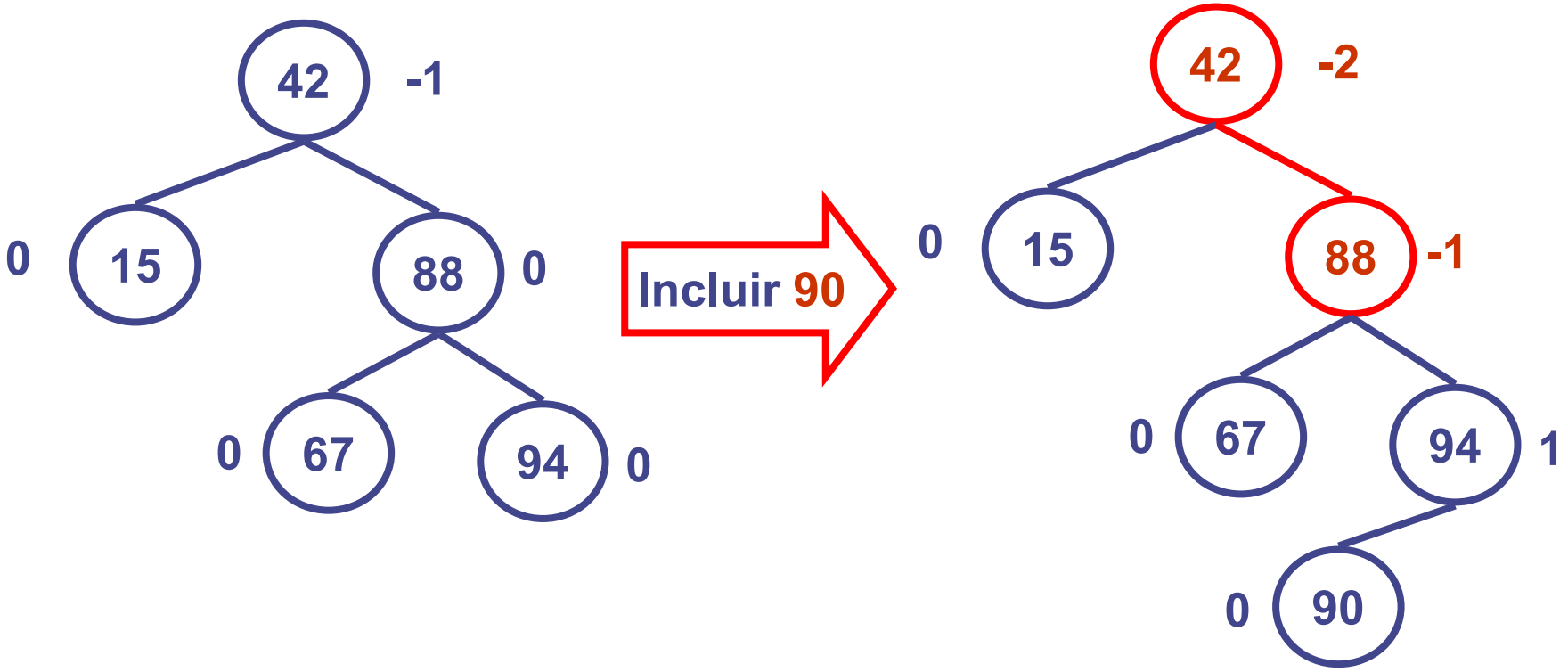
Completamente Balanceada
FB = 0

AVL – Rotação Simples à Esquerda

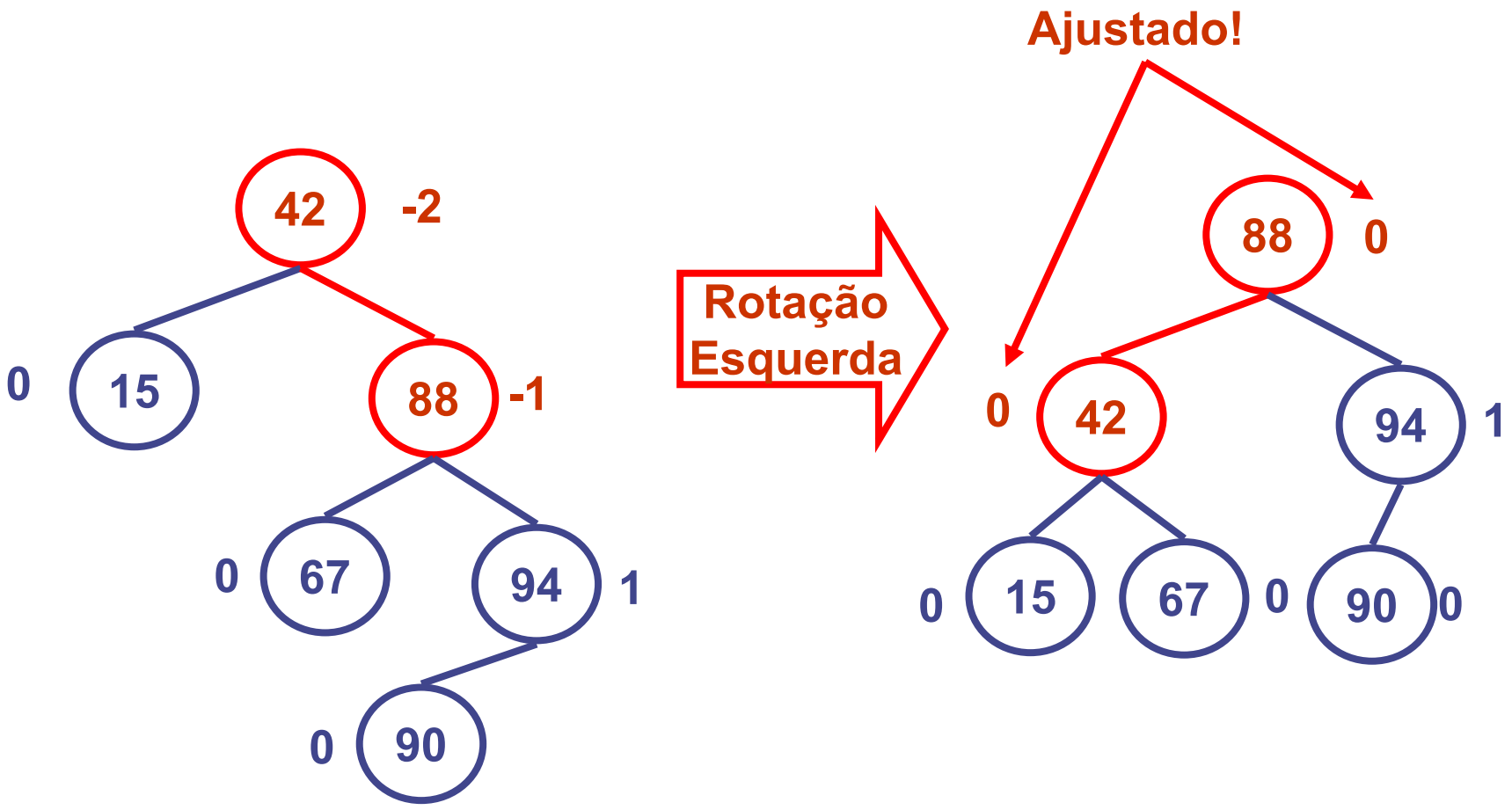



```
private Node doLeftRotation(Node k1) {
    Node k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = 1 + Math.max(height(k1.left), height(k1.right));
    k2.height = 1 + Math.max(height(k2.left), height(k2.right));
    return k2;
}
```

