ESTRUTURAS DE DADOS

2024/2025

Aula 10

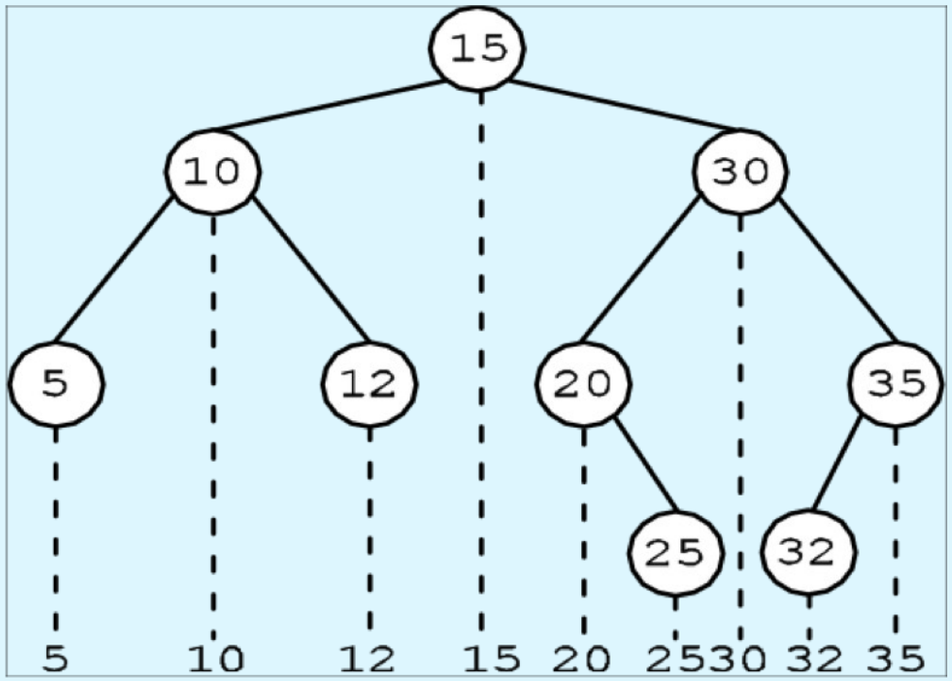
- Árvores Binárias de Pesquisa
- Balanceamento
- Árvores AVL
- Árvores Binárias de Pesquisa na Plataforma de Colecções do *Java*

Ricardo Santos | rjs@estg.ipp.pt

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Árvores Binárias de Pesquisa

- Uma árvore binária de pesquisa é uma árvore binária em que os elementos são ordenados de forma a que:



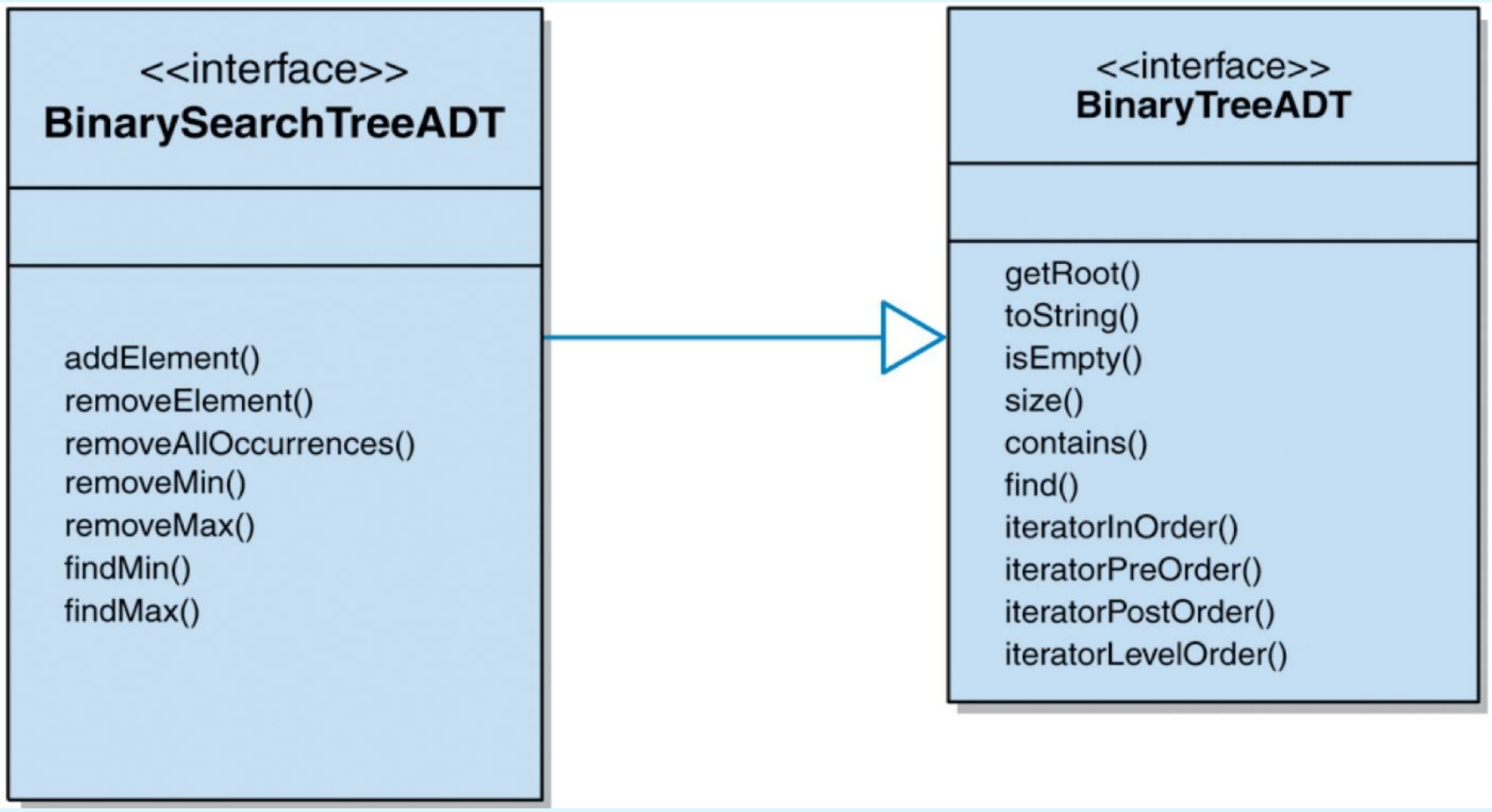
- Para qualquer nó numa árvore, todos os elementos nos nós da sub-árvore esquerda são menores que o elemento do nó e todos os elementos da sub-árvore direita são maiores ou iguais ao elemento do nó
  - Esta será a propriedade das árvores binárias de pesquisa.
  - As sub-árvores esquerda e direita são árvores binárias de pesquisa

- **Dado este refinamento para a nossa definição anterior de uma árvore binária, agora podemos incluir operações adicionais**

