ESTRUTURAS DE DADOS

2024/2025

Aula 11

- *Heaps*
- Filas de Prioridade
- Implementar *Heaps* com listas ligadas
- Implementar *Heaps* com *array*
- Implementar filas de prioridade
- Eficiência

Ricardo Santos | rjs@estg.ipp.pt

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Acervos (*heaps*)

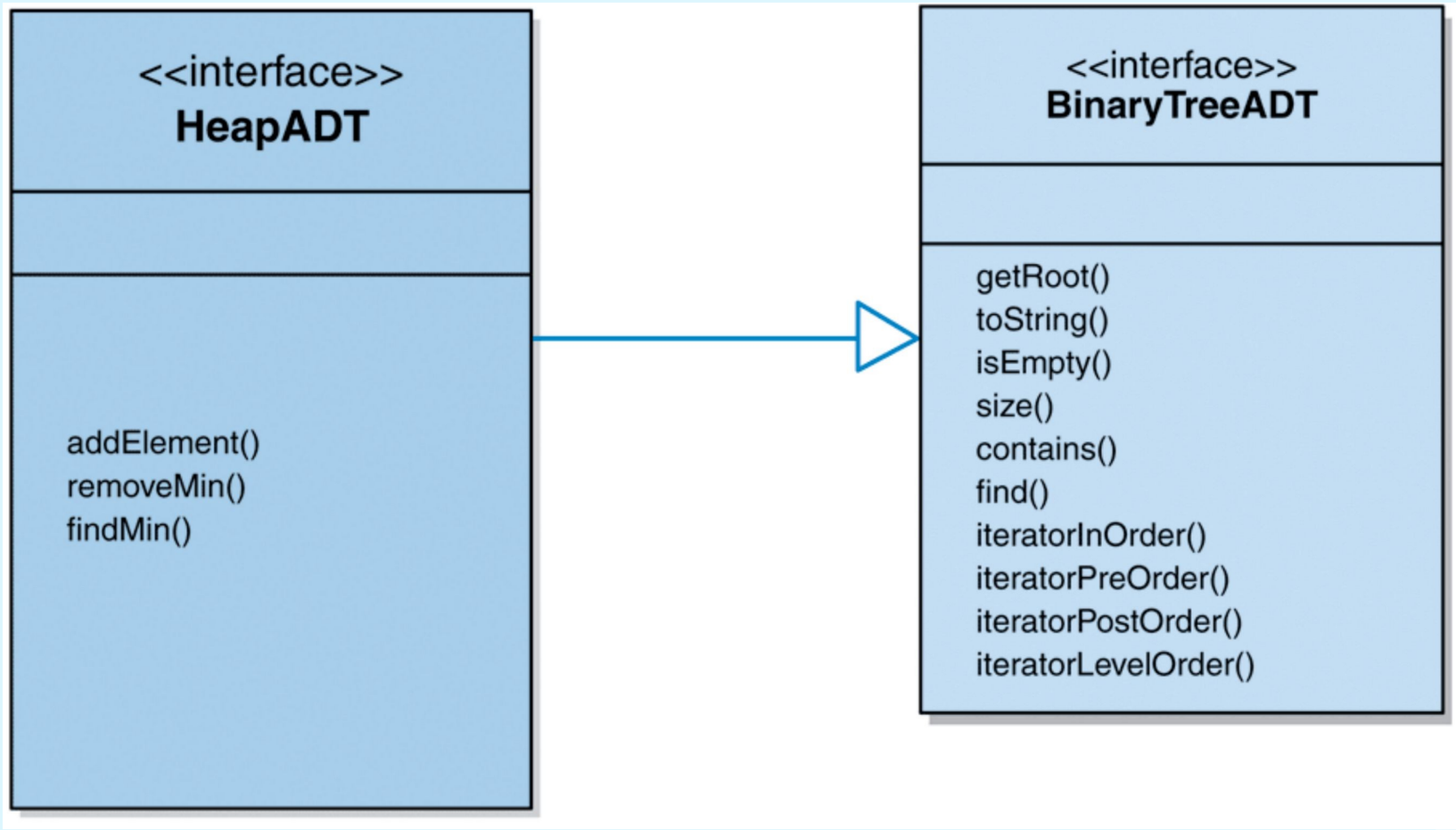
- Um acervo (*heap*) é uma árvore binária com duas propriedades
- No caso das *minheaps*:
  - É uma árvore completa
  - Para cada nó, o nó é menor ou igual ao filho esquerdo e ao filho direito
- Esta definição descreve uma *minheap*

- Além das operações herdadas de uma árvore binária, uma *heap* tem as seguintes operações adicionais:
  - `addElement`
  - `removeMin`
  - `findMin`