AVL – Rotação Simples à Esquerda

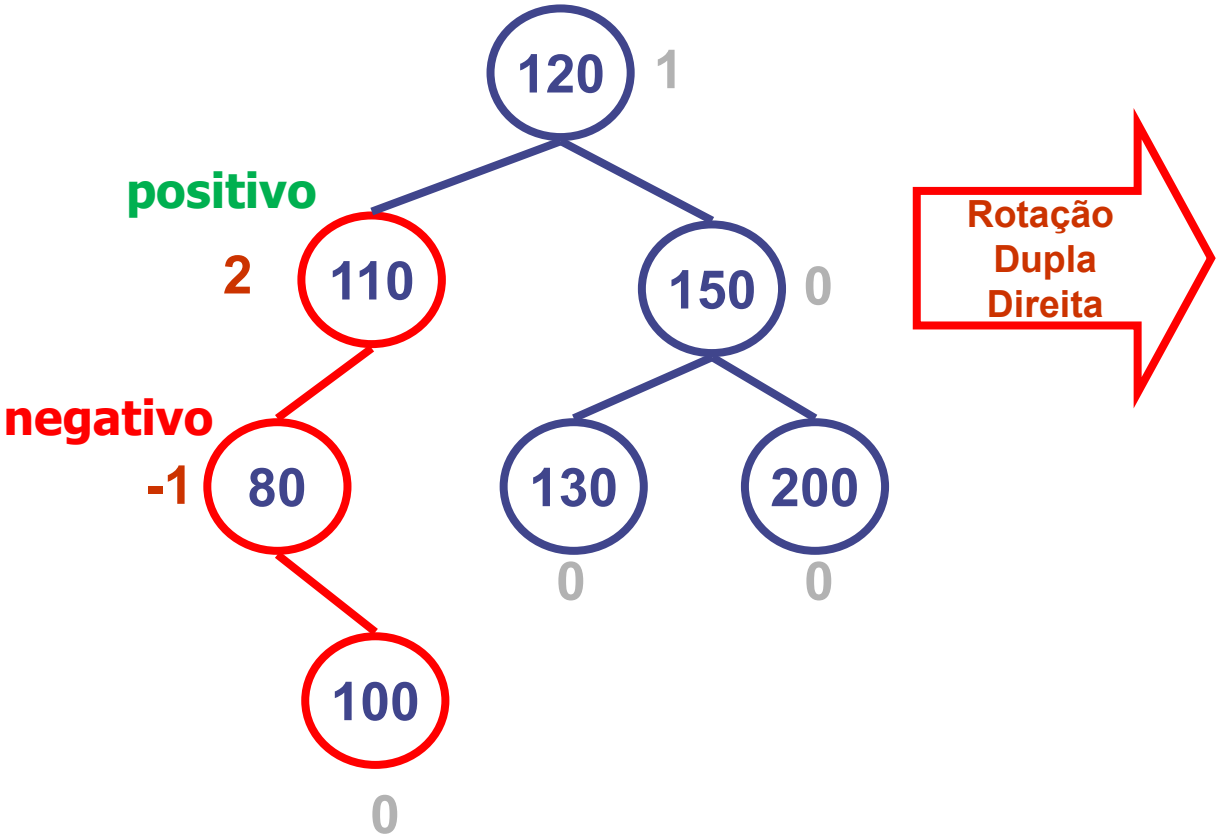


AVL – Rotação Simples à Esquerda

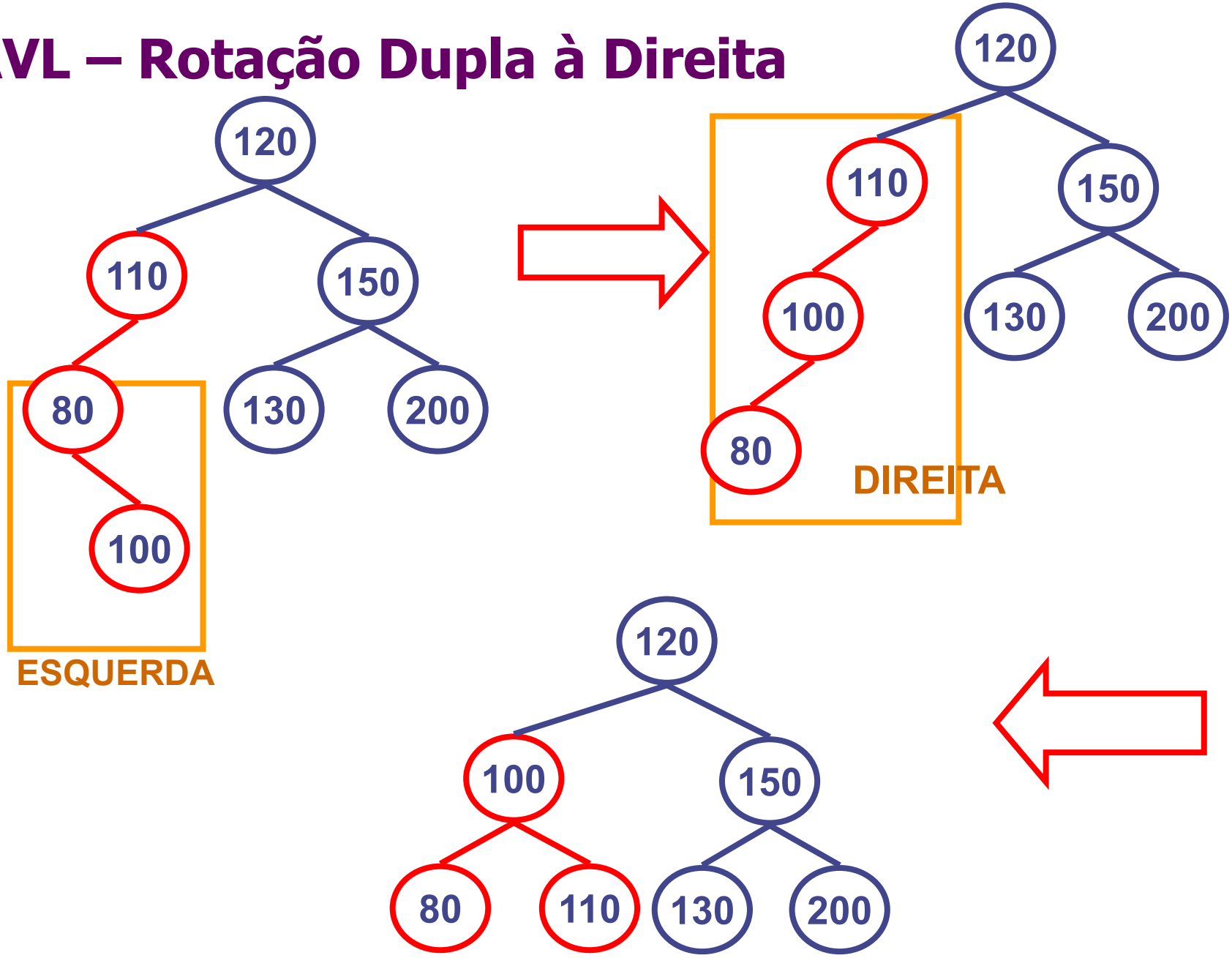


AVL – Rotação Dupla à Direita

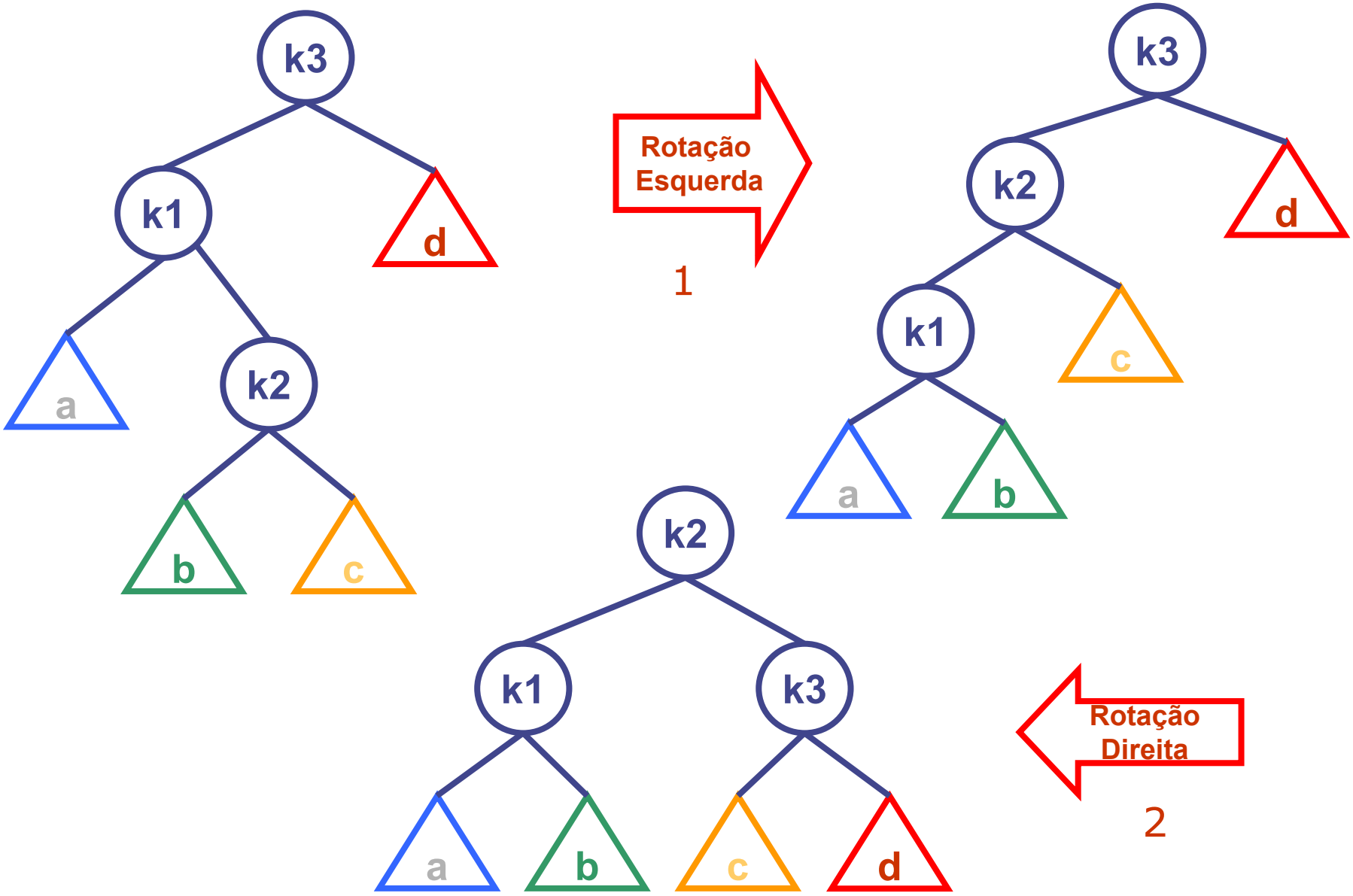
- Toda vez que uma subárvore fica com um fator:
  - positivo e sua subárvore da esquerda tem um fator negativo