Operações de uma Árvore Binária de Pesquisa

Operação	Descrição
`addElement`	Adiciona um elemento à árvore
`removeElement`	Remove um elemento da árvore
`removeAllOccurrences`	Remove todas as ocorrências do elemento da árvore
`removeMin`	Remove o menor elemento na árvore
`removeMax`	Remove o maior elemento na árvore
`findMin`	Retorna uma referência do menor elemento na árvore
`findMax`	Retorna uma referência do maior elemento na árvore

Interface
BinarySearchTreeADT



Interface BinarySearchTreeADT

```
/**
 * BinarySearchTreeADT defines the interface to a binary search tree.
 *
 */

public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T> {

  /**
   * Adds the specified element to the proper location in this tree.
   *
   * @param element  the element to be added to this tree
   */
  public void addElement (T element);
```

```
/**
 * Removes and returns the specified element from this tree.
 *
 * @param targetElement  the element to be removed from this tree
 * @return               the element removed from this tree
 */
public T removeElement (T targetElement);

/**
 * Removes all occurences of the specified element from this tree.
 *
 * @param targetElement   the element that the list will
 *                        have all instances of it removed
 */
public void removeAllOccurrences (T targetElement);

/**
 * Removes and returns the smallest element from this tree.
 *
 * @return  the smallest element from this tree.
 */
public T removeMin();
```

```
    /**
     * Removes and returns the largest element from this tree.
     *
     * @return  the largest element from this tree
     */
    public T removeMax();

    /**
     * Returns a reference to the smallest element in this tree.
     *
     * @return  a reference to the smallest element in this tree
     */
    public T findMin();

    /**
     * Returns a reference to the largest element in this tree.
     *
     * @return  a reference to the largest element in this tree
     */
    public T findMax();

}
```

Implementar Árvores Binárias de Pesquisa com Listas Ligadas

- Podemos simplesmente estender a nossa definição de uma `LinkedBinaryTree` para criar uma `LinkedBinarySearchTree`

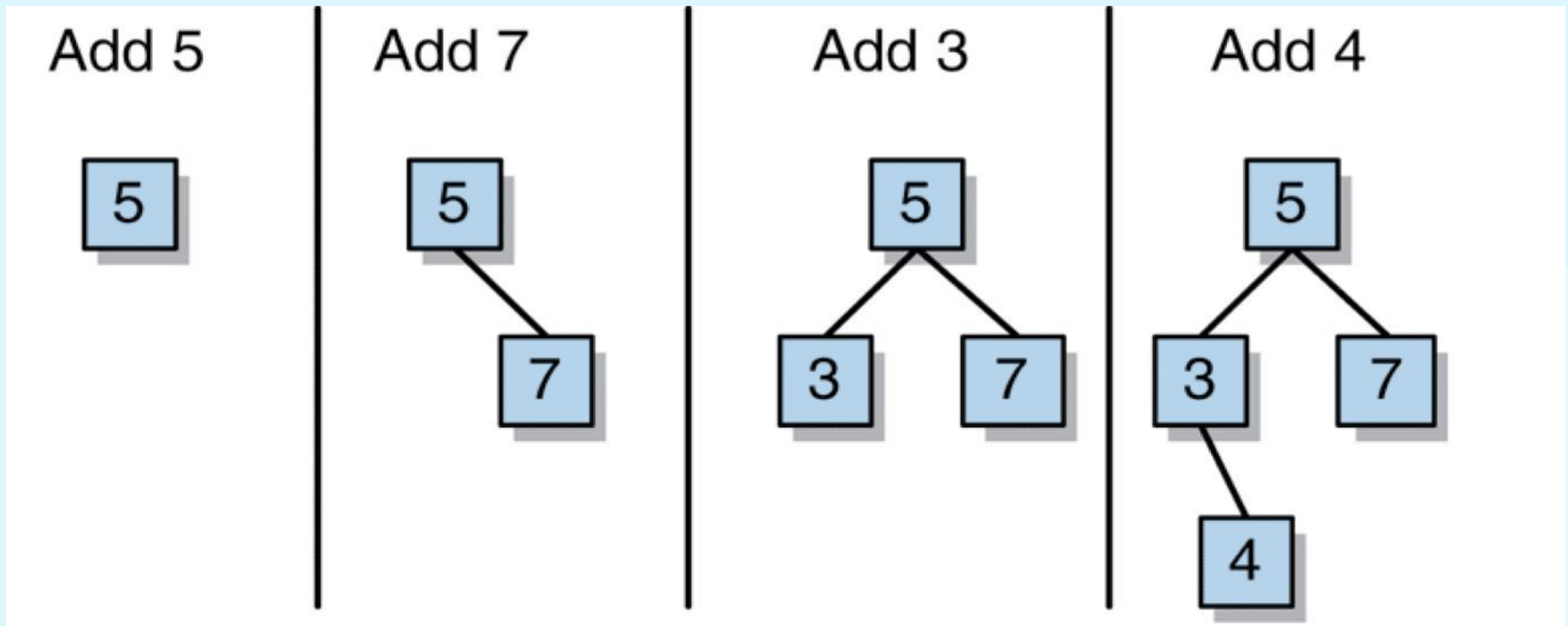
- Esta classe irá fornecer dois constructores, um para criar uma árvore vazia e o outro para criar uma de árvore binária de um elemento

```
public class LinkedBinarySearchTree<T>  extends LinkedBinaryTree<T>
                                implements BinarySearchTreeADT<T> {
   /**
    * Creates an empty binary search tree.
    */
   public LinkedBinarySearchTree() {

      super();
   }

   /**
    * Creates a binary search with the specified element as its root.
    *
    * @param element  the element that will be the root of the new
    * binary search tree
    */
   public LinkedBinarySearchTree (T element) {

      super (element);
   }
```

- Agora que sabemos mais sobre como esta árvore deverá ser usada (e estruturada), é possível definir um método para adicionar um elemento à árvore

- O método `addElement` encontra o local apropriado para o elemento dado e adiciona-o (no local) como uma folha