Operações de uma *heap*

Operação	Descrição
`addElement`	Adiciona um determinado elemento à heap
`removeMin`	Remove o menor elemento na heap
`findMin`	Retorna uma referência do menor elemento na heap

Interface `HeapADT`



Interface HeapADT

```
public interface HeapADT<T> extends BinaryTreeADT<T> {
   /**
    * Adds the specified object to this heap.
    *
    * @param obj  the element to added to this head
    */
   public void addElement (T obj);

   /**
    * Removes element with the lowest value from this heap.
    *
    * @return  the element with the lowest value from this heap
    */
   public T removeMin();

   /**
    * Returns a reference to the element with the lowest value in
    * this heap.
    *
    * @return  a reference to the element with the lowest value
    * in this heap
    */
   public T findMin();
}
```

Operação `addElement`

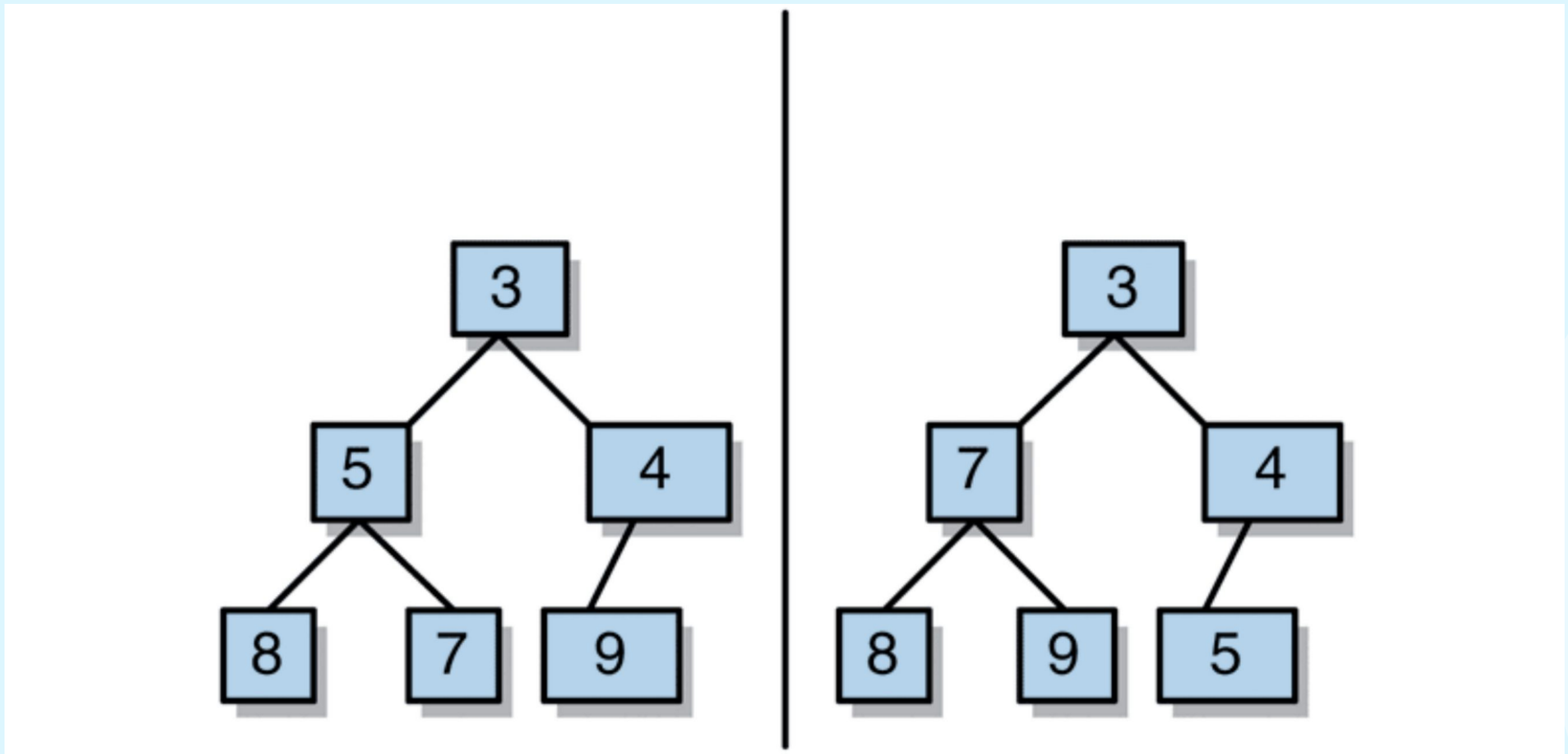
- O método `addElement` adiciona um determinado elemento para o local apropriado na *heap*
- O local adequado é o local que irá manter a integridade da árvore
- Existe apenas um local correcto para a inserção de um novo nó
  - Ou a próxima posição livre da esquerda no nível **h**
  - Ou a primeira posição no nível **h+1** se o nível **h** estiver cheio

