ESTRUTURAS DE DADOS

2023/2024

Aula 09

- Árvores
- Árvores Binárias
- Travessias
- Implementar Árvores Binárias

Ricardo Santos | rjs@estg.ipp.pt

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Árvores

- Uma árvore é uma estrutura não-linear definida pelo conceito de que cada nó da árvore, com excepção do primeiro nó ou nó raiz, tem exactamente um pai

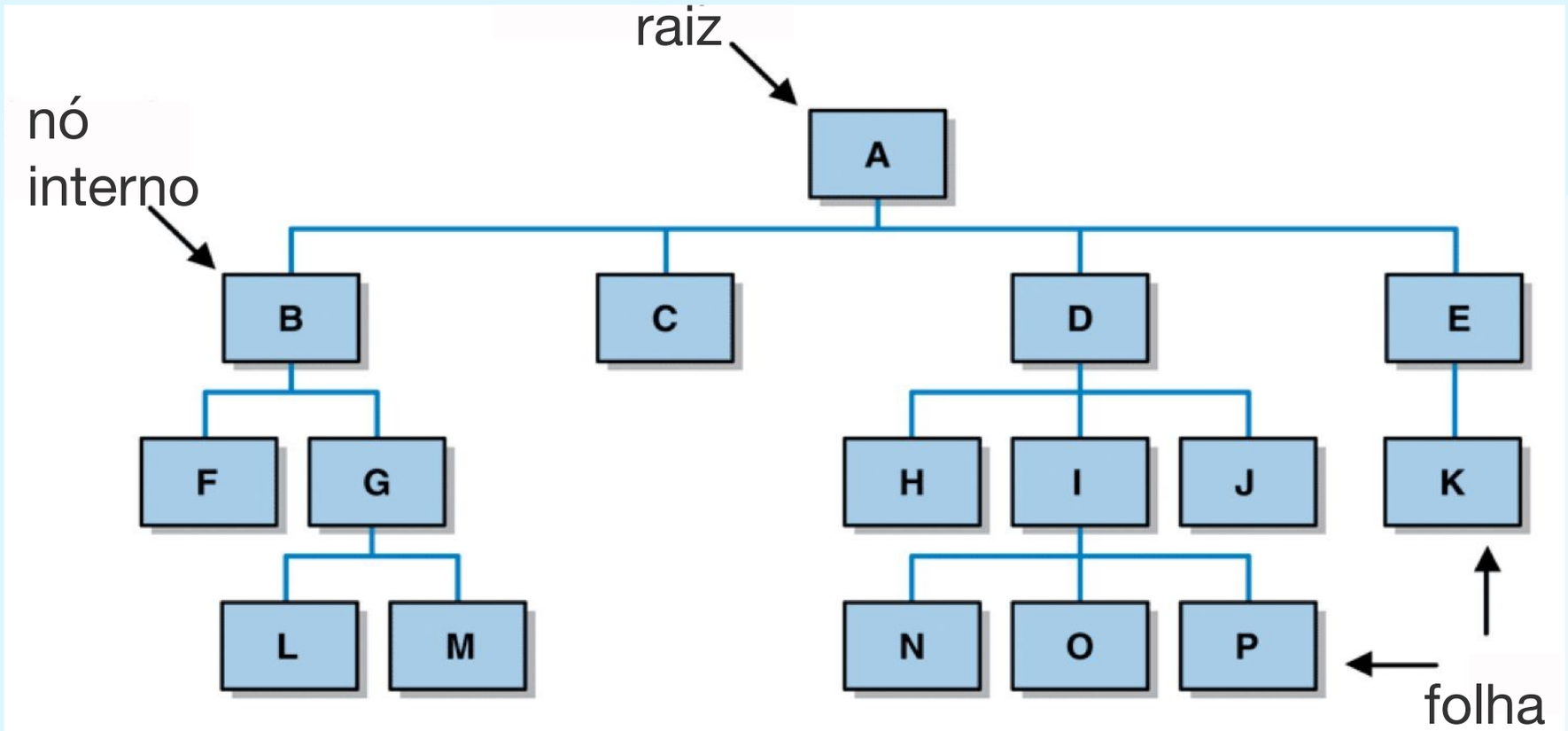
- No caso das árvores, as operações são dependentes do tipo de árvore e sua utilização

Definições

- Para discutir árvores, é necessário primeiro ter um vocabulário comum
- Já foram apresentados alguns termos:
  - **nó** - que se refere a um local na árvore onde um elemento está armazenado e
  - **raiz** - que se refere ao nó na base da árvore ou a um nó na árvore que não tem um pai

- Cada nó da árvore aponta para os nós que estão directamente abaixo
- Esses nós são referidos como seus **filhos**
- Por essa ordem de ideias um filho de um filho é chamado de **neto**, um filho de um neto é chamado de **bisneto**
- Um nó que não tenha pelo menos um filho é chamado de uma **folha**
- Um nó que não é a raiz e tem pelo menos um filho é chamado de **nó interno**