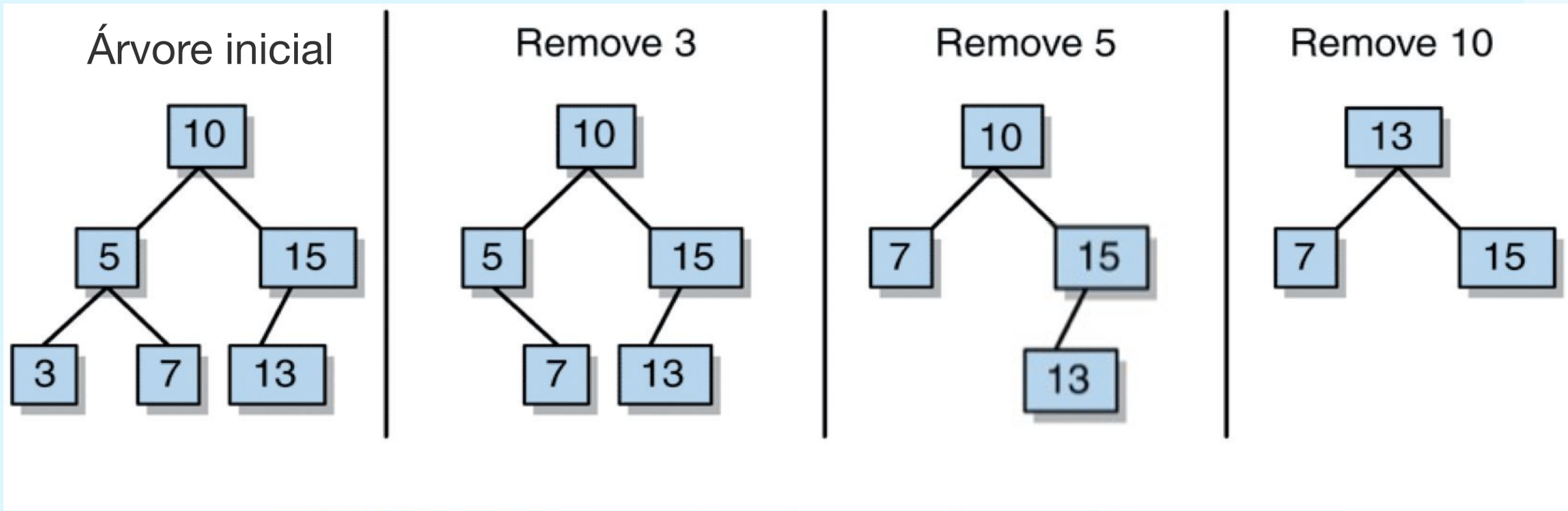
Adicionar Elementos



```
/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its key value.  Note that
 * equal elements are added to the right.
 *
 * @param element  the element to be added to the binary search
 * tree
 */
public void addElement (T element) {
   BinaryTreeNode<T> temp = new BinaryTreeNode<T> (element);
   Comparable<T> comparableElement = (Comparable<T>)element;

   if (isEmpty())
      root = temp;
   else {
      BinaryTreeNode<T> current = root;
      boolean added = false;
      while (!added) {
         if (comparableElement.compareTo(current.element) < 0) {
            if (current.left == null)
```

```
            {
                current.left = temp;
                added = true;
            }
            else
                current = current.left;
        }
        else
        {
            if (current.right == null)
            {
                current.right = temp;
                added = true;
            }
            else
                current = current.right;
        }
    }
    count++;
}
```

Remover Elementos



- Remover elementos de uma árvore de binária de pesquisa requer:
  - Encontrar o elemento a ser removido
  - Se esse elemento não é uma folha, de seguida substituí-la com o seu sucessor ***inorder***
  - Retornar o elemento removido
- O método `removeElement` faz uso de um método de substituição privado (`replacement`) para encontrar o elemento adequado para substituir um elemento não-folha que é removido

```
/**
   * Removes the first element that matches the specified target
   * element from the binary search tree and returns a reference to
   * it.  Throws a ElementNotFoundException if the specified target
   * element is not found in the binary search tree.
   *
   * @param targetElement   the element being sought in the binary
   *                        search tree
   * @throws ElementNotFoundException  if an element not found
   *                                   exception occurs
   */
public T removeElement (T targetElement)
                        throws ElementNotFoundException {
   T result = null;
   if (!isEmpty())
   {
      if (((Comparable)targetElement).equals(root.element))
      {
         result =  root.element;
         root = replacement (root);
         count--;
      }
```

```
else
      {
         BinaryTreeNode<T> current, parent = root;
         boolean found = false;

         if (((Comparable)targetElement).compareTo(root.element) < 0)
            current = root.left;
         else
            current = root.right;

      while (current != null && !found)
      {
         if (targetElement.equals(current.element))
         {
            found = true;
            count--;
            result =  current.element;

            if (current == parent.left)
            {
               parent.left = replacement (current);
            }
```

```
else
                {
                   parent.right = replacement (current);
                }
             }
             else
             {
                parent = current;

                if (((Comparable)targetElement).compareTo(current.element) < 0)
                   current = current.left;
                else
                   current = current.right;
             }
          } //while

          if (!found)
             throw new ElementNotFoundException("binary search tree");
       }
    } // end outer if
    return result;
 }
```

```
/**
 * Returns a reference to a node that will replace the one
 * specified for removal.  In the case where the removed node has
 * two children, the inorder successor is used as its replacement.
 *
 * @param node  the node to be removeed
 * @return      a reference to the replacing node
 */
protected BinaryTreeNode<T> replacement (BinaryTreeNode<T> node)
{
   BinaryTreeNode<T> result = null;

   if ((node.left == null)&&(node.right==null))
      result = null;

   else if ((node.left != null)&&(node.right==null))
      result = node.left;

   else if ((node.left == null)&&(node.right != null))
      result = node.right;
```

```
else
      {
         BinaryTreeNode<T> current = node.right;
         BinaryTreeNode<T> parent = node;
         while (current.left != null)
         {
            parent = current;
            current = current.left;
         }
         if (node.right == current)
            current.left = node.left;

         else
         {
            parent.left = current.right;
            current.right = node.right;
            current.left = node.left;
         }
         result = current;
      }
      return result;
   }
```