- Depois de termos localizado a posição correcta para o novo nó temos de ter em conta a propriedade da ordenação

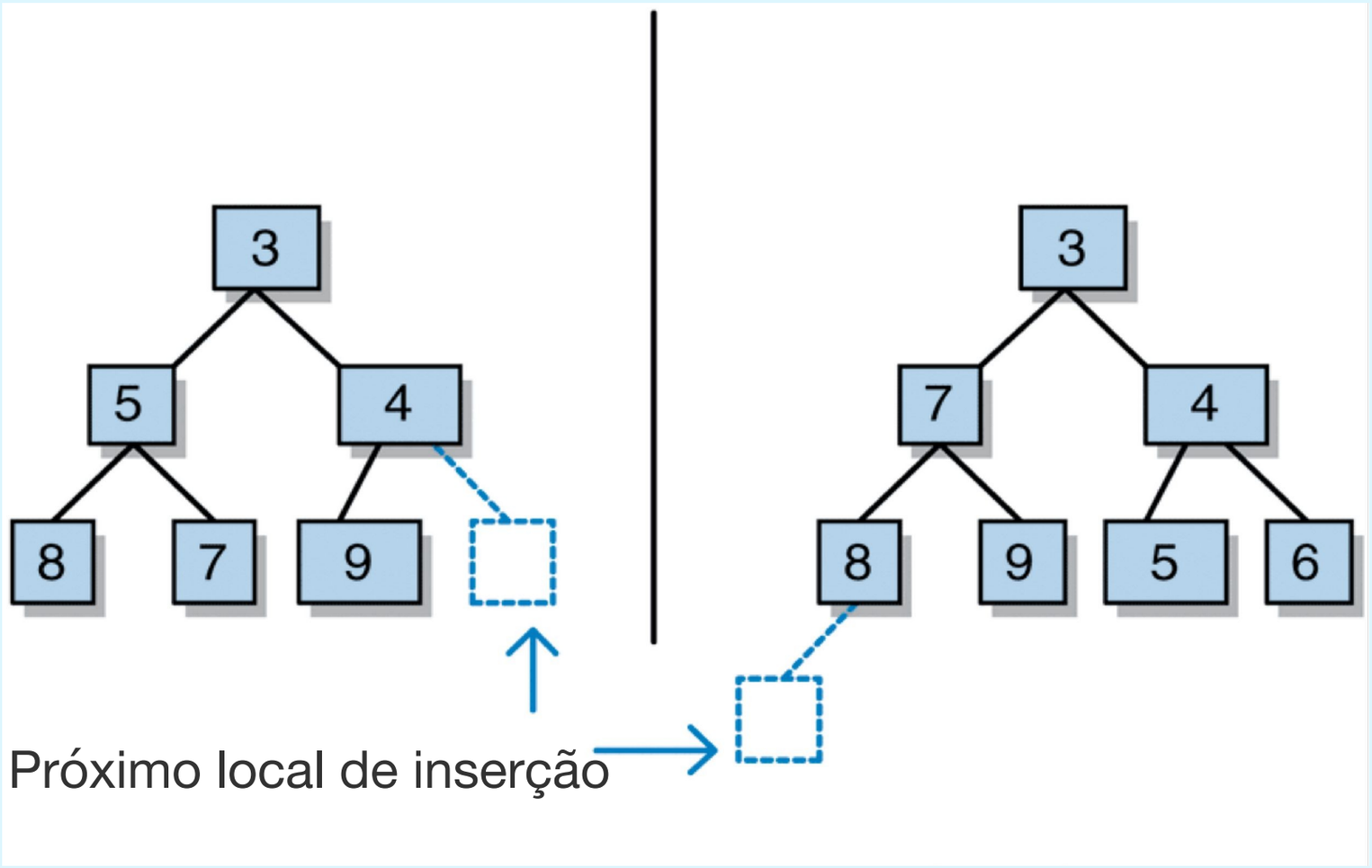
- Temos apenas de comparar o valor do novo nó com o valor do pai e realizar a troca de valores caso seja necessário

- Continuamos o processo subindo pela árvore até que o novo valor seja maior que o seu pai ou então o novo valor passa a ser a raiz da *heap*