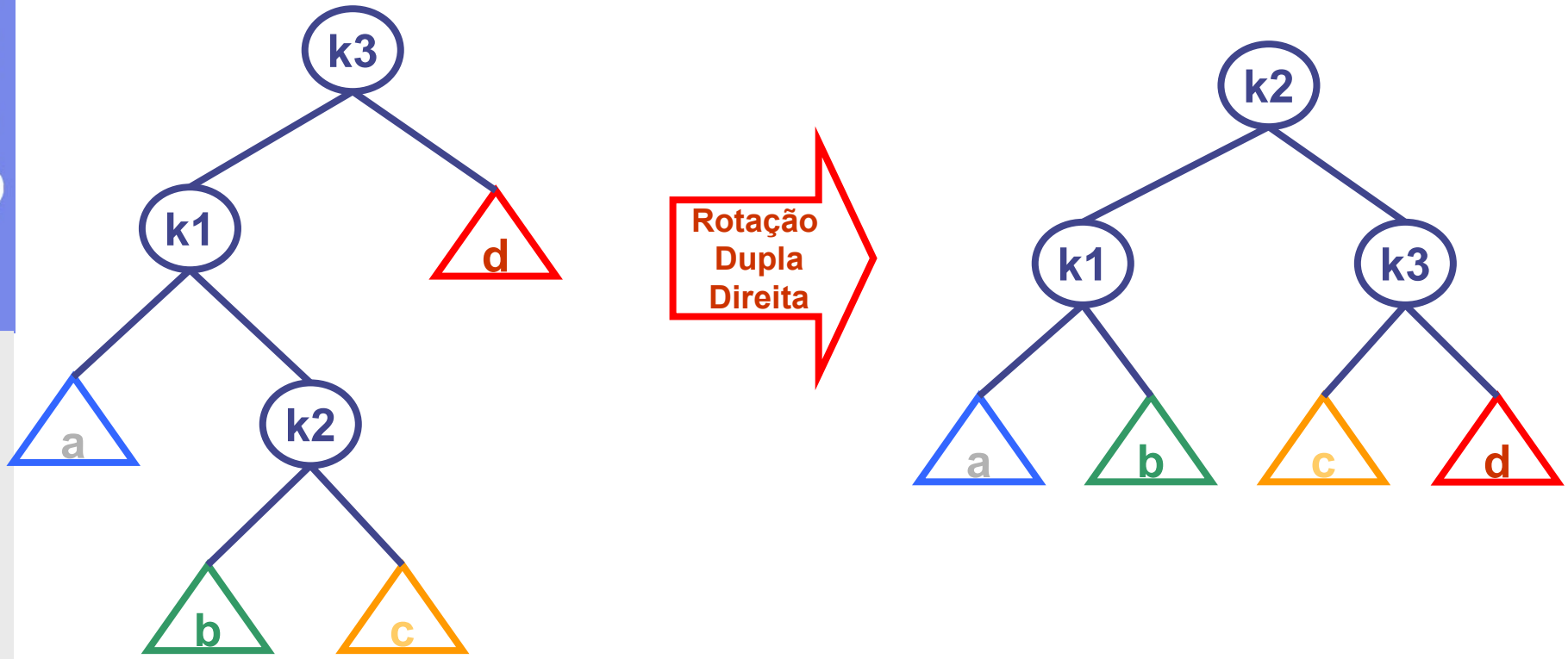
AVL – Rotação Dupla à Direita



AVL – Rotação Dupla à Direita

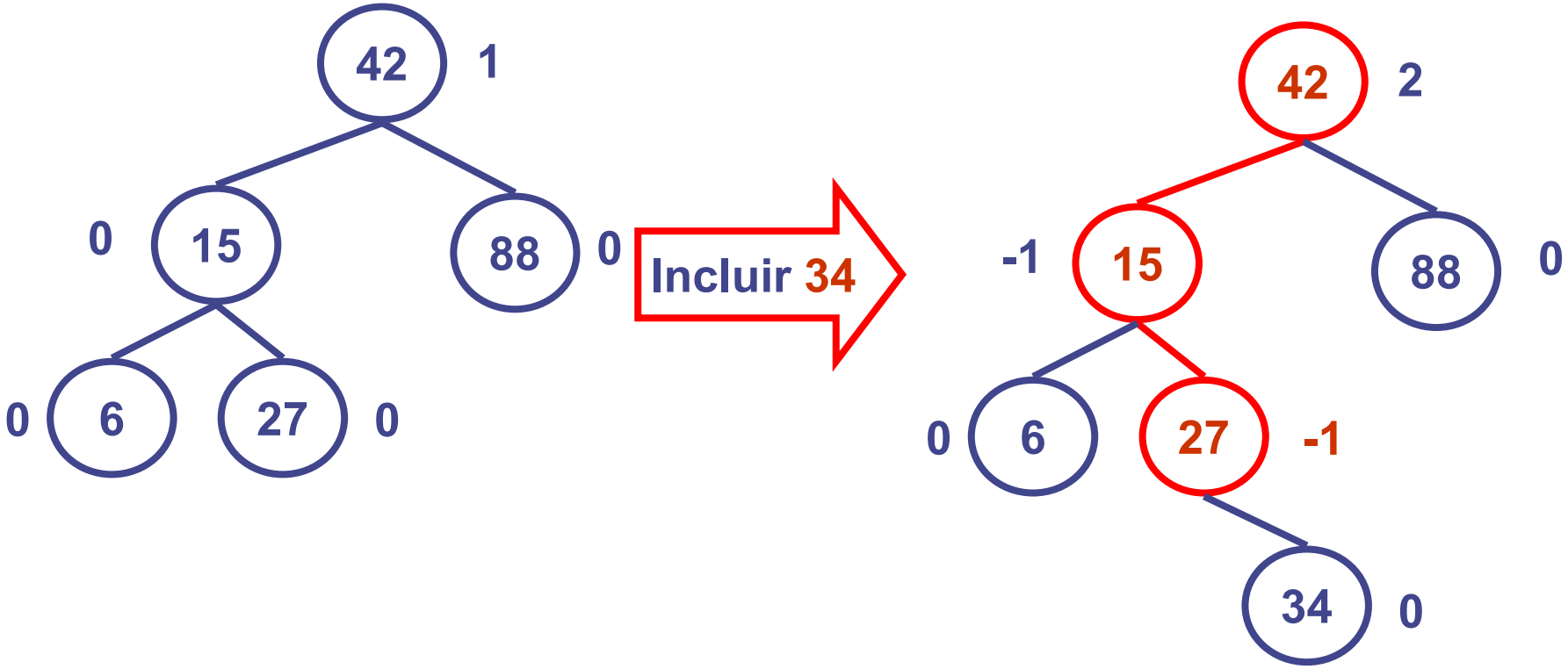


AVL – Rotação Dupla à Direita

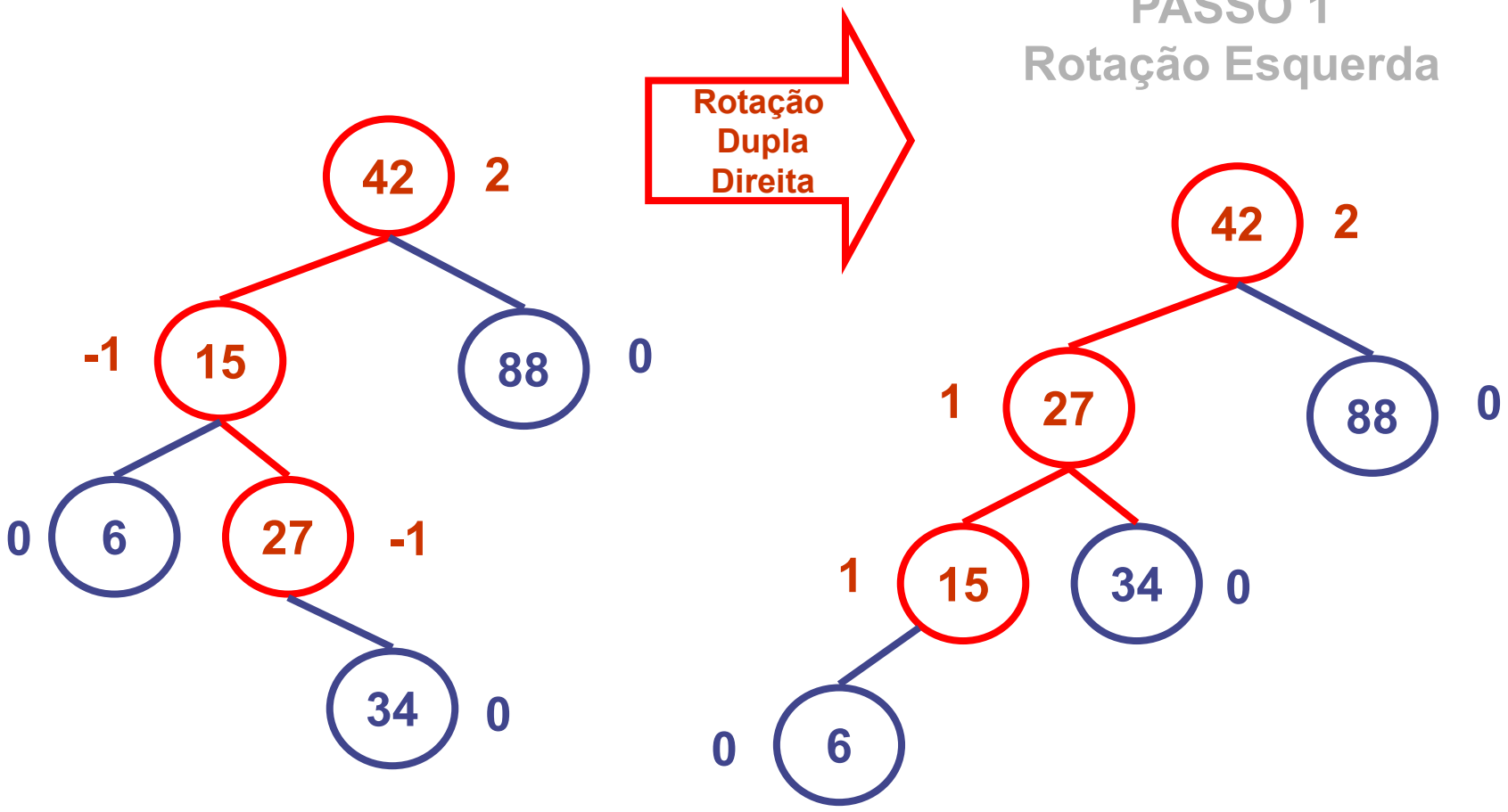



```
} else if (balanceFactor(node) > 1) {
    if (balanceFactor(node.left) < 0) {
        node.left = doLeftRotation(node.left);
    }
    node = doRightRotation(node);
}
```