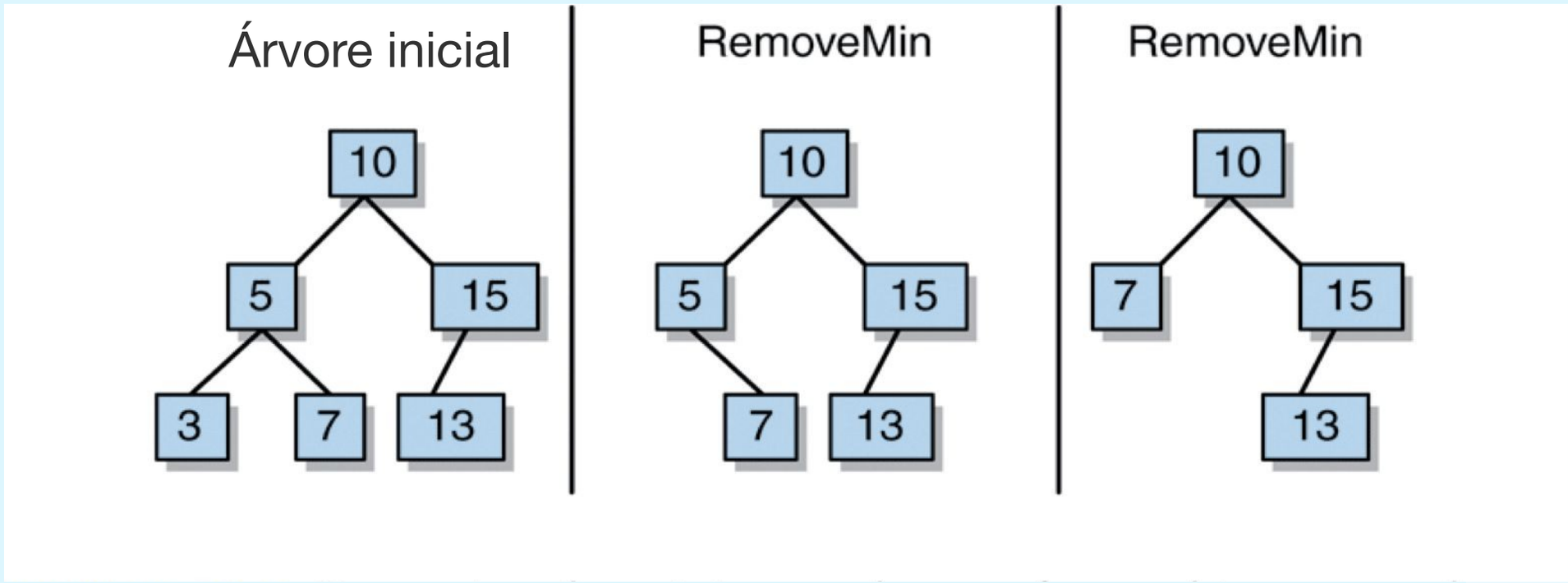
Remover todas as Ocorrências

- O método `removeAllOccurrences` remove todas as ocorrências de um elemento da árvore
- Este método usa o método `removeElement`
- Este método faz uma distinção entre a primeira chamada e as sucessivas chamadas ao método `removeElement`
- **Exercício:** Implementar este método `removeAllOccurrences`

Remover o elemento mínimo de uma Árvore Binária de Pesquisa

- Existem três casos para a localização do mínimo elemento de uma árvore binária de pesquisa:
  - Se a raiz não tem filho esquerdo, então a raiz é o elemento mínimo e o filho à direita da raiz torna-se a nova raiz
  - Se o nó mais à esquerda da árvore é uma folha, então colocamos a referência do filho esquerdo do pai a *null*
  - Se o nó mais à esquerda da árvore é um nó interno, então colocamos a referência do filho esquerdo do pai a apontar para o filho direito do nó a ser removido

Remover o elemento mínimo de uma Árvore Binária de Pesquisa



- **Exercício:** Implementar este método `removeMin`

Implementar Árvores Binárias de Pesquisa com *arrays*

- Tal como fizemos para a `LinkedBinarySearchTree` podemos estender a nossa `ArrayBinaryTree` para criar uma `ArrayBinarySearchTree`

Classe
ArrayBinarySearchTree

```
/**
 * ArrayBinarySearchTree implements a binary search tree
 * using an array.
 *
 */

public class ArrayBinarySearchTree<T>  extends ArrayBinaryTree<T>
                                 implements BinarySearchTreeADT<T>
{
   protected int height;
   protected int maxIndex;
```

```
/**
 * Creates an empty binary search tree.
 */
public ArrayBinarySearchTree()
{
   super();
   height = 0;
   maxIndex = -1;
}

/**
 * Creates a binary search with the specified element as its root
 *
 * @param element  the element that will become the root of
 * the new tree
 */
public ArrayBinarySearchTree (T element)
{
   super(element);
   height = 1;
   maxIndex = 0;
}
```

```
/**
 * Adds the specified object to this binary search tree in the
 * appropriate position according to its key value.  Note that
 * equal elements are added to the right.  Also note that the
 * index of the left child of the current index can be found by
 * doubling the current index and adding 1.  Finding the index
 * of the right child can be calculated by doubling the current
 * index and adding 2.
 *
 * @param element  the element to be added to the search tree
 */
public void addElement (T element)
{
   if (tree.length < maxIndex*2+3)
      expandCapacity();
   Comparable<T> tempelement = (Comparable<T>)element;

   if (isEmpty())
   {
      tree[0] = element;
      maxIndex = 0;
   }
   else
```

```
{
        boolean added = false;
        int currentIndex = 0;

        while (!added)
        {
           if (tempelement.compareTo((tree[currentIndex]) ) < 0)
           {
              /** go left */
              if (tree[currentIndex*2+1] == null)
              {
                 tree[currentIndex*2+1] = element;
                 added = true;
                 if (currentIndex*2+1 > maxIndex)
                    maxIndex = currentIndex*2+1;
              }
              else
                 currentIndex = currentIndex*2+1;
        }
```

```
            else
            {
               /** go right */
               if (tree[currentIndex*2+2] == null)
               {
                  tree[currentIndex*2+2] = element;
                  added = true;
                  if (currentIndex*2+2 > maxIndex)
                     maxIndex = currentIndex*2+2;
               }
               else
                  currentIndex = currentIndex*2+2;
            }
         }
      }

      height = (int)(Math.log(maxIndex + 1) / Math.log(2)) + 1;
      count++;
   }
```

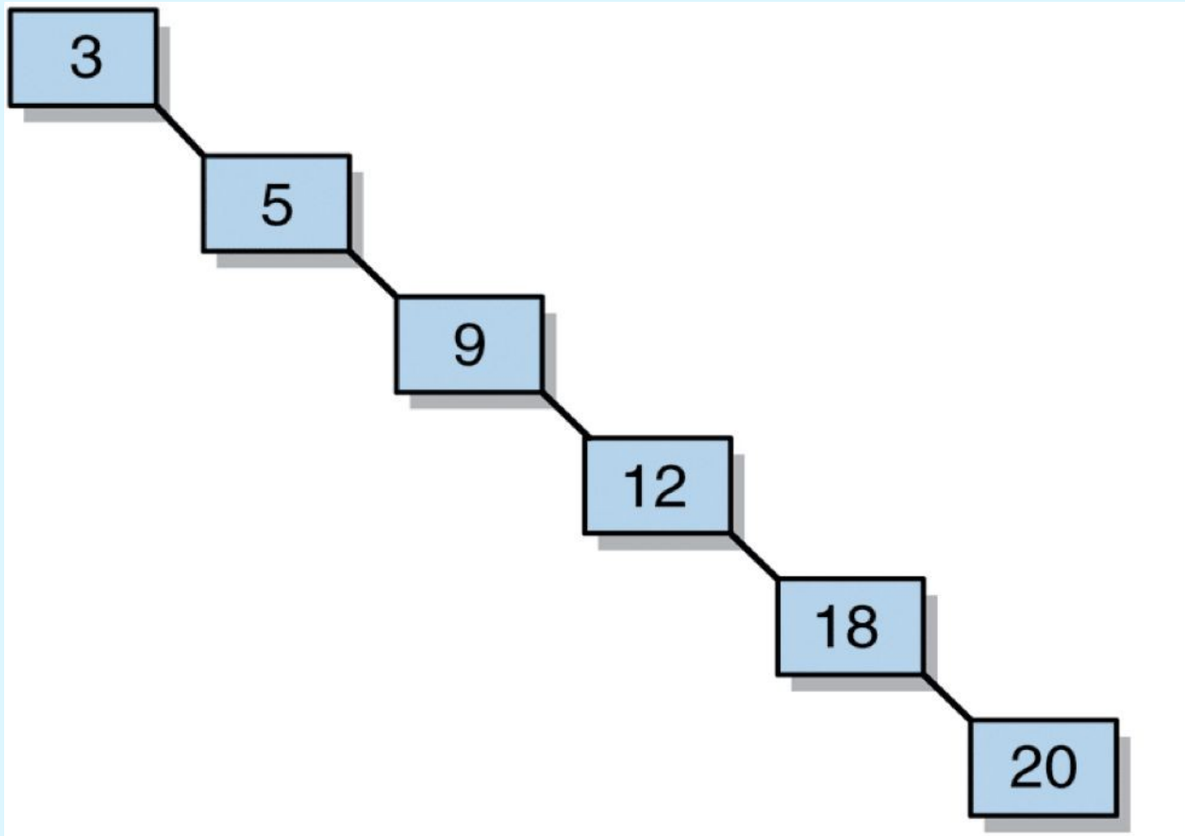
- **Exercício:** Implementar uma lista ordenada com recurso a uma árvore binária de pesquisa