Terminologia



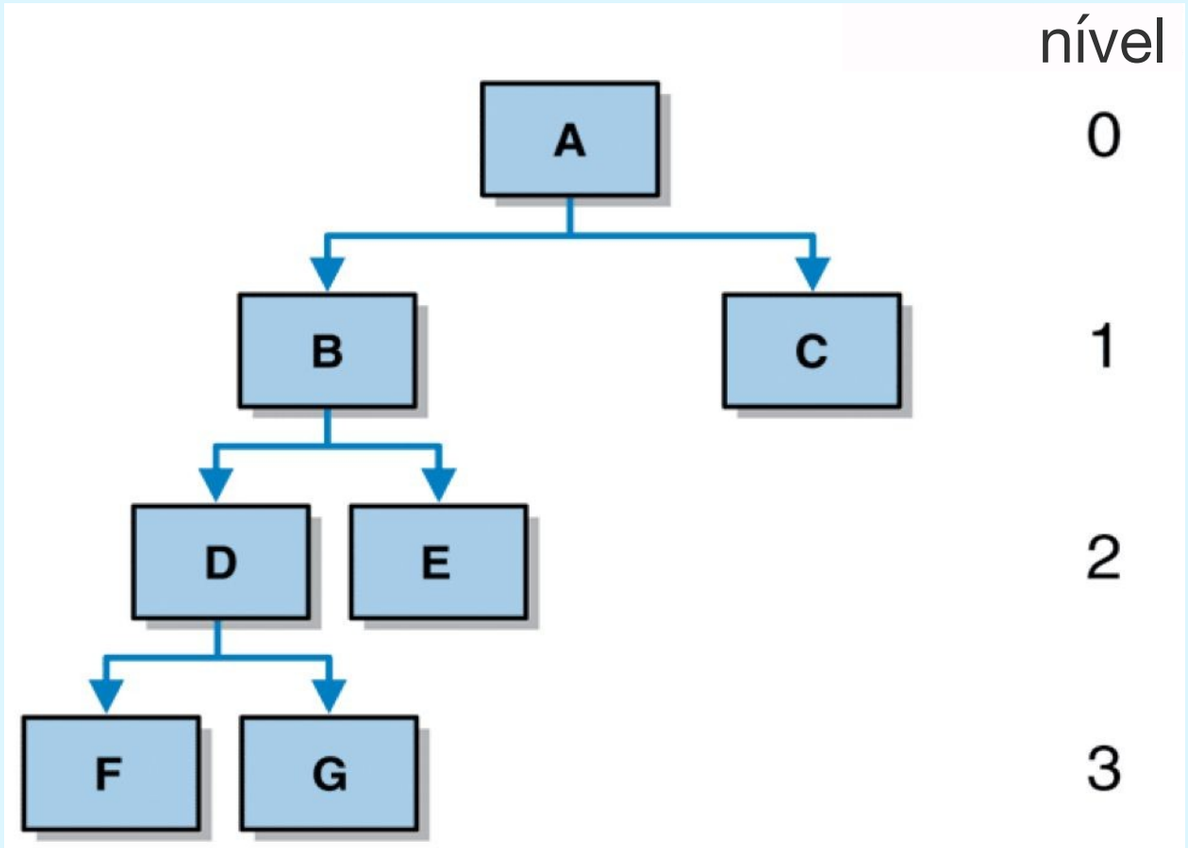
Definições

- Qualquer nó abaixo de outro nó e num caminho a partir desse nó é chamado de **descendente do nó**
- Qualquer nó acima de outro nó e num caminho de ligação a partir da raiz para esse nó é chamado **ancestral do nó**
- Todos os filhos do mesmo nó são chamados de **irmãos**
- Uma árvore que limita cada nó para não mais do que **n** filhos é chamada de **árvore n-ária**

- Cada nó da árvore está num nível específico ou a uma certa profundidade da árvore
- O nível de um nó é o comprimento do caminho desde a raiz até o nó
- Esta extensão de caminho é determinada pelo número de *links* que devem ser seguidos para obter o nó através da raiz
- A raiz é considerada como nível 0, os filhos da raiz estão no nível 1, os netos da raiz estão no nível 2, por assim adiante

- A altura ou a ordem de uma árvore é o comprimento do caminho mais longo desde a raiz até uma folha
- Assim, a altura ou a ordem da árvore no *slide* seguinte é 3
- O caminho da raiz **(A)** até à folha **(F)** é de comprimento 3
- O caminho da raiz **(A)** até à folha **(C)** é de comprimento 1