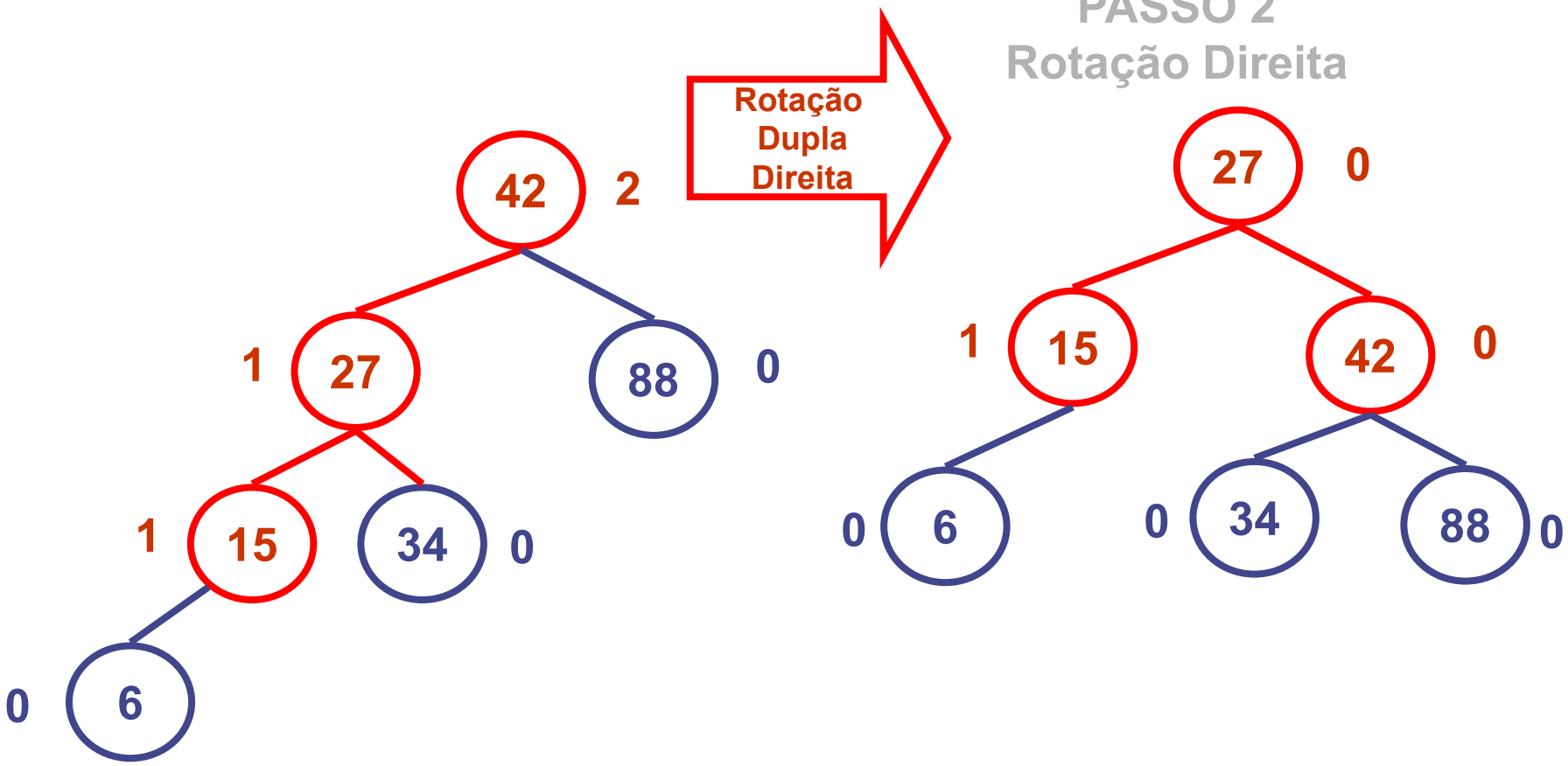
AVL – Rotação Dupla à Direita



AVL – Rotação Dupla à Direita

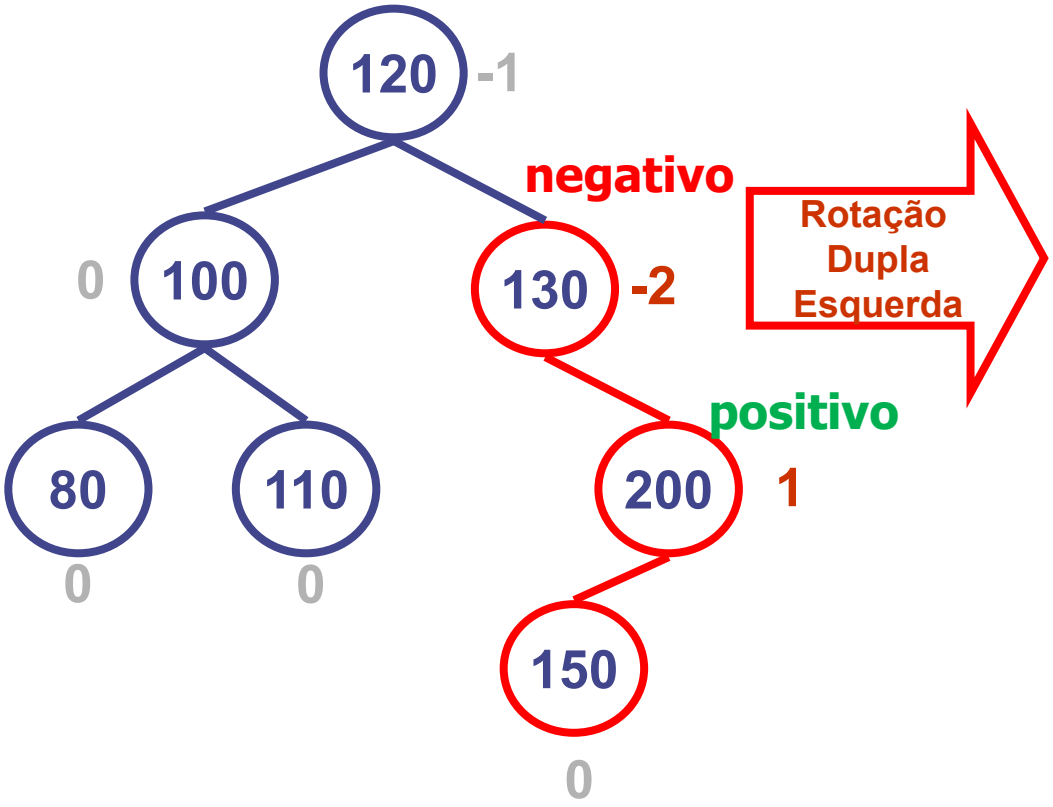


AVL – Rotação Dupla à Direita

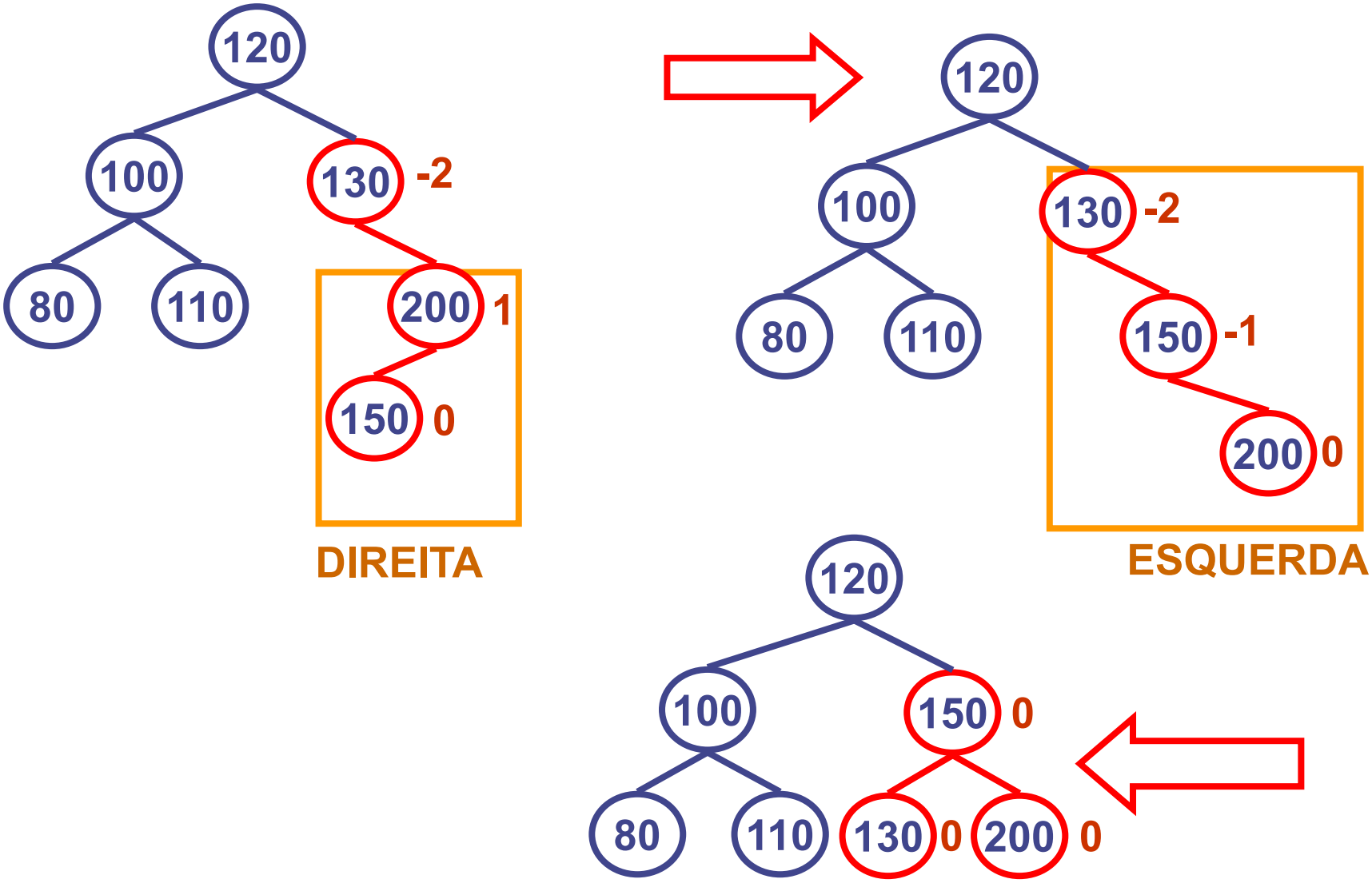