Árvores Binárias de Pesquisa Balanceadas

- Porquê que o nosso pressuposto de balanceamento é tão importante?
- O que acontece se inserirmos os seguintes números na respectiva ordem sem fazermos o balanceamento da árvore:

```
3  5  9  12  18  20
```

Árvore Binária Degenerada



Árvore Binária Degenerada

- A árvore resultante é chamada de árvore binária degenerada

- As árvores binárias de pesquisa degeneradas são muito menos eficientes do que as árvores binárias de pesquisa balanceadas (**O(n)** na localização em oposição a **O(logn)**)

Balancear Árvores Binárias

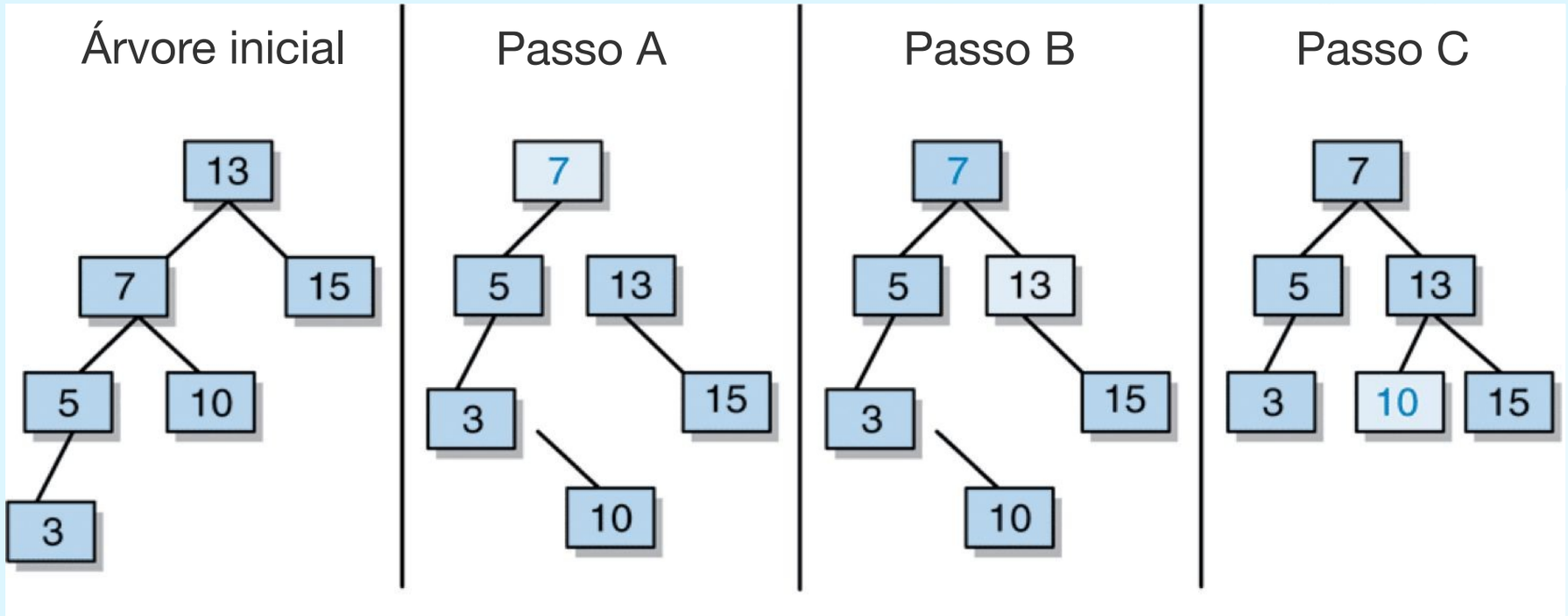
- Existem muitas abordagens para balancear árvores binárias
- Um método é a força bruta
  - Escrever uma travessia em-ordem para um ficheiro
  - Usar uma pesquisa recursiva binária do ficheiro para reconstruir a árvore

- Soluções melhores envolvem algoritmos, tais como **árvores vermelhas e pretas** e as **árvores AVL** que persistentemente mantêm o equilíbrio da árvore
- A maioria de todos estes algoritmos fazem uso de rotações para balancear a árvore
- Vamos analisar cada uma das possíveis rotações

Rotação à Direita

- A rotação à direita vai resolver um não balanceamento se for causado por um longo caminho na sub-árvore esquerda do filho esquerdo da raiz
  - Fazer do elemento filho esquerdo da raiz o novo elemento raiz.
  - Fazer do antigo elemento raiz, filho à direita da nova raiz.
  - Fazer do antigo filho direito da nova raiz, filho esquerdo da antiga raiz