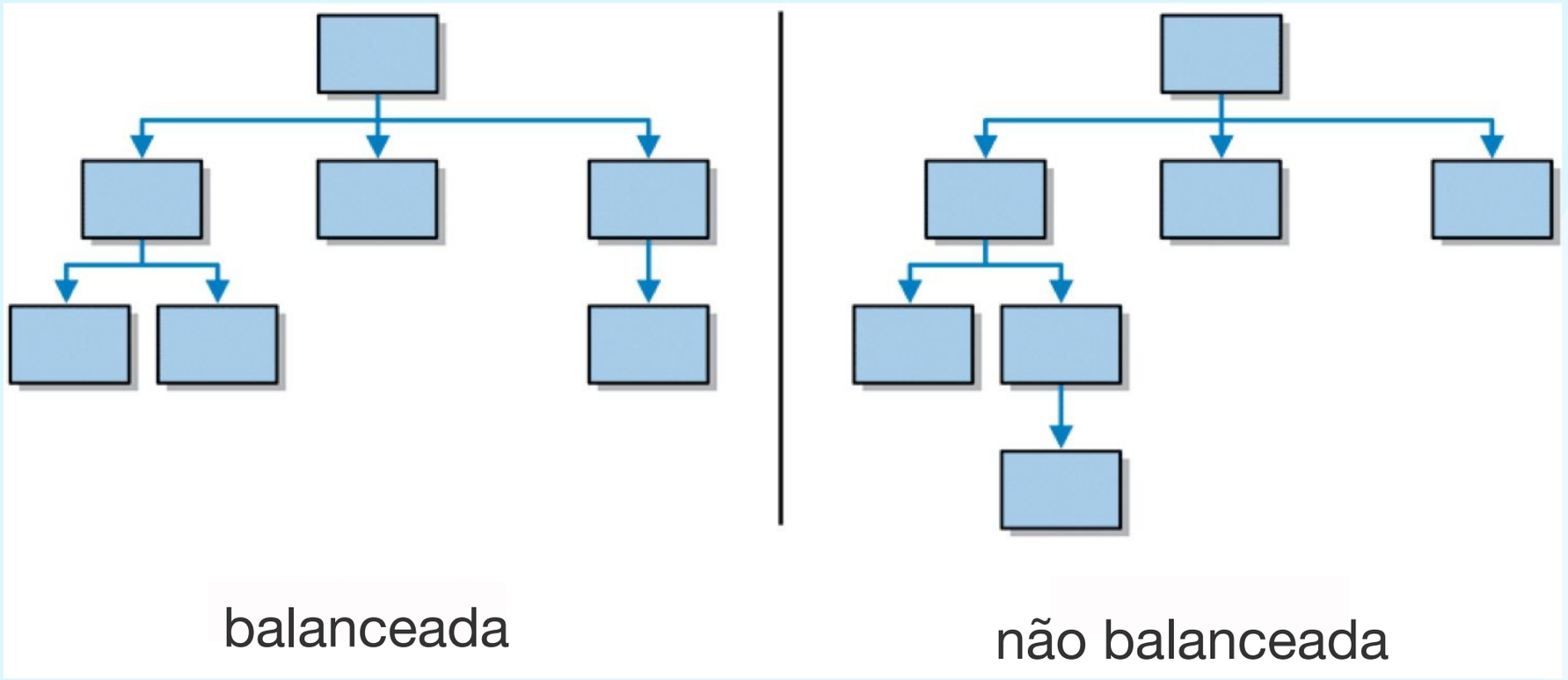
Comprimento e nível do caminho



Definições

- Uma árvore é considerada equilibrada quando todas as folhas da árvore estão aproximadamente na mesma profundidade
- Embora o uso do termo "aproximadamente" não seja intelectualmente satisfatório, a definição real é dependente do algoritmo usado
- Alguns algoritmos definem equilibrado quando todas as folhas estão no nível **h** ou **h-1**, onde **h** é a altura da árvore e onde **$\log_N n$ h** = é uma árvore n-ária

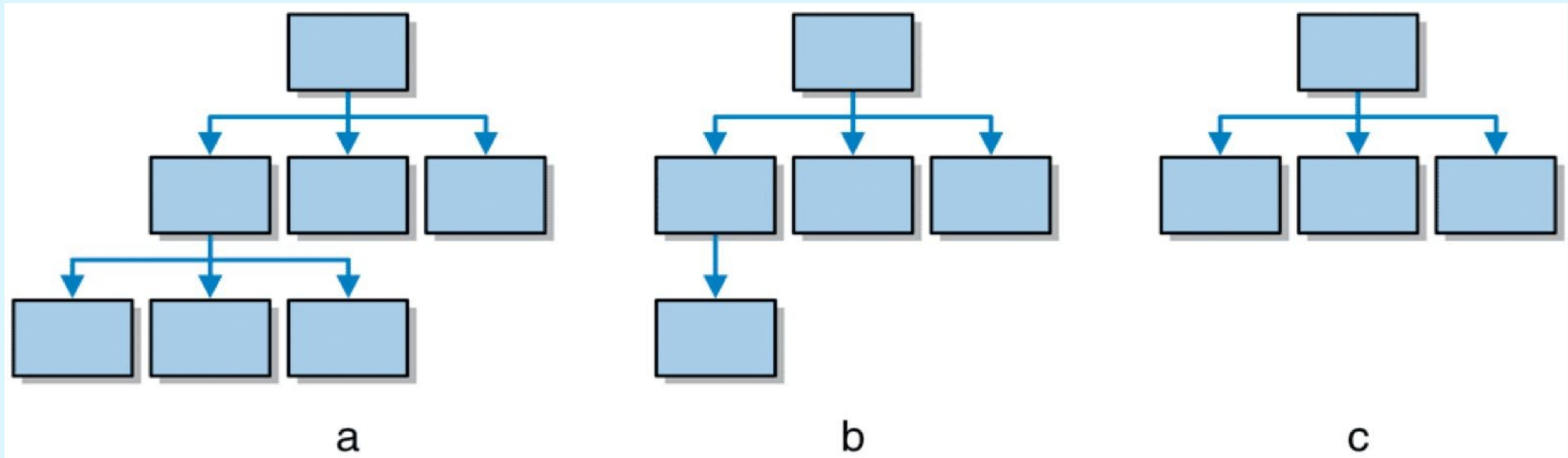
Árvores Balanceadas e Não Balanceadas



Definições

- O conceito de uma árvore completa está relacionado com o equilíbrio de uma árvore
- Uma árvore é considerada completa se estiver balanceada e se todas as folhas no nível **h** estiverem no lado esquerdo da árvore
- Esta definição tem implicações na forma como a árvore é armazenada em certas implementações

Alguns Exemplos de Árvores



Implementar Árvores com Listas Ligadas

- Embora não seja possível discutir os detalhes da implementação de uma árvore, sem definir o tipo de árvore e o seu uso, podemos olhar para as estratégias gerais de implementação de árvores

- A implementação mais óbvia da árvore é uma estrutura ligada

- Cada nó pode ser definido como uma classe `TreeNode`, como fizemos com a classe `LinearNode` para as listas ligadas

- Cada nó deve conter uma referência para o elemento a ser armazenado no nó, assim como as referências para cada um dos possíveis filhos desse nó

- Dependendo da aplicação, pode também ser útil armazenar uma referência em cada nó para o seu pai