AVL – Rotação Dupla à Esquerda

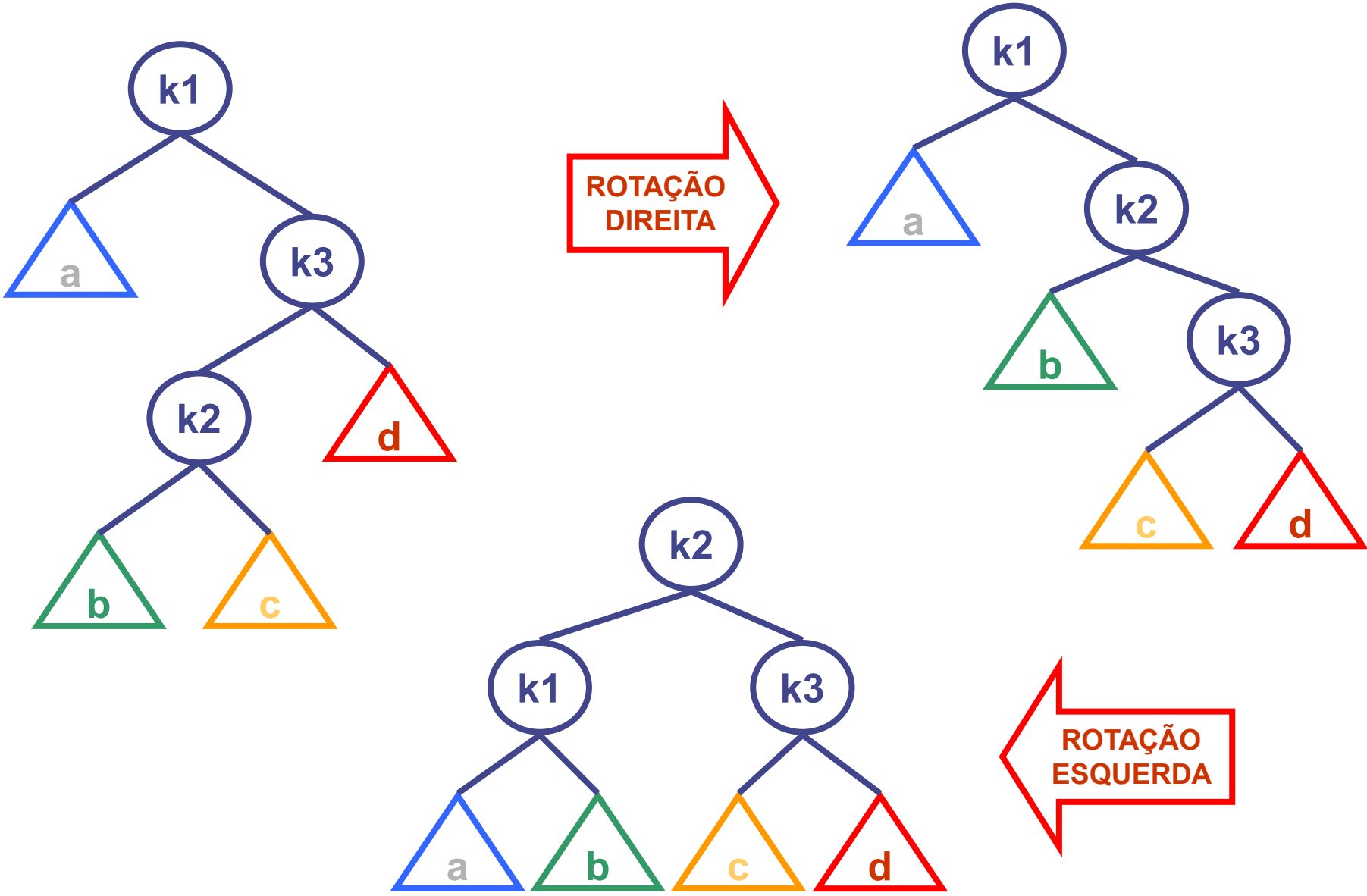
- Toda vez que uma subárvore fica com um fator:
  - **negativo** e sua subárvore da direita tem um fator **positivo**