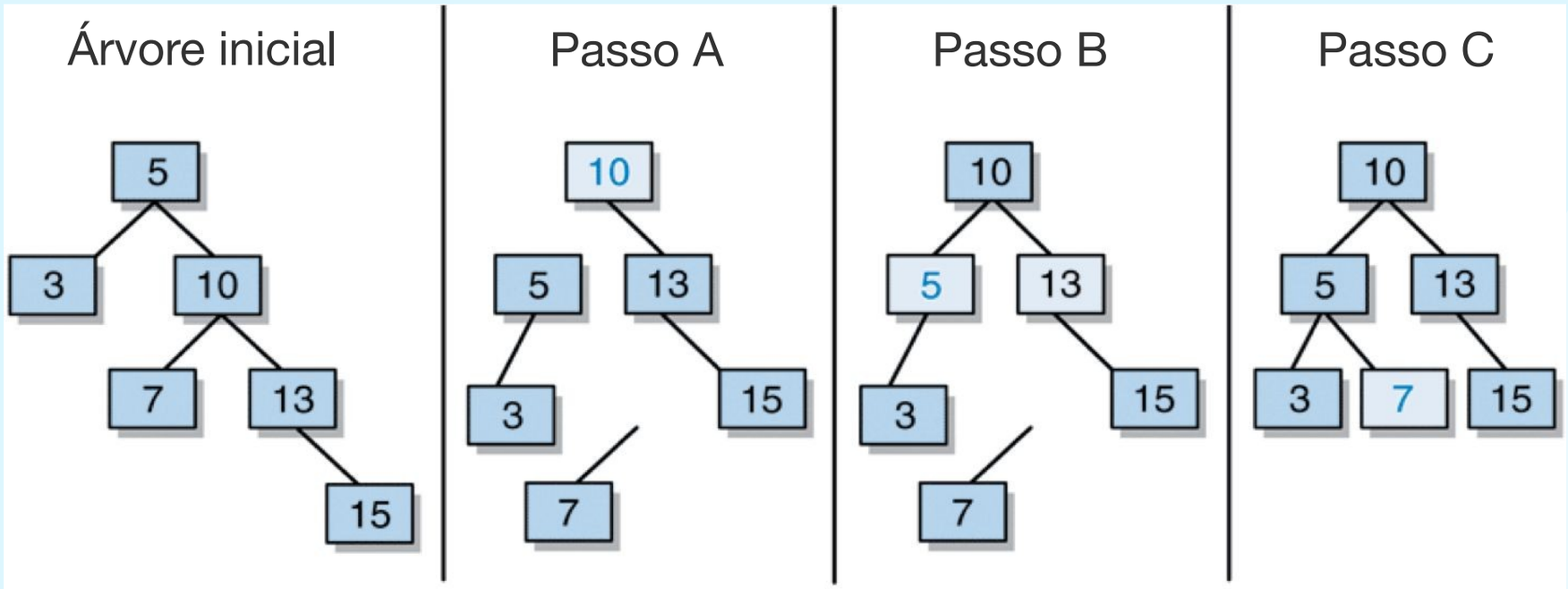
Balancear uma Árvore depois de uma Rotação à Direita



Rotação à Esquerda

- A rotação à esquerda vai resolver um desequilíbrio que é causada por um longo caminho na sub-árvore direita do filho direito da raiz
  - Fazer do elemento filho direito da raiz o novo elemento raiz
  - Fazer do antigo elemento raiz, filho à esquerda da nova raiz.
  - Fazer do antigo filho esquerdo da nova raiz, filho direito da antiga raiz

Balancear uma Árvore depois de uma Rotação à Esquerda

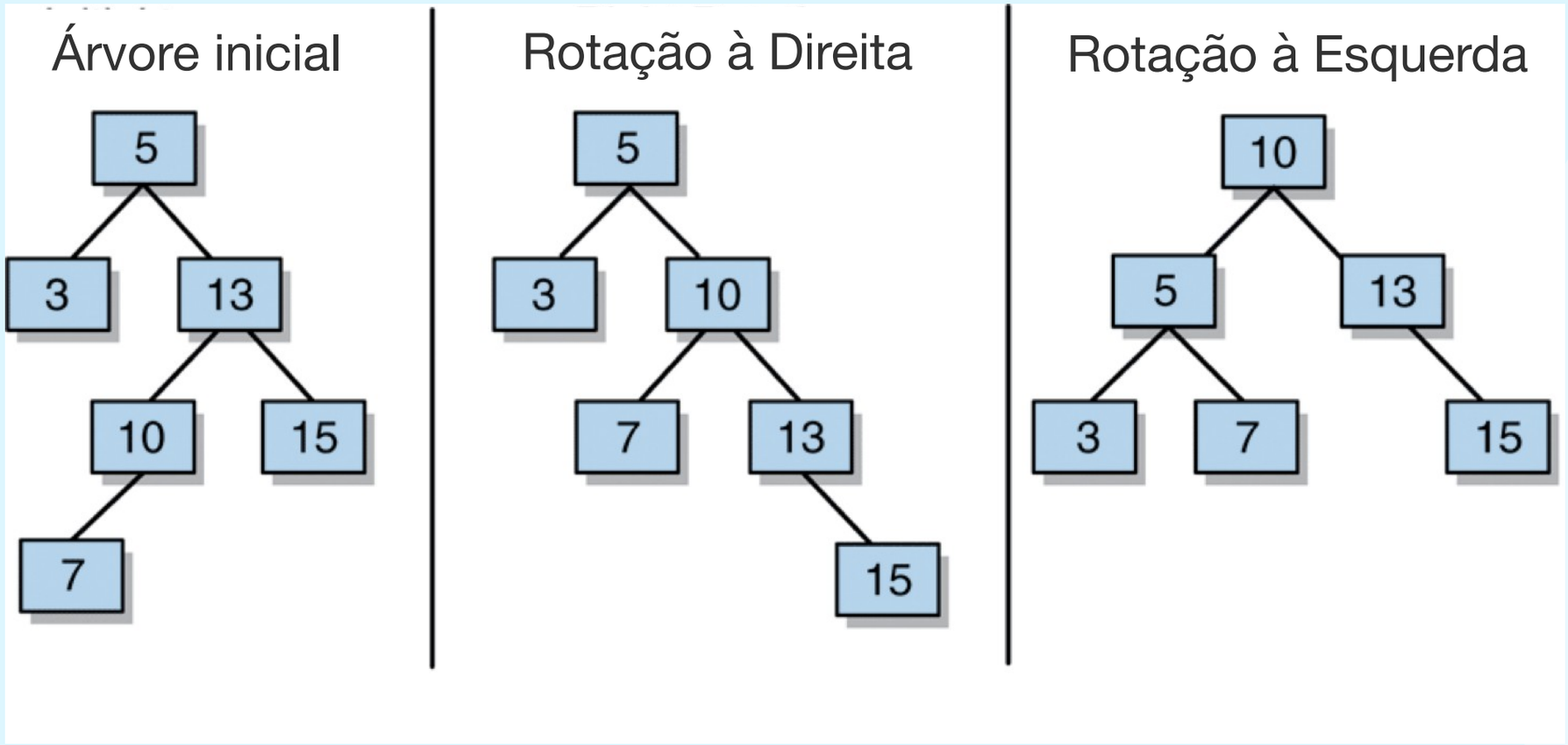


Rotação Direita-Esquerda

- A rotação Direita-Esquerda vai resolver um não balanceamento se for causado por um longo caminho na sub-árvore esquerda do filho à direita da raiz

- Executar uma rotação à direita do filho esquerdo do filho direito da raiz em torno do filho à direita da raiz e, em seguida realizar uma rotação à esquerda do filho direito resultante da raiz em torno da raiz