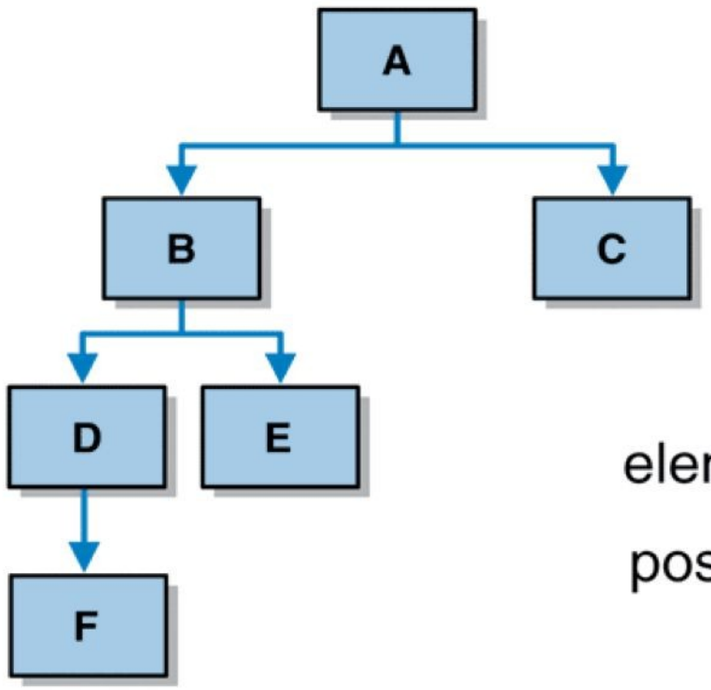
Implementar Árvores com *arrays*

- Para certos tipos de árvores, especialmente árvores binárias, pode ser usada uma estratégia computacional para armazenar uma árvore com recurso a um *array*

- Para qualquer elemento armazenado na posição **n** do *array*, o filho esquerdo será armazenado na posição **((2 * n) + 1)** e o filho direito será armazenado na posição **(2 * (n +1))**

- Esta estratégia pode ser gerida em termos de capacidade da mesma forma que fizemos para outras colecções baseadas em *array*
- No entanto, apesar da elegância conceptual desta solução existem inconvenientes
- Por exemplo, se a árvore em que estamos a armazenar não está completa ou relativamente completa, poderemos estar a desperdiçar uma grande quantidade de memória alocada no *array* para posições da árvore livres que não contêm dados

Implementar Árvores com *arrays*



element	A	B	C	D	E			F
position	0	1	2	3	4	5	6	7

- Uma segunda possibilidade para a implementação em *array* das árvores é modelada segundo a forma de gestão de memória dos Sistemas Operativos

- Em vez de atribuir os elementos da árvore para a posição do *array* por localização na árvore, as posições da *array* são alocadas de forma contígua em que o primeiro a chegar, é primeiro a ser servido

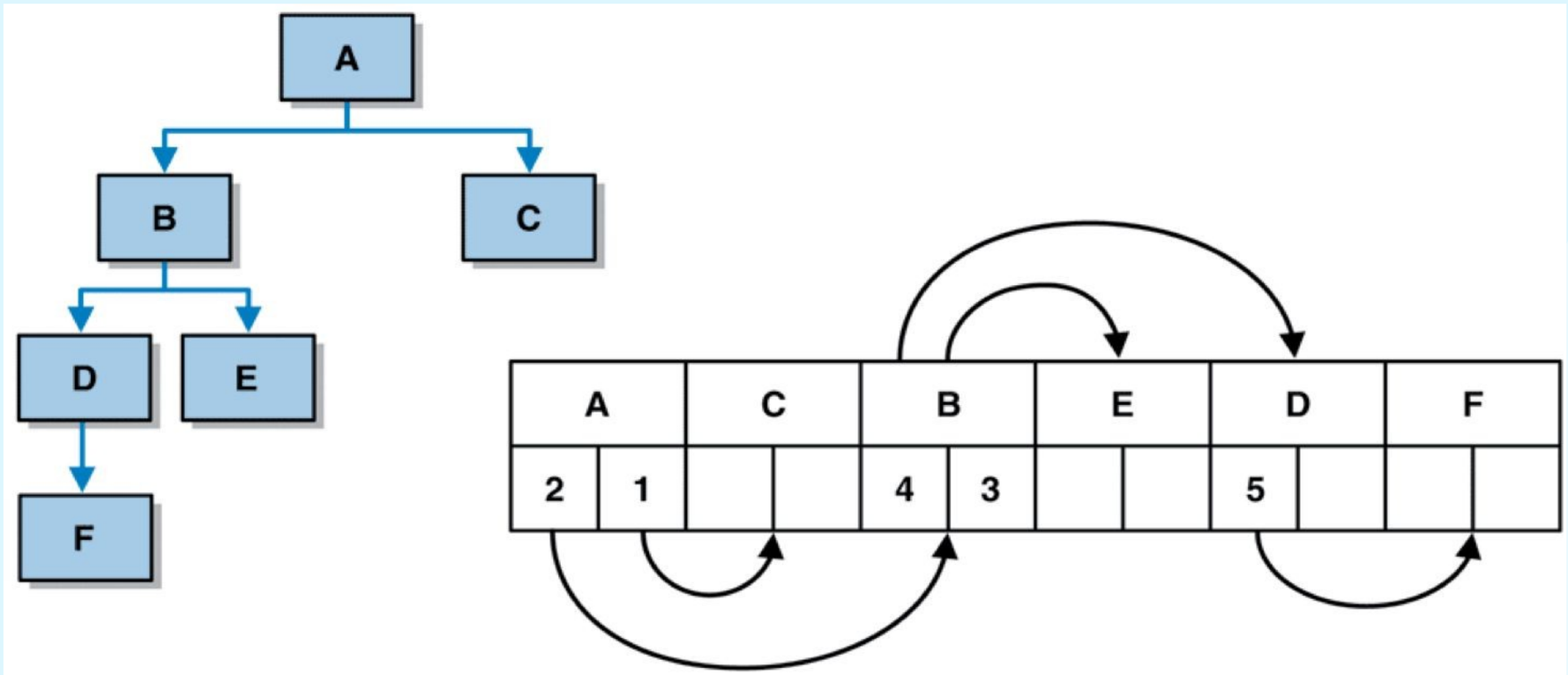
- Cada elemento do *array* será uma classe nó semelhante à classe `TreeNode` que analisámos anteriormente

- No entanto, em vez de armazenarmos as variáveis da referência do objecto para os seus filhos (e talvez do pai), seria armazenado em cada nó o índice do *array* de cada filho (e talvez do pai)

- Esta abordagem permite que os elementos sejam armazenados de forma contígua no *array* de modo que o espaço não seja desperdiçado