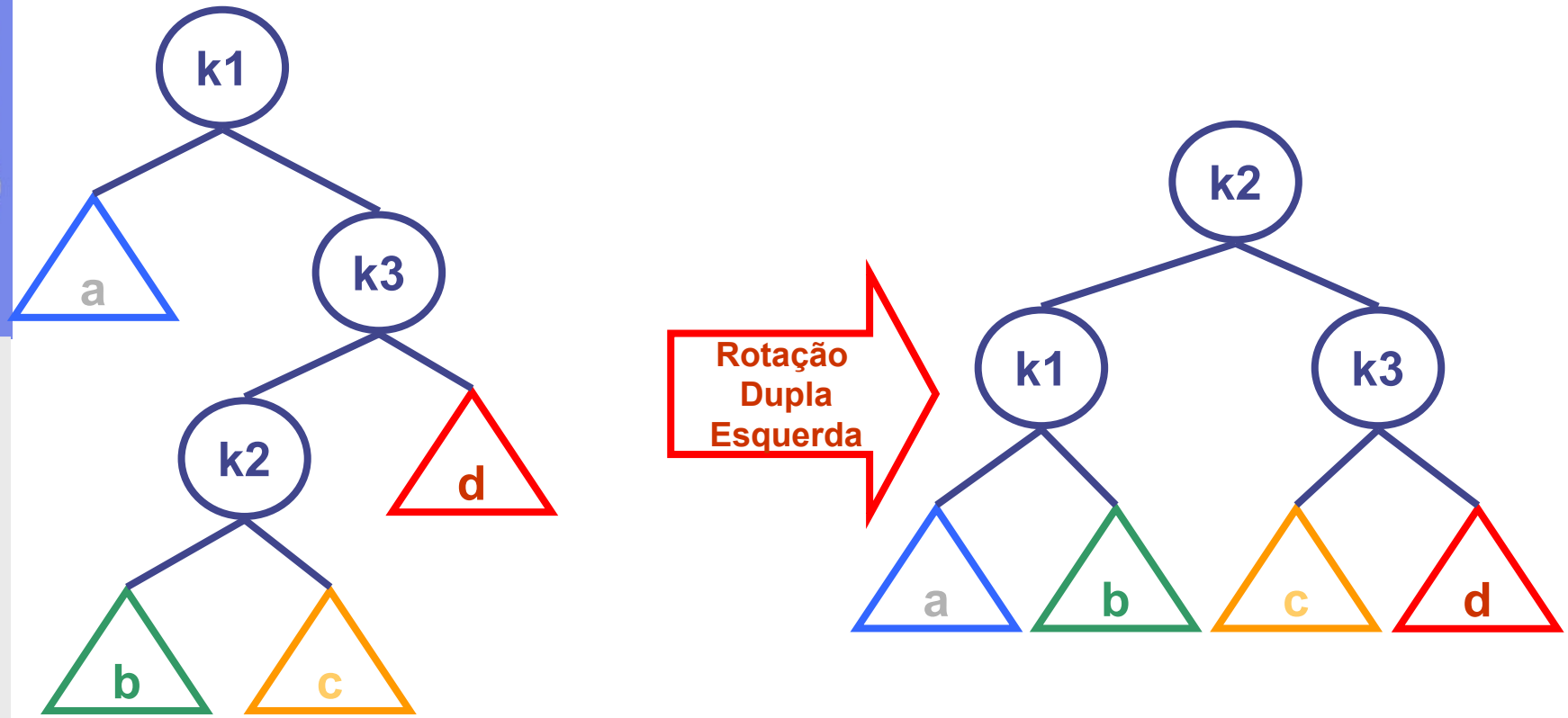
AVL – Rotação Dupla à Esquerda



AVL – Rotação Dupla à Esquerda



AVL – Rotação Dupla à Esquerda




```
if (balanceFactor(node) < -1) {
    if (balanceFactor(node.right) > 0) {
        node.right = doRightRotation(node.right);
    }
    node = doLeftRotation(node);
```

AVL – Inserção

```
public void insert(K key, V value) {
    root = insert(root, key, value);
}

private Node insert(Node node, K key, V value) {
    if (node == null) {
        return new Node(key, value);
    } else if (key.compareTo(node.key) > 0) {
        node.right = insert(node.right, key, value);
    } else if (key.compareTo(node.key) < 0) {
        node.left = insert(node.left, key, value);
    }
    node.height = 1 + Math.max(height(node.left), height(node.right));
    return balance(node);
}
```

```
private int height(Node node) {
    return node != null ? node.height : -1;
}

private int balanceFactor(Node node) {
    return height(node.left) - height(node.right);
}
```

AVL – Inserção (cont.)

```
private Node balance(Node node) {
    if (balanceFactor(node) < -1) {
        if (balanceFactor(node.right) > 0) {
            node.right = doRightRotation(node.right);
        }
        node = doLeftRotation(node);
    } else if (balanceFactor(node) > 1) {
        if (balanceFactor(node.left) < 0) {
            node.left = doLeftRotation(node.left);
        }
        node = doRightRotation(node);
    }
    return node;
}
```

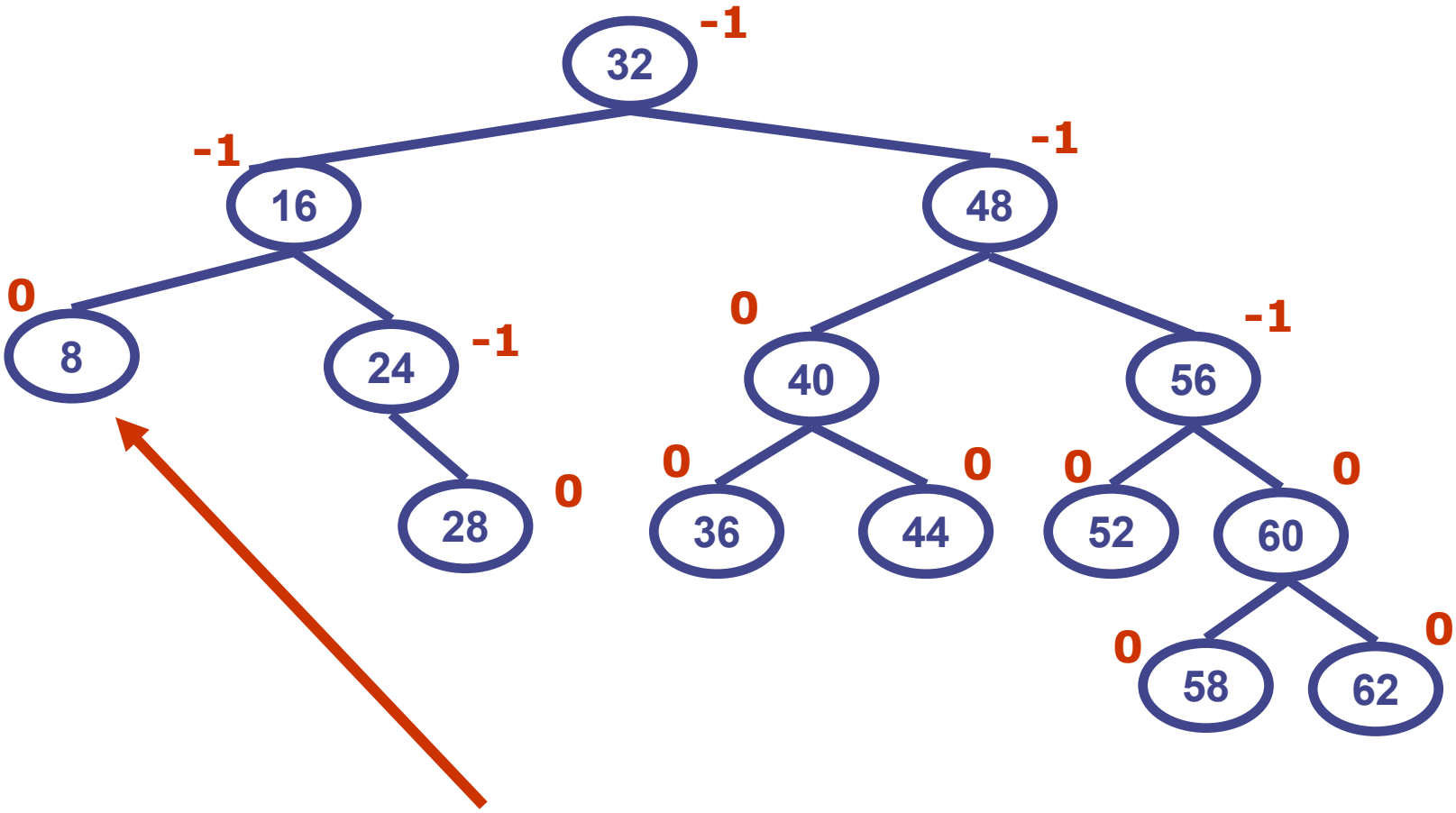
AVL – Outros Contratos

- **Search:** é o mesmo utilizado na BST;

```
@Override
public V search(K key) {
   return search(root, key);
}

private V search(Node node, K key) {
   if (node == null) {
       return null;
   } else if (key.compareTo(node.key) == 0) {
       return node.value;
   }
   return search(node.next(key), key);
}
```