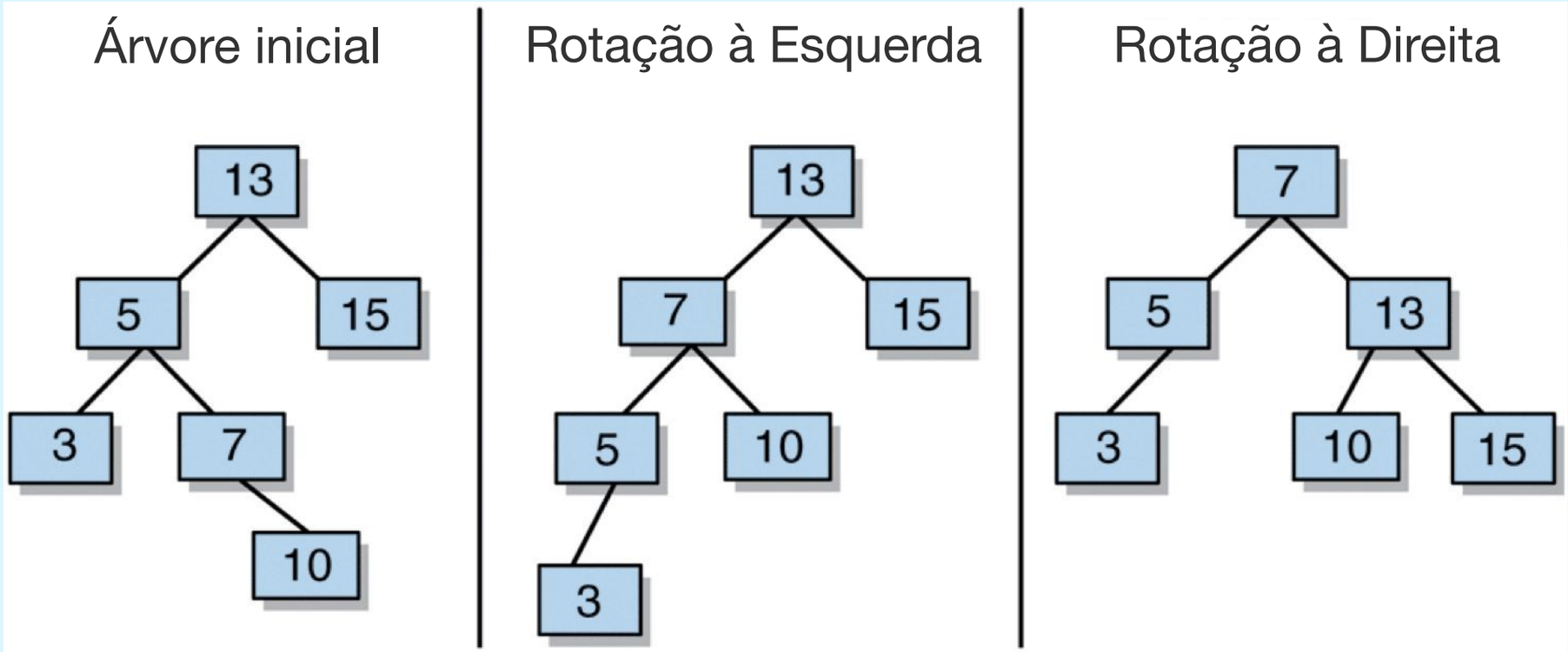
Rotação Direita-Esquerda



Rotação Esquerda-Direita

- A rotação Esquerda-Direita vai resolver um não balanceamento, se for causado por um longo caminho na sub-árvore direita do filho esquerdo da raiz
- Realizar uma rotação à esquerda do filho direito do filho esquerdo da raiz em torno do filho esquerdo da raiz, e, em seguida, executar uma rotação à direita do filho esquerdo resultante da raiz em torno da raiz

Rotação Esquerda-Direita

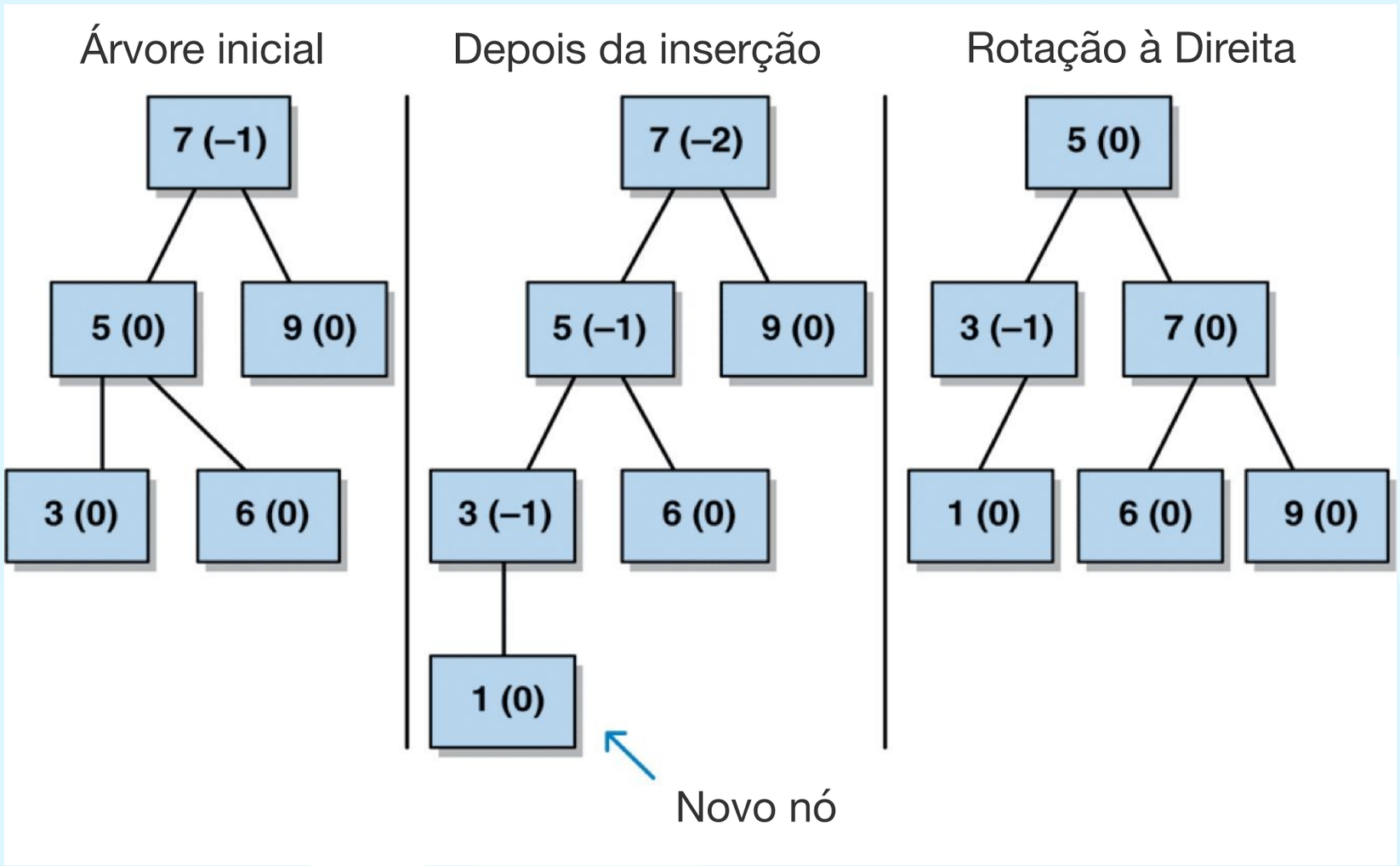


Árvores AVL

- As árvores AVL acompanham a diferença de altura entre as sub-árvores direita e esquerda de cada nó
- Esta diferença é chamada de factor de balanceamento
- Se o factor de balanceamento de qualquer nó é menor que -1 ou maior que 1, então, a sub-árvore precisa ser balanceada
- O factor de balanceamento de qualquer nó só pode ser alterado, quer através de inserção ou remoção de nós na árvore