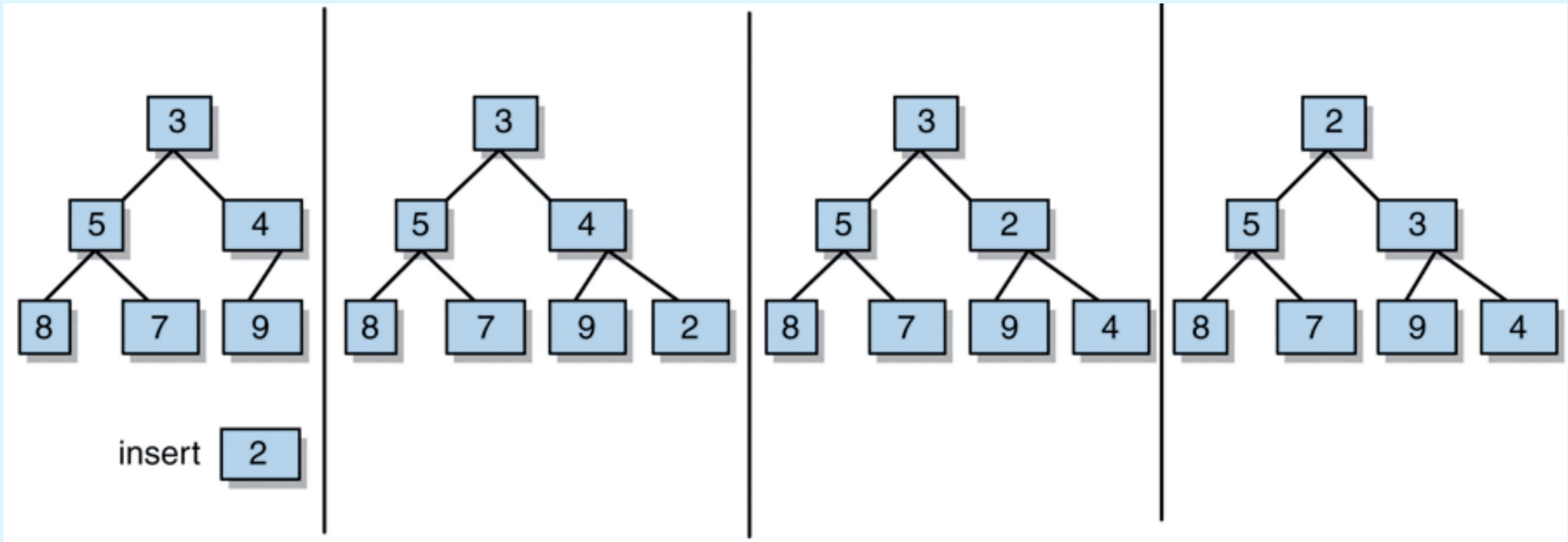
Duas *minheaps* com os mesmos dados



Pontos de inserção para a *heap*



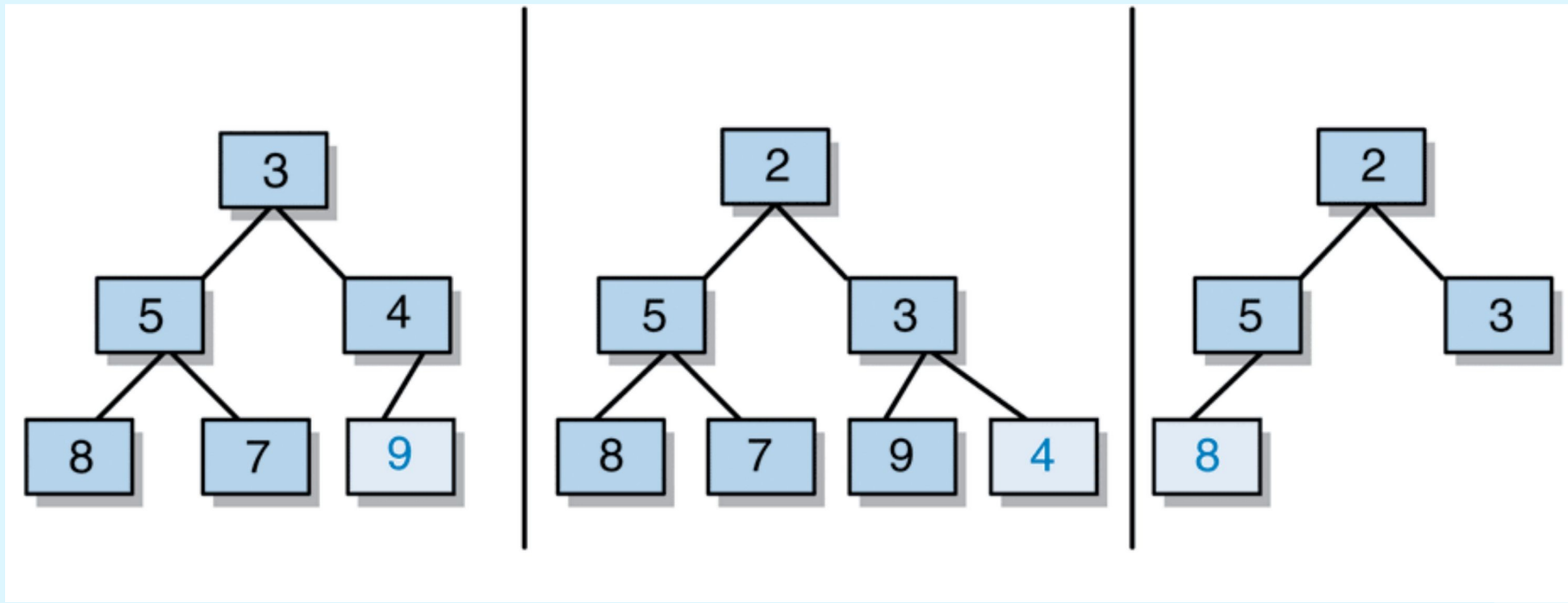
Inserção e reordenação na *heap*



Operação `removeMin`

- O método `removeMin` remove o elemento mínimo da *heap*
- O elemento mínimo é sempre armazenado na raiz
- Assim, temos de devolver o elemento raiz e substituí-lo por um outro elemento
- O elemento de substituição é sempre a última folha
- A última folha é sempre o último elemento no nível **h**