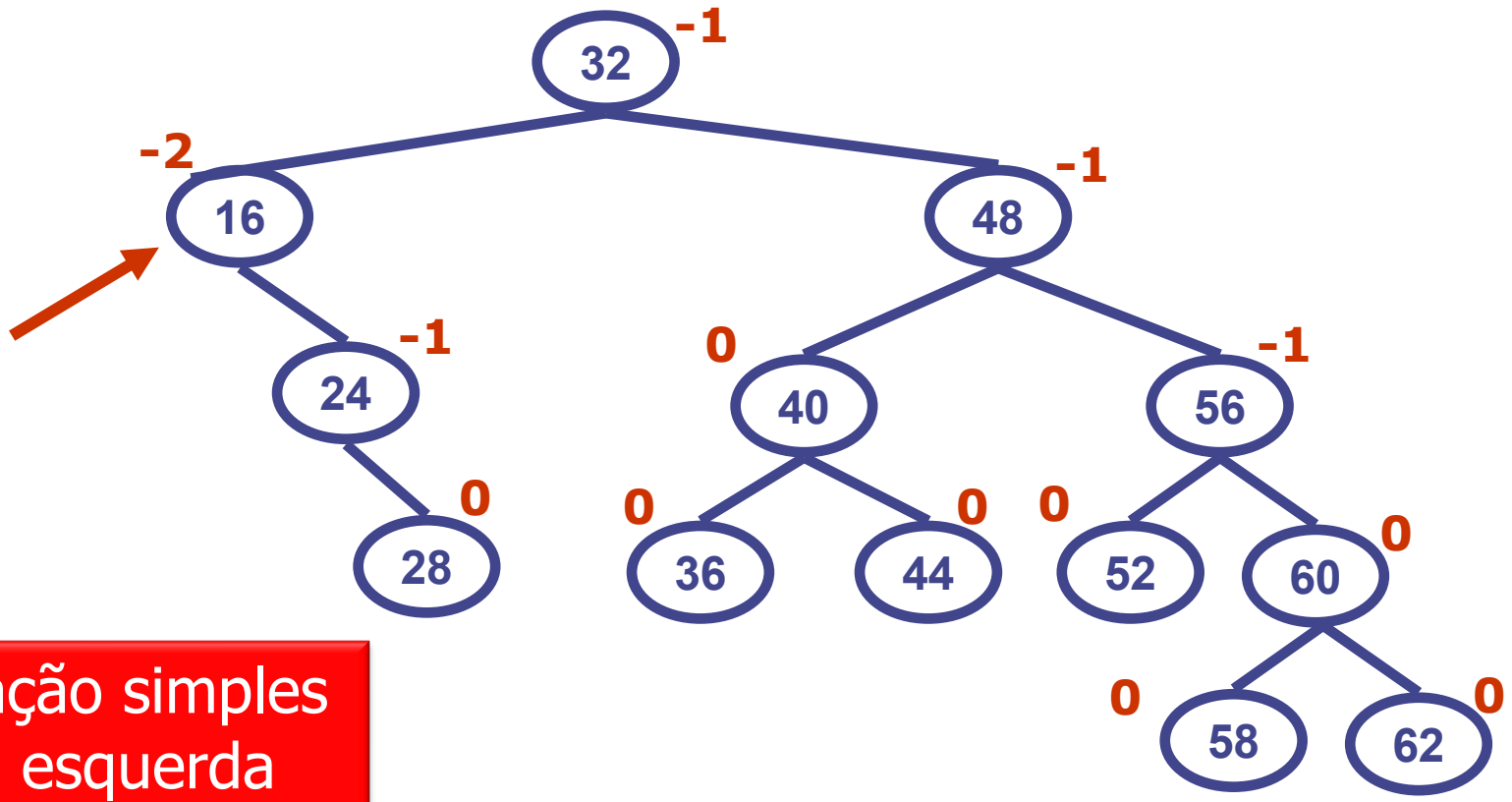
- **{Pré, in, Pós, Level}-order e toString:** são os mesmos da BST;
- **Delete:** usaremos uma variação da exclusão por cópia recursiva, pois necessitamos recalcular a altura da subárvore.

AVL – Exclusão



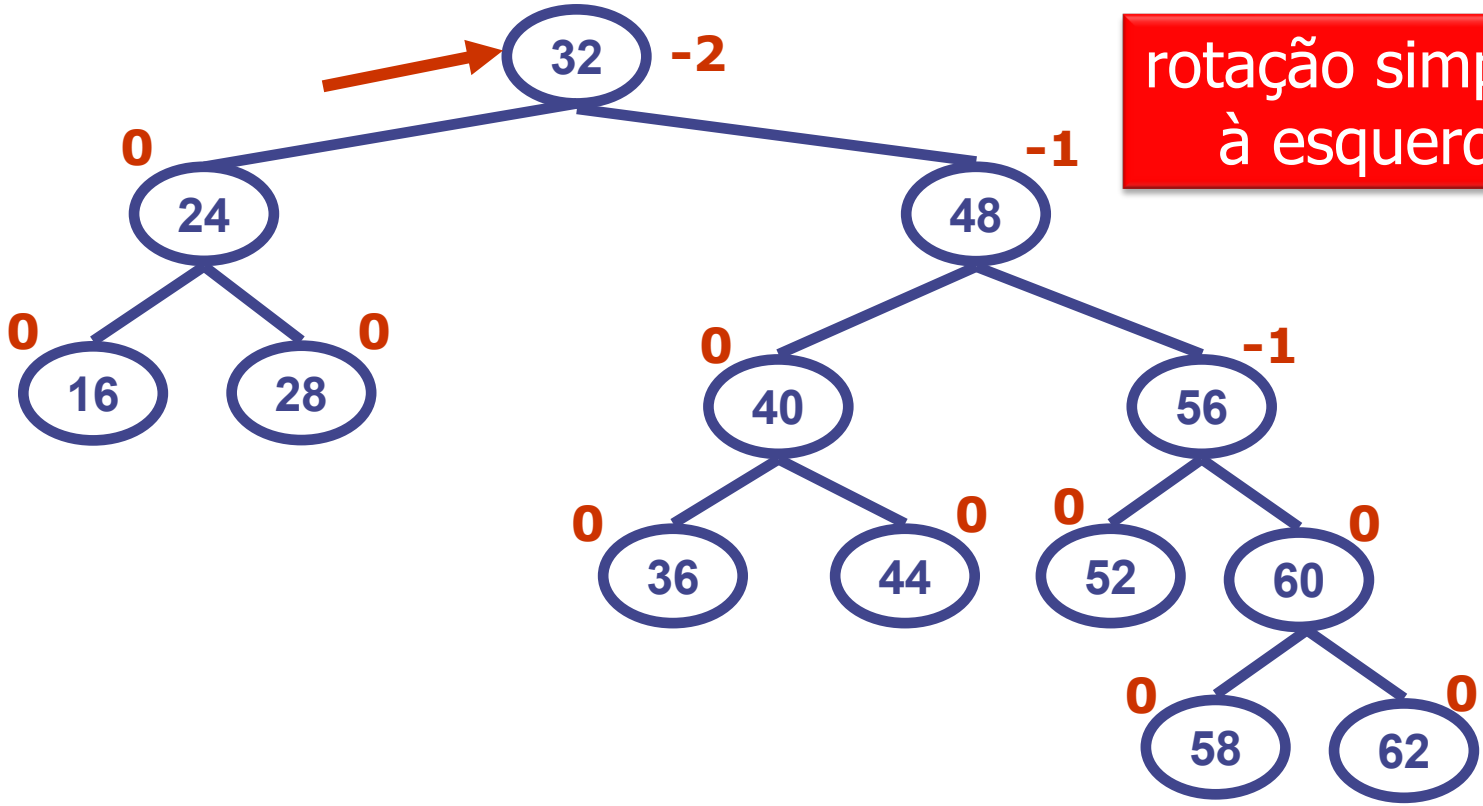
Excluindo 8

AVL – Exclusão (cont.)