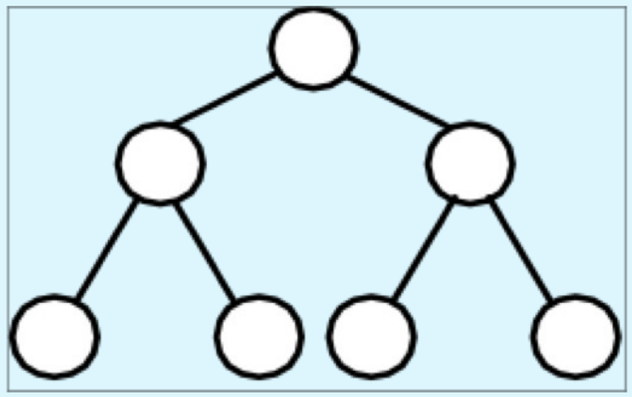
- No entanto, esta abordagem aumenta a sobrecarga (piora a performance) para excluir elementos na árvore uma vez que os restantes elementos terão de ser deslocados para manter a contiguidade ou então terá de ser gerida uma lista de posições livres

Ligações simuladas num *array* para a implementação de uma Árvore baseada em *array*

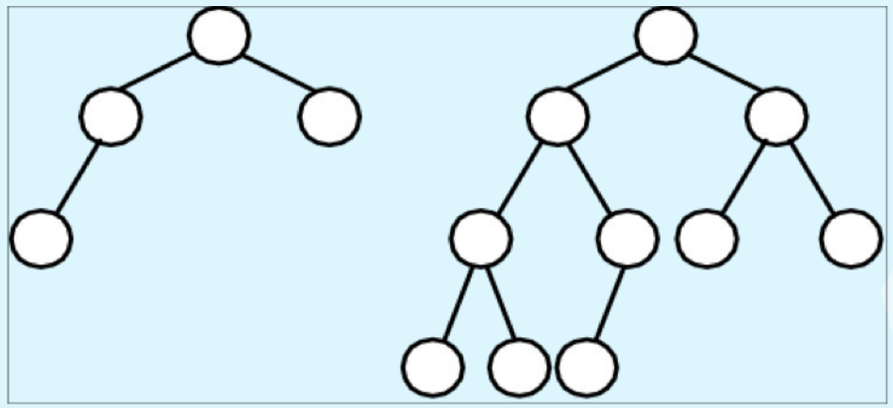


Árvores Binárias

- Uma árvore binária é uma árvore com uma aridade de 2 - um nó pode ter 0, 1, ou 2 filhos



Árvore binária cheia com altura 3



Árvore binária completa com altura 3 e 4

<u>Árvore binária cheia</u> com altura ***h*** é uma em que todos os nós no nível ***1*** ao nível ***h – 1*** têm dois filhos

<u>Árvore binária completa</u> com altura ***h*** é uma em que todos os nós do nível ***1*** ao ***h-2*** têm dois filhos e todos os filhos dos nós no nível ***h-1*** são contíguos e à esquerda da árvore