Exemplos da última folha na *heap*

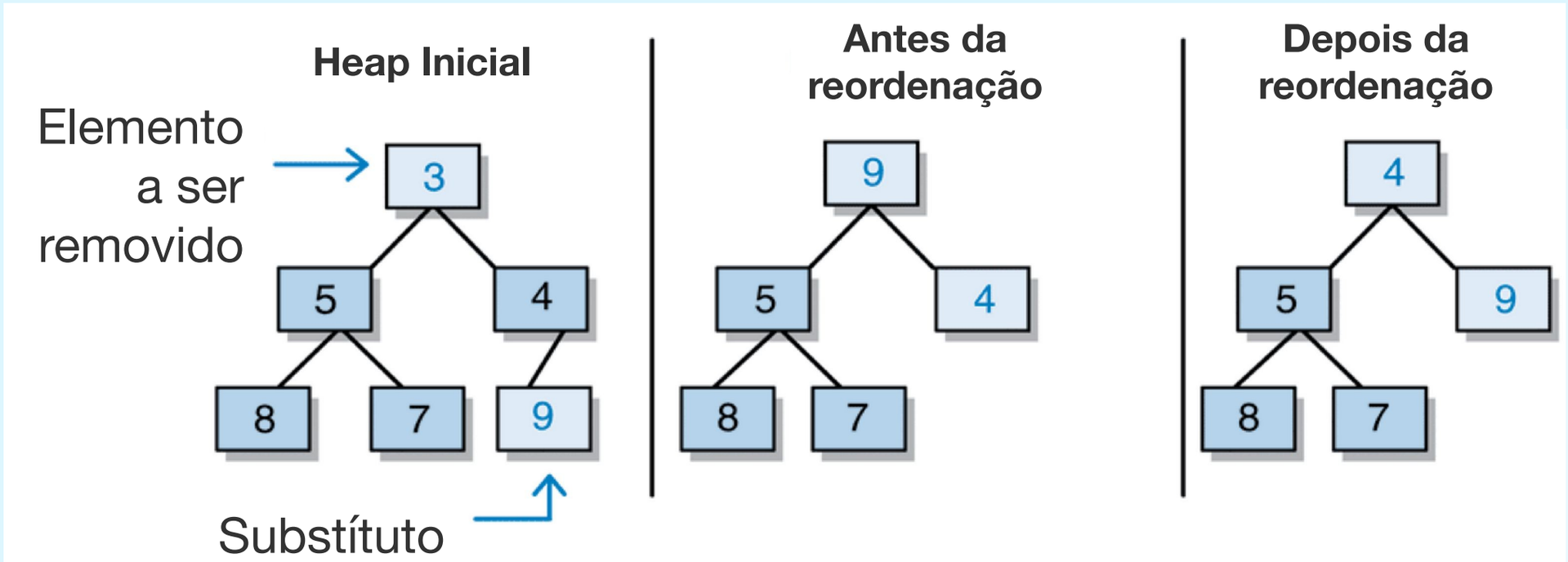


- Uma vez que o elemento armazenado na última folha tenha sido transferido para a raiz, a *heap* deve ser reordenada

- Isso é feito comparando o novo elemento de raiz com os seus filhos de valor menor (direito e esquerdo) e trocando-os com a raiz caso seja necessário

- Este processo é repetido pela árvore a baixo até que o elemento seja uma folha ou menor que ambos os seus filhos