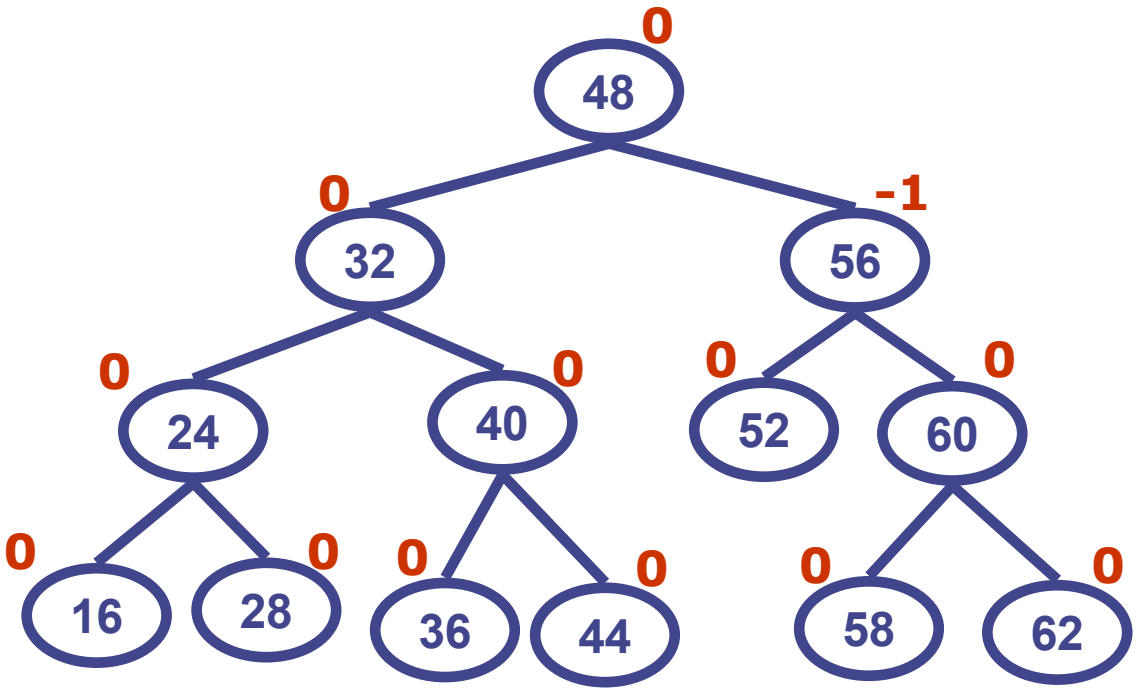
rotação simples
à esquerda

AVL – Exclusão (cont.)



Raiz está com fator -2, então, faz o balanço de todos os nós que se encontram desbalanceados. 48 será a nova raiz, e o 32 será o filho à esquerda.

AVL – Exclusão (cont.)



AVL – Exclusão (cont.)

```java
@Override
public boolean delete(K key) {
    if (search(key) != null) {
        root = deleteByCopying(root, key);
        return true;
    }
    return false;
}
```

```java
private Node deleteByCopying(Node node) {
    if (node.right == null) {
        return node.left;
    }
    node.right = deleteByCopying(node.right);
    node.height = 1 + Math.max(height(node.left), height(node.right));
    return balance(node);
}
```

AVL – Exclusão (cont.)

```
public Node deleteByCopying(Node node, K key) {
    if (key.compareTo(node.key) < 0) {
        node.left = deleteByCopying(node.left, key);
     } else if (key.compareTo(node.key) > 0) {
        node.right = deleteByCopying(node.right, key);
     } else {
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        } else {
            Node temp = node;
            node = node.left;
            while (node.right != null)
                node = node.right;

            node.left = deleteByCopying(temp.left);
            node.right = temp.right;
        }
     }
     node.height = 1 + Math.max(height(node.left), height(node.right));
     return balance(node);
}
```

AVL - Complexidade

- Busca é O(log n)
  - altura de árvore é O(log n), não necessita reestruturação
- Inserção é O(log n)
  - busca inicial é O(log n)
  - reestruturação para manter balanceamento é O(log n)
- Exclusão é O(log n)
  - busca inicial é O(log n)
  - reestruturação para manter balanceamento é O(log n)

Complexidade - Comparativo

Estrutura de Dados	Operação	Caso médio	Pior Caso
BST	Busca	O(log(n))	O(n)
	Inserção	O(log(n))	O(n)
	Exclusão	O(log(n))	O(n)
AVL	Busca	O(log(n))	O(log(n))
	Inserção	O(log(n))	O(log(n))
	Exclusão	O(log(n))	O(log(n))

AVL

Simuladores de árvore AVL:

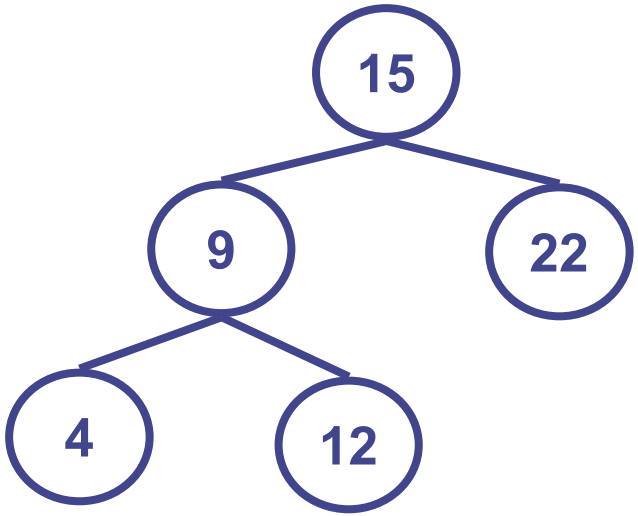
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html. Cuide que é mostrado a altura + 1 e não o fator.

https://visualgo.net/bn/bst. Tem vários tipos de árvores, inclusive a AVL. Não mostra o fator, mas mostra passo-a-passo o que está acontecendo.

http://www.cs.armstrong.edu/liang/animation/web/AVLTree.html. O mais simples.

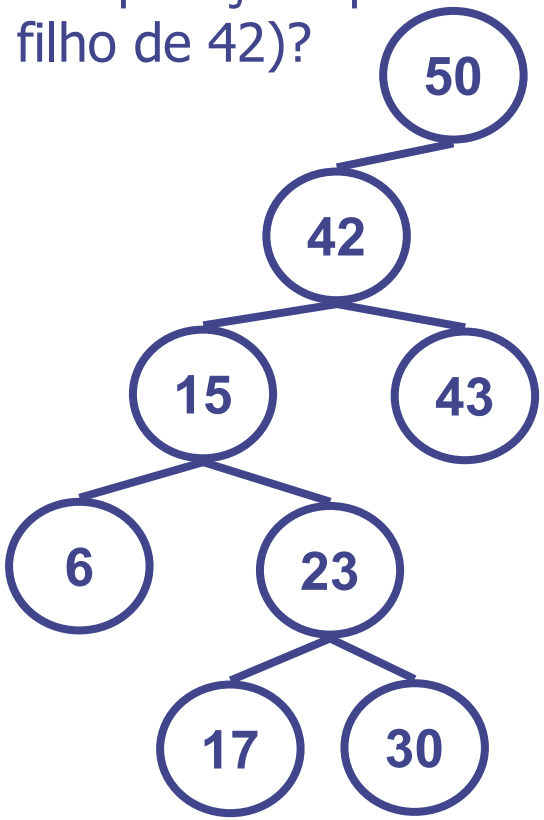
Exercícios Teóricos

- **Exercício 2.** Considere a árvore abaixo, no qual 12 está abaixo de 9 e 15.
  - Fazendo a rotação à direita em 9 onde ficará 12?
  - Terminada a rotação à direita, tente agora a rotação à esquerda em 15.



Exercícios Teóricos

- **Exercício 3.** Considere a árvore abaixo:
  - Execute a rotação à esquerda em 15. O que acontece com o nó 42 (que é pai de 15)?
  - Execute agora a operação oposta em 42. O que acontece com 15 (que é filho de 42)?



Exercícios Teóricos

- **Exercício 4.** Monte a **árvore AVL** (passo a passo) para as seguintes chaves: 10, 20, 30, 40 e 45 (nesta ordem) indicando em cada passo:
  1) Qual elemento foi inserido
  2) O grau de balanceamento de cada nó
  3) Qual rotação foi realizada, se necessária.

- **Exercício 5.** Monte a **árvore AVL** (passo a passo) para as seguintes inserções de chaves 10, 15, 7, 25, 30, 27, 49 (nesta ordem), indicando a cada passo qual elemento foi inserido e qual rotação foi realizada.

Referências Bibliográficas

- ASCENCIO, A. F. G; ARAÚJO, G. S. **Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010. 432 p.

- CORMEN, Thomas H. et al. **Introduction to algorithms.** 3. ed. Cambridge: MIT, 2009. xix. 1292 p.

- JAQUES, Patrícia. **Lâminas Árvores Binárias – Programação II**, Unisinos.