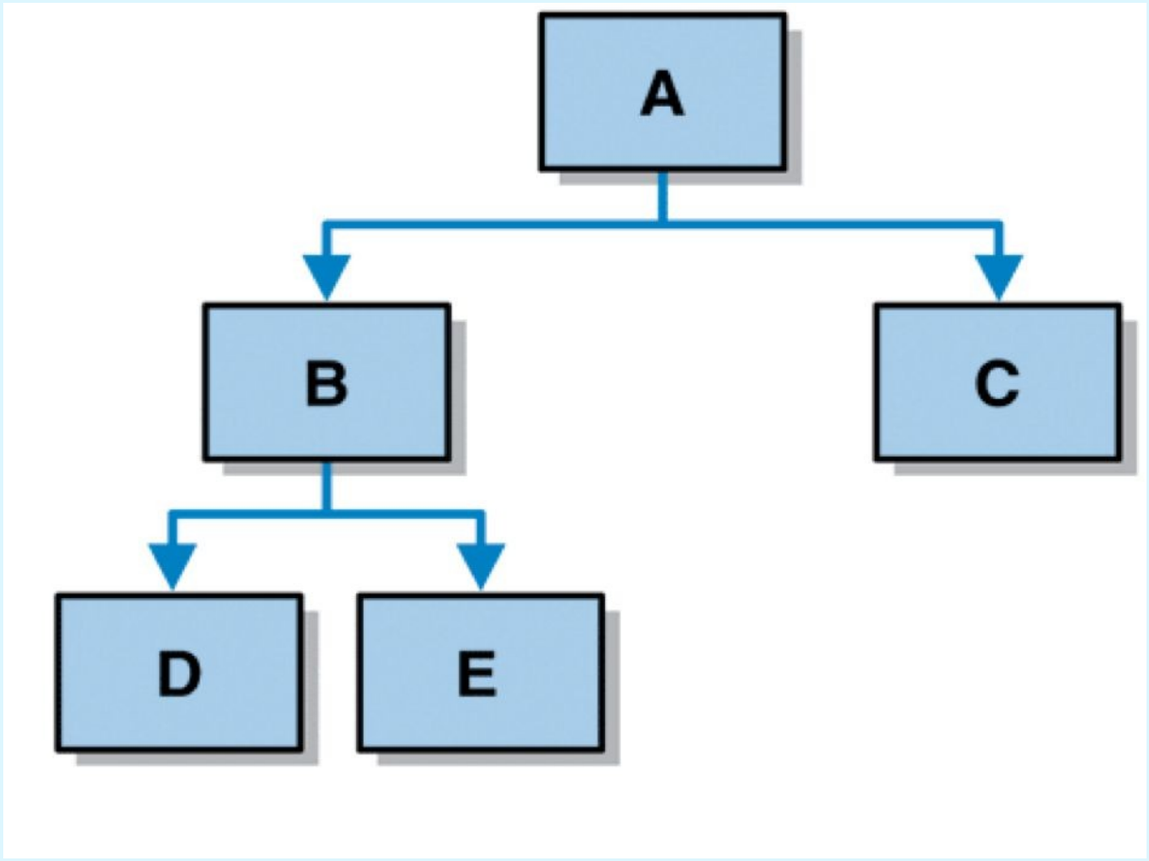
Árvores Binárias: Propriedades

- **Propriedade 1**
  - Uma árvore binária cheia de altura **h** tem $2^h - 1$ nós
- **Propriedade 2**
  - A altura da árvore binária cheia com **n** nós é $\log_2 n$. Este será também o tamanho do caminho mais longo de um árvore binária cheia
- **Propriedade 3**
  - A altura da árvore binária com **n** nós é no mínimo $\log_2 n$ e no máximo **n** (isto ocorre quando nenhum dos nós tem mais de um filho). Será também o tamanho do maior caminho

Implementar Árvores com *arrays*

- Existem quatro algoritmos básicos para a travessia de uma árvore:
  - pré-ordem (*preorder*)
  - em-ordem (*inorder*)
  - pós-ordem (*postorder*)
  - nível-ordem (*levelorder*)

Árvore Completa



Travessia Pré-Ordem

- Dito em pseudocódigo, o algoritmo para uma travessia pré-ordem de uma árvore binária é:

```
Visit node

Traverse(left child)

Traverse(right child)
```

Travessia Pré-Ordem

- A travessia pré-ordem é realizada, visitando cada nó, começando pela a raiz e de seguida pelos seus filhos

- Dada a árvore binária completa do próximo slide, um percurso pré-ordem seria:

```
A B D E C
```

Travessia Em-Ordem

- Dito em pseudocódigo, o algoritmo para uma travessia em-ordem de uma árvore binária é:

```
Traverse(left child)

Visit node

Traverse(right child)
```

Travessia Em-Ordem

- A travessia em-ordem é realizada ao visitar o filho esquerdo do nó, o nó, então todos os nós do filho restantes a partir da raiz
- Um percurso em em-ordem da árvore anterior produz a ordem:

```
D  B  E  A  C
```

Travessia Pós-Ordem

- Dito em pseudocódigo, o algoritmo para uma travessia pós-ordem de uma árvore binária é:

```
Traverse(left child)

Traverse(right child)

Visit node
```

Travessia Pós-Ordem

- A travessia pós-ordem é realizado ao visitar os filhos, e depois o nó a partir da raiz
- Dada a mesma árvore, um percurso pós-ordem produz a seguinte ordem:

D E B C A

Travessia Nível-Ordem

- Dito em pseudocódigo, o algoritmo para uma travessia nível-ordem de uma árvore binária é:

```
Create a queue called nodes
Create an unordered list called results
Enqueue the root onto the nodes queue
While the nodes queue is not empty {
    Dequeue the first element from the queue
    If that element is not null
        Add that element to the rear of the results list
        Enqueue the children of the element on the nodes queue
    Else
        Add null on the result list
}
Return an iterator for the result list
```

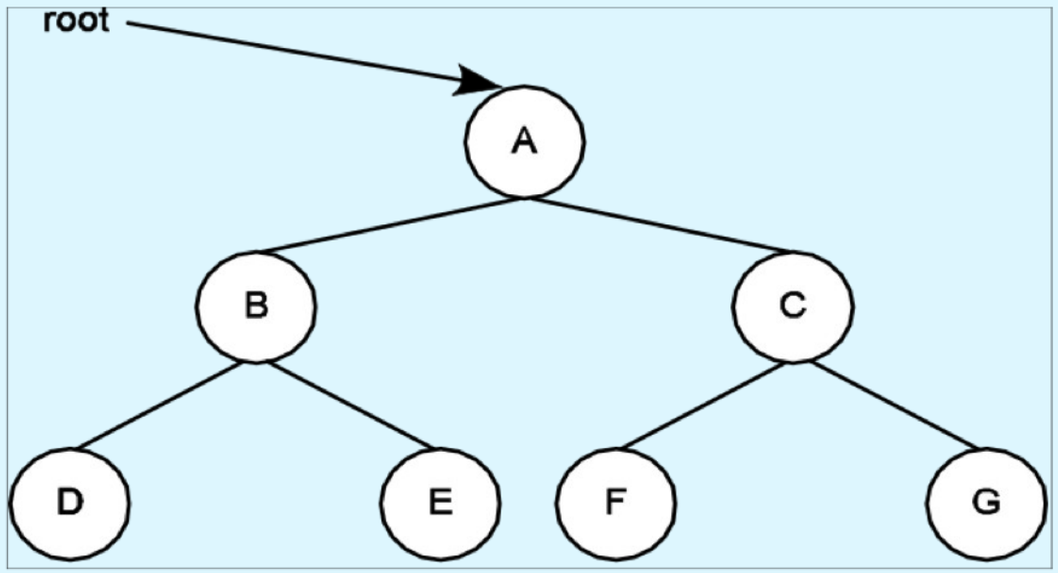
Travessia Nível-Ordem

- A travessia nível-ordem é realizada, visitando todos os nós em cada nível, um nível de cada vez, começando pela raiz
- Dada a mesma árvore, uma passagem nível-ordem produz a ordem:

```
A B C D E
```

Resumo das Travessias

- Considere a seguinte árvore binária



- Para as várias travessias analisadas: a ordem e que os nós são visitados é a seguinte:

Ordem da Travessia	Ordem em que os nós são visitados
`preorder( node )` *if node is not null* *visit this node* *preorder(node's left child)* *preorder(node's right child)*	**`preorder(root)`** visita os nós na seguinte ordem: A B D E C F G
`inorder( node )` *if node is not null* *inorder(node's left child)* *visit this node* *inorder(node's right child)*	**`inorder(root)`** visita os nós na seguinte ordem: D B E A F C G
`postorder( node )` *if node is not null* *postorder(node's left child)* *postorder(node's right child)* *visit this node*	**`postorder(root)`** visita os nós na seguinte ordem: D E B F G C A
levelorder(node) *if node is not null* *add node to a queue* *while the queue is not empty* *get the node at the head of the queue* *visit this node* *if the node has children* *put them in the queue in left to right order*	**`levelorder(root)`** visita os nós na seguinte ordem: A B C D E F G

Nota: os algoritmos são recursivos!