Remoção e reordenamento numa *heap*



Usar *heaps*: Filas de Prioridade

- Uma fila de prioridade é uma colecção que segue duas regras de ordenação:
  - Os itens que têm maior prioridade saem primeiro
  - Itens com a mesma prioridade usam um método primeiro a entrar primeiro a sair (*first in first out*) para determinar sua ordenação
- Uma fila de prioridade pode ser implementada com recurso a uma lista de filas onde cada fila representa os itens de uma determinada prioridade

- Outra solução é usar uma *minheap*

- Ao ordenar a árvore por prioridade temos a primeira ordenação

- No entanto, a ordenação primeiro a entrar primeiro a sair (*first in first out*) tem de ser manipulada

- A solução é criar um objecto `PriorityQueueNode` que armazena o elemento a ser colocado na fila, a prioridade do elemento e a ordem de chegada do elemento

- Depois, temos apenas de definir o método `compareTo` para a classe `PriorityQueueNode` que compara primeiro a prioridade e só depois o tempo de chegada

- A classe `PriorityQueue` deverá estender a classe `Heap` e armazenar `PriorityQueueNodes`

Classe PriorityQueueNode

```
/**
 * PriorityQueueNode represents a node in a priority queue
 * containing a comparable object, order, and a priority value.
 *
 */

public class PriorityQueueNode<T> implements
                              Comparable<PriorityQueueNode> {
   private static int nextorder = 0;
   private int priority;
   private int order;
   private T element;
```

```
 /**
  * Creates a new PriorityQueueNode with the specified data.
  *
  * @param obj   the element of the new priority queue node
  * @param prio  the integer priority of the new queue node
  */
public PriorityQueueNode (T obj, int prio) {
     element = obj;
     priority = prio;
     order = nextorder;
     nextorder++;
}

/**
 * Returns the element in this node.
 *
 * @return  the element contained within this node
 */
public T getElement() {
     return element;
}
```

```
/**
 * Returns the priority value for this node.
 *
 * @return  the integer priority for this node
 */
public int getPriority() {
   return priority;
}

/**
 * Returns the order for this node.
 *
 * @return  the integer order for this node
 */
public int getOrder() {
   return order;
}

/**
 * Returns a string representation for this node.
 *
 */
public String toString() {
   String temp = (element.toString() + priority + order);
   return temp;
}
```

```
/**
 * Returns the 1 if the current node has higher priority than
 * the given node and -1 otherwise.
 *
 * @param obj  the node to compare to this node
 * @return     the integer result of the comparison of the obj
 *             node and this one
 */
public int compareTo(PriorityQueueNode obj)
{
   int result;
   PriorityQueueNode<T> temp = obj;
   if (priority > temp.getPriority())
      result = 1;
   else if (priority < temp.getPriority())
      result = -1;
   else if (order > temp.getOrder())
      result = 1;
   else
      result = -1;
   return result;
  }
}
```

Classe PriorityQueue

```
/**
 * PriorityQueue demonstrates a priority queue using a Heap.
 *
 */

public class PriorityQueue<T> extends
                    ArrayHeap<PriorityQueueNode<T>> {

   /**
    * Creates an empty priority queue.
    */
   public PriorityQueue() {
      super();
   }
```

```
/**
 * Adds the given element to this PriorityQueue.
 *
 * @param object  the element to be added to the priority queue
 * @param priority  the integer priority of the element to be added
 */
public void addElement (T object, int priority) {
   PriorityQueueNode<T> node =
              new PriorityQueueNode<T> (object, priority);
 super.addElement(node);
}
   /**
    * Removes the next highest priority element from this priority
    * queue and returns a reference to it.
    *
    * @return  a reference to the next highest priority element
    *          in this queue
    */
   public T removeNext() {
      PriorityQueueNode<T> temp =
                    (PriorityQueueNode<T>)super.removeMin();
      return temp.getElement();
   }
}
```

Implementar *Heaps* com Listas Ligadas

- A implementação *minheap* com recurso a uma lista ligada seria simplesmente uma extensão da nossa classe `LinkedBinaryTree`

- No entanto, uma vez que cada nó precisa ter uma referência para o pai, vamos criar uma classe `HeapNode` para estender à nossa classe `BinaryTreeNode` que usámos anteriormente

Classe HeapNode

```
public class HeapNode<T> extends BinaryTreeNode<T>{
   protected HeapNode<T> parent;

   /**
    * Creates a new heap node with the specified data.
    *
    * @param obj  the data to be contained within
    *             the new heap nodes
    */
   HeapNode (T obj) {
      super(obj);
      parent = null;
   }
}
```

Método `addElement`

- O método `addElement` deve realizar três tarefas:
  - Adicionar o novo nó no local apropriado
  - Reordenar a *heap*
  - Redefinir a referência `lastNode` para apontar para o novo ultimo nó