- Se o factor de balanceamento de um nó é -2, significa que a sub-árvore esquerda tem um caminho que é longo demais
- Se o factor de balanceamento do filho esquerdo é -1, significa que o caminho longo é o da sub-árvore esquerda do filho esquerdo
- Neste caso, uma rotação simples à direita do filho esquerdo em torno do nó original irá resolver o balanceamento

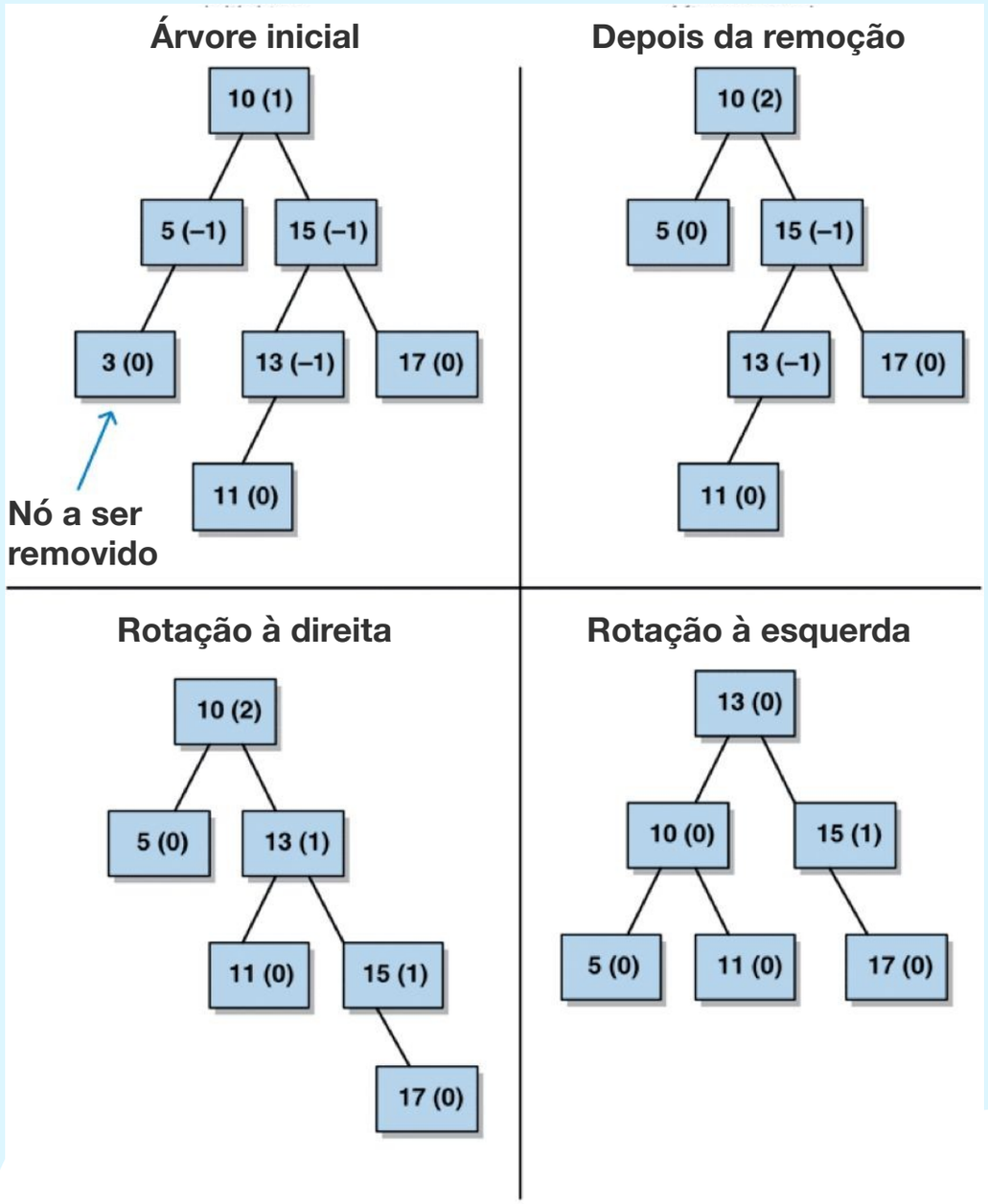
Rotação à Direita numa Árvore AVL



- Se o factor de balanceamento de um nó é +2, significa que a sub-árvore direita tem um caminho que é longo demais
- Então, se o factor de balanceamento do filho direito é +1, significa que o caminho longo é da sub-árvore direita do filho direito
- Neste caso, uma rotação simples à esquerda do filho direito em torno do nó original vai resolver o balanceamento

- Se o factor de balanceamento de um nó é +2, significa que a sub-árvore direita tem um caminho que é longo demais
- Então, se o factor de balanceamento do filho direito é -1, significa que o caminho longo é da sub-árvore esquerda do filho direito
- Neste caso, uma rotação Direita-Esquerda irá resolver o balanceamento

Rotação à Direita-Esquerda numa Árvore AVL



- Se o factor de balanceamento de um nó é -2, significa a sub-árvore esquerda tem um caminho que é longo demais
- Então, se o factor de balanceamento do filho esquerdo é +1, significa que o caminho longo é da sub-árvore direita do filho esquerdo
- Neste caso, uma rotação Esquerda-Direita irá resolver o balanceamento

Árvores de Pesquisa Binária na Plataforma de Colecções do *Java*

- A plataforma de colecções do *Java* fornece duas implementações para árvores binárias de pesquisa
  - `TreeSet`
  - `TreeMap`
- Para entender a diferença entre estas duas implementações, é preciso primeiro discutir a diferença entre um *set* e um *map*

Sets e Maps

- Na terminologia da plataforma de colecções do *Java*, todas as colecções que temos discutido até agora seriam consideradas sets (conjuntos)
- Um `set` é um grupo onde todos os dados associados a um objecto está armazenado na colecção
- Um `map` é uma colecção onde as chaves que fazem referência a um objecto são armazenadas na colecção, mas os demais dados serão armazenados separadamente

- Os `maps` são úteis porque permitem manipular as chaves dentro de uma colecção, em vez de todo o objecto
- Isso permite que as colecções sejam menores, mais eficientes e mais fáceis de gerir
- Permite também que para o mesmo objecto pode ser parte de várias colecções ao ter chaves em cada uma delas

Classes `TreeSet` e `TreeMap`

- Ambas as classes `TreeSet` e `TreeMap` são implementações de árvores binárias de pesquisa vermelhas e pretas