Implementar Árvores Binárias

- Como exemplo de possíveis implementações das árvores, vamos explorar a implementação de uma árvore binária

- Tendo especificado que estamos a implementar uma árvore binária, podemos identificar um conjunto de operações possíveis que seriam comuns para todas as árvores binárias

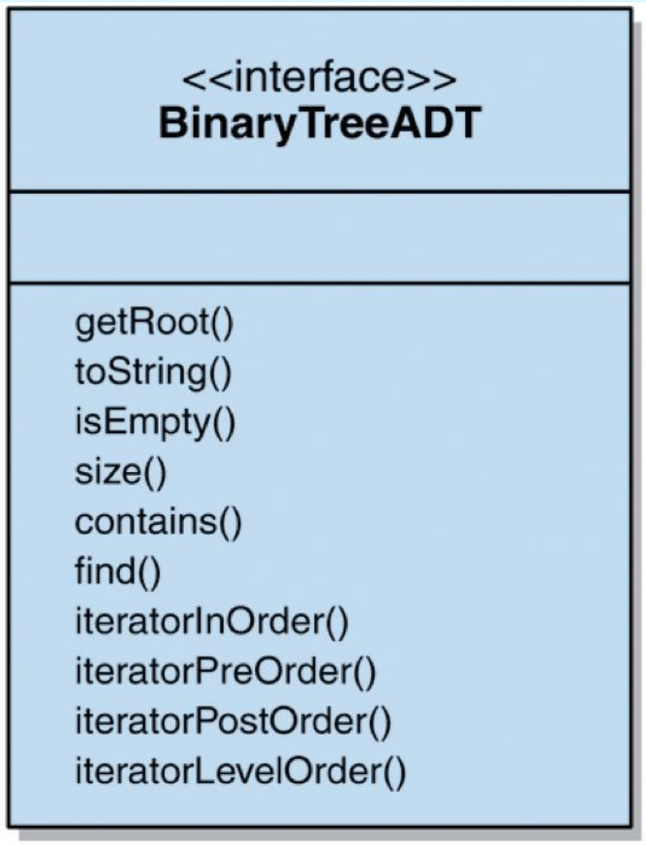
- Note porém, que para além de construtores, nenhuma destas operações adicionam elementos à árvore

- **Não é possível definir uma operação para adicionar um elemento à árvore até sabermos mais sobre como a árvore irá ser usada**

Operações de uma Árvore Binária

Operação	Descrição
`getRoot`	Retorna uma referência à raiz da árvore binária
`isEmpty`	Determina se a árvore está vazia
`size`	Retorna o número de elementos na árvore
`contains`	Determina se o alvo específico está na árvore
`find`	Retorna uma referência ao elemento de destino especificado se for encontrado
`toString`	Retorna uma representação de string da árvore
`iteratorInOrder`	Retorna um iterador para uma travessia do inroder da árvore
`iteratorPreOrder`	Retorna um iterador para uma travessia pré-roder da árvore
`iteratorPostOrder`	Retorna um iterador para uma travessia postroder da árvore
`iteratorLevelOrder`	Retorna um iterador para uma travessia de levelroder da árvore

Interface `BinaryTreeADT`



```
public interface BinaryTreeADT<T>
{
   /**
    * Returns a reference to the root element
    *
    * @return                  a reference to the root
    */
   public T getRoot ();

   /**
    * Returns true if this binary tree is empty and false otherwise.
    *
    * @return  true if this binary tree is empty
    */
  public boolean isEmpty();

  /**
   * Returns the number of elements in this binary tree.
   *
   * @return  the integer number of elements in this tree
  */
  public int size();
```

```
/**
 * Returns true if the binary tree contains an element that
 * matches the specified element and false otherwise.
 *
 * @param targetElement  the element being sought in the tree
 * @return               true if the tree contains the target element
 */
public boolean contains (T targetElement);

/**
 * Returns a reference to the specified element if it is found in
 * this binary tree.  Throws an exception if the specified element
 * is not found.
 *
 * @param targetElement  the element being sought in the tree
 * @return               a reference to the specified element
 */
public T find (T targetElement);

/**
 * Returns the string representation of the binary tree.
 *
 * @return  a string representation of the binary tree
 */
public String toString();
```

```
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with the root.
 *
 * @return  an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorInOrder();
```

```
/**
 * Performs a preorder traversal on this binary tree by calling an
 * overloaded, recursive preorder method that starts
 * with the root.
 *
 * @return  an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPreOrder();
```

```
    /**
     * Performs a postorder traversal on this binary tree by
     * calling an overloaded, recursive postorder
     * method that starts with the root.
     *
     * @return  an iterator over the elements of this binary tree
     */
    public Iterator<T> iteratorPostOrder();


    /**
     * Performs a levelorder traversal on the binary tree,
     * using a queue.
     *
     * @return  an iterator over the elements of this binary tree
     */
    public Iterator<T> iteratorLevelOrder();
}
```

Implementar uma Árvore Binária com Listas Ligadas

- Vamos examinar uma implementação em lista ligada de uma árvore binária
- A nossa implementação terá de acompanhar o nó que é a raiz da árvore, bem como a contagem de elementos na árvore

```
protected int count;

protected BinaryTreeNode<T> root;
```

Classe `LinkedBinaryTree`

- Será necessário fornecer dois construtores
  - Um para criar uma árvore a null
  - Um para criar uma árvore com um único elemento