Classe `LinkedHeap`

```
/**
 * Heap implements a heap.
 *
 */

public class LinkedHeap<T> extends LinkedBinaryTree<T>
                         implements HeapADT<T> {

   public HeapNode<T> lastNode;

   public LinkedHeap() {
      super();
   }
```

```
/**
 * Adds the specified element to this heap in the
 * appropriate position according to its key value
 * Note that equal elements are added to the right.
 * @param obj  the element to be added to this head
 */
public void addElement (T obj) {
   HeapNode<T> node = new HeapNode<T>(obj);

   if (root == null)
      root=node;
   else {
      HeapNode<T> next_parent = getNextParentAdd();
      if (next_parent.left == null)
         next_parent.left = node;
      else
         next_parent.right = node;

      node.parent = next_parent;
   }
   lastNode = node;
   count++;
   if (count>1)
      heapifyAdd();
}
```

LinkedHeap: Operação `addElement`

- A operação `addElement` faz uso de dois métodos privados:
  - `getNextParentAdd` que retorna uma referência para o nó que será o pai do novo nó
  - `heapifyAdd` que reordena a *heap* após a inserção

```
/**
 * Returns the node that will be the parent of the new node
 *
 * @return  the node that will be a parent of the new node
 */
private HeapNode<T> getNextParentAdd() {
   HeapNode<T> result = lastNode;

   while ((result != root) &&
                  (result.parent.left != result))
      result = result.parent;

   if (result != root)
      if (result.parent.right == null)
         result = result.parent;
      else {
         result = (HeapNode<T>)result.parent.right;
         while (result.left != null)
            result = (HeapNode<T>)result.left;
      }
   else
      while (result.left != null)
         result = (HeapNode<T>)result.left;

   return result;
}
```

```
/**
 * Reorders this heap after adding a node.
 */
private void heapifyAdd() {
   T temp;
   HeapNode<T> next = lastNode;

   temp = next.element;

   while ((next != root) && (((Comparable)temp).compareTo
                              (next.parent.element) < 0))
   {
      next.element = next.parent.element;
      next = next.parent;
   }
   next.element = temp;
}
```

LinkedHeap: Operação `removeMin`

- A operação `removeMin` deve realizar três tarefas:
  - Substituir o elemento armazenado na raiz pelo elemento armazenado na última folha
  - Reordenar a *heap*, se necessário
  - Retornar o elemento raiz original

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it.
 * Throws an EmptyCollectionException if the heap is empty.
 *
 * @return    the element with the lowest value in this heap
 * @throws EmptyCollectionException  if an empty collection
 *                                   exception occurs
 */
public T removeMin() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException ("Empty Heap");

   T minElement =  root.element;

   if (count == 1)
   {
      root = null;
      lastNode = null;
   }
   else
   {
```

```
        HeapNode<T> next_last = getNewLastNode();
        if (lastNode.parent.left == lastNode)
            lastNode.parent.left = null;
        else
            lastNode.parent.right = null;

        root.element = lastNode.element;
        lastNode = next_last;
        heapifyRemove();
    }

    count--;

    return minElement;
}
```

- Como a operação `addElement`, a operação `removeMin` faz uso de dois métodos privados
  - `getNewLastNode` que retorna uma referência para o novo último nó na *heap*
  - `heapifyRemove` que reordena a *heap* após a remoção

```
/**
  * Returns the node that will be the new last node after
  * a remove.
  *
  * @return  the node that willbe the new last node after
  * a remove
  */
 private HeapNode<T> getNewLastNode(){

    HeapNode<T> result = lastNode;

    while ((result != root) && (result.parent.left == result))
       result = result.parent;

    if (result != root)
       result = (HeapNode<T>)result.parent.left;

    while (result.right != null)
       result = (HeapNode<T>)result.right;

    return result;
 }
```

```
/**
 * Reorders this heap after removing the root element.
 */
private void heapifyRemove(){
   T temp;
   HeapNode<T> node = (HeapNode<T>)root;
   HeapNode<T> left = (HeapNode<T>)node.left;
   HeapNode<T> right = (HeapNode<T>)node.right;
   HeapNode<T> next;

   if ((left == null) && (right == null))
      next = null;
   else if (left == null)
      next = right;
   else if (right == null)
      next = left;
   else if (((Comparable)left.element).compareTo(right.element) < 0)
      next = left;
   else
      next = right;
```

```
temp = node.element;
     while ((next != null) && (((Comparable)next.element).compareTo
                                (temp) < 0))
     {
        node.element = next.element;
        node = next;
        left = (HeapNode<T>)node.left;
        right = (HeapNode<T>)node.right;

        if ((left == null) && (right == null))
           next = null;
        else if (left == null)
           next = right;
        else if (right == null)
           next = left;
        else if (((Comparable)left.element).compareTo(right.element) < 0)
           next = left;
        else
           next = right;
     }
     node.element = temp;
  }
```

Implementar *Heaps* com *arrays*

- Uma implementação em *array* de uma *heap* pode-se tornar uma alternativa mais simples
- Numa implementação em *array*, a localização dos pais e filhos pode ser sempre calculada
- Dado que a raiz está na posição **0**, então para qualquer nó armazenado na posição **n** do *array*, o seu filho esquerdo está em posição de **2n + 1** e seu filho direito está na posição **2 (n +1)**
- Isso significa que o pai está na posição **(n-1) / 2**

Classe ArrayHeap

```
/**
 * ArrayHeap provides an array implementation of a minheap.
 *
 */

public class ArrayHeap<T> extends ArrayBinaryTree<T>
                          implements HeapADT<T>
{
   public ArrayHeap()
   {
      super();
   }
```

- Tal como a versão em lista ligada, a operação `addElement` para a implementação em *array* de uma heap deve realizar três tarefas:
  - Adicionar o novo nó,
  - Reordenar a *heap*,
  - Incrementar a contagem por um
- A versão `ArrayHeap` deste método exige apenas um método privado, `heapifyAdd` que reordena a *heap* após a inserção

```
/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value.
 * Note that equal elements are added to the right.
 *
 * @param obj  the element to be added to this heap
 */
public void addElement (T obj)
{

   if (count==tree.length)
      expandCapacity();

   tree[count] =obj;
   count++;

   if (count>1)
      heapifyAdd();
}
```

```
/**
 * Reorders this heap to maintain the ordering property after
 * adding a node.
 */
private void heapifyAdd()
{
   T temp;
   int next = count - 1;

   temp = tree[next];

   while ((next != 0) && (((Comparable)temp).compareTo
                          (tree[(next-1)/2]) < 0))
   {

      tree[next] = tree[(next-1)/2];
      next = (next-1)/2;
   }

   tree[next] = temp;
}
```

ArrayHeap: Operação `removeMin`

- A operação `removeMin` deve realizar três tarefas:
  - Substituir o elemento armazenado na raiz pelo elemento armazenado na última folha
  - Reordenar a *heap*, se necessário
  - Retornar o elemento da raiz original
- Como a operação `addElement`, a operação `removeMin` faz uso de um método privado, `heapifyRemove` para reordenar a *heap*

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it.
 * Throws an EmptyCollectionException if the heap is empty.
 *
 * @return     a reference to the element with the
 *             lowest value in this head
 * @throws EmptyCollectionException  if an empty collection
 *                                   exception occurs
 */
public T removeMin() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException ("Empty Heap");

   T minElement = tree[0];
   tree[0] = tree[count-1];
   heapifyRemove();
   count--;

   return minElement;
}
```

```
/**
 * Reorders this heap to maintain the ordering property.
 */
private void heapifyRemove() {
   T temp;
   int node = 0;
   int left = 1;
   int right = 2;
   int next;

   if ((tree[left] == null) && (tree[right] == null))
      next = count;
   else if (tree[left] == null)
      next = right;
   else if (tree[right] == null)
      next = left;
   else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
      next = left;
   else
      next = right;
   temp = tree[node];
```

```
while ((next < count) && (((Comparable)tree[next]).compareTo
                          (temp) < 0))
     {
        tree[node] = tree[next];
        node = next;
        left = 2*node+1;
        right = 2*(node+1);
        if ((tree[left] == null) && (tree[right] == null))
           next = count;
        else if (tree[left] == null)
           next = right;
        else if (tree[right] == null)
           next = left;
        else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
           next = left;
        else
           next = right;
     }
     tree[node] = temp;
  }
```

Eficiência das implementações de uma *heap*

- A operação `addElement` é **O (log n)**, para ambas as implementações
- A operação `removeMin` é **O (log n)**, para ambas as implementações
- O funcionamento `findMin` é **O (1)**, para ambas as implementações

Usar *heaps*: Ordenação da *Heap*

- Dada a propriedade da ordenação de uma *heap*, é natural que se pense em usar uma *heap* para ordenar uma lista de objectos
- Uma abordagem seria simplesmente adicionar todos os objectos numa *heap* e depois removê-los um de cada vez em ordem crescente

- A inserção numa *heap* é **O (log n)** para qualquer nó, portanto, será **O (n log n)** para construir uma *heap* com **n** nós
- No entanto, também é possível construir uma *heap* com recurso a um *array*
- Como sabemos que a posição relativa de cada um dos pais e respectivos filhos no *array*, temos simplesmente de começar com o primeiro nó não-folha do *array* e compará-lo com os seus filhos - fazendo trocas se necessário

- De seguida trabalhamos para trás no *array* até chegarmos à raiz
- Já que no máximo irá ser necessário fazer apenas duas comparações para cada nó não folha esta abordagem é **O(n)** para construir a *heap*