Classe `LinkedBinaryTree`

```
/**
* LinkedBinaryTree implements the BinaryTreeADT interface
*
*/

public class LinkedBinaryTree<T> implements BinaryTreeADT<T>
{
   protected int count;
   protected BinaryTreeNode<T> root;
```

```
/**
 * Creates an empty binary tree.
 */
public LinkedBinaryTree()
{
   count = 0;
   root = null;
}

/**
 * Creates a binary tree with the specified element as its root.
 *
 * @param element  the element that will become the root of the
 * new binary tree
 */
public LinkedBinaryTree (T element)
{
   count = 1;
   root = new BinaryTreeNode<T> (element);
}
```

Classe `BinaryTreeNode`

- Será também necessária uma classe para representar cada nó na árvore
- Uma vez que esta é uma árvore binária, vamos criar uma classe `BinaryTreeNode` que contêm uma referência ao elemento armazenado no nó, assim como referências para cada um dos seus filhos

```
/**
 * BinaryTreeNode represents a node in a binary tree with a left and
 * right child.
 */

public class BinaryTreeNode<T> {

   protected T element;
   protected BinaryTreeNode<T> left, right;

   /**
    * Creates a new tree node with the specified data.
    *
    * @param obj  the element that will become a part of
    * the new tree node
   */
   BinaryTreeNode (T obj) {
      element = obj;
      left = null;
      right = null;
   }
```

```
/**
 * Returns the number of non-null children of this node.
 * This method may be able to be written more efficiently.
 *
 * @return  the integer number of non-null children of this node
 */
public int numChildren() {

   int children = 0;

   if (left != null)
      children = 1 + left.numChildren();

   if (right != null)
      children = children + 1 + right.numChildren();

   return children;
  }
}
```

Métodos `find` e `findagain`

- O método `find` dá-nos um excelente exemplo da recursividade que é possível dada a natureza de uma árvore
- Usamos o método `findagain` privado para suportar o método público `find`
- Desta forma podemos fazer uma maior distinção entre a invocação do método original e as chamadas recursivas subsequentes

```
/**
  * Returns a reference to the specified target element if it is
  * found in this binary tree.  Throws a NoSuchElementException if
  * the specified target element is not found in the binary tree.
  *
  * @param targetElement  the element being sought in this tree
  * @return       a reference to the specified target
  * @throws ElementNotFoundException  if an element not found
  *                                   exception occurs
  */

public T find(T targetElement) throws ElementNotFoundException
{
   BinaryTreeNode<T> current = findAgain( targetElement, root );

   if( current == null )
      throw new ElementNotFoundException("binary tree");

   return (current.element);
}
```

```
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree.
 *
 * @param targetElement  the element being sought in this tree
 * @param next           the element to begin searching from
 */
private BinaryTreeNode<T> findAgain(T targetElement,
                                    BinaryTreeNode<T> next) {

   if (next == null)
      return null;

   if (next.element.equals(targetElement))
      return next;

   BinaryTreeNode<T> temp = findAgain(targetElement, next.left);

   if (temp == null)
      temp = findAgain(targetElement, next.right);

   return temp;
}
```

Método `iteratorInOrder`

- Tal como o método `find`, o método `iteratorInOrder` usa um método privado, para a travessia em-ordem, para suportar a recursividade

- As travessias para uma árvore podem ser implementadas como os métodos `toString` ou iteradores ou ambos

```
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with
 * the root.
 *
 * @return  an in order iterator over this binary tree
 */

public Iterator<T> iteratorInOrder() {

   ArrayUnorderedList<T> tempList = new ArrayUnorderedList<T>();
   inorder (root, tempList);

   return tempList.iterator();
}
```

```
/**
 * Performs a recursive inorder traversal.
 *
 * @param node       the node to be used as the root
 *                   for this traversal
 * @param tempList   the temporary list for use in this traversal
 */

protected void inorder (BinaryTreeNode<T> node,
                        ArrayUnorderedList<T> tempList) {
   if (node != null) {

      inorder (node.left, tempList);
      tempList.addToRear(node.element);
      inorder (node.right, tempList);
   }
}
```

Implementação em *array* de uma Árvore Binária

- Vamos examinar a implementação em *array* de uma árvore binária
- A nossa implementação precisa manter o controlo do *array* que contém a árvore, assim como a contagem dos elementos na árvore

```
protected int count;

protected T[ ] tree;
```

A classe `ArrayBinaryTree`

- Será necessário fornecer dois construtores
  - Um para criar uma árvore nula
  - Um para criar uma árvore com apenas um único elemento

```
public class ArrayBinaryTree<T> implements BinaryTreeADT<T> {
   protected int count;
   protected T[] tree;
   private final int CAPACITY = 50;

   /**
    * Creates an empty binary tree.
    */
   public ArrayBinaryTree() {
      count = 0;
      tree = (T[]) new Object[capacity];
   }

   /**
    * Creates a binary tree with the specified element as its root.
    *
    * @param element  the element which will become the root
    * of the new tree
    */
   public ArrayBinaryTree (T element) {
      count = 1;
      tree = (T[]) new Object[capacity];
      tree[0] = element;
   }
```

```
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree.  Throws a NoSuchElementException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement   the element being sought in the tree
 * @return                true if the element is in the tree
 * @throws ElementNotFoundException  if an element not found
 *                                   exception occurs
 */
public T find (T targetElement) throws ElementNotFoundException {
   T temp=null;
   boolean found = false;

   for (int ct=0; ct<count && !found; ct++)
      if (targetElement.equals(tree[ct])) {
       found = true;
         temp = tree[ct];
      }

   if (!found)
      throw new ElementNotFoundException("binary tree");

   return temp;
}
```

Método `iteratorInOrder`

- Como para a implementação da árvore com listas ligadas o método `iteratorInOrder` usa um método privado, `inorder`, para suportar a recursividade

- As travessias para uma árvore podem ser implementadas como os métodos `toString` ou iteradores

```
/**
 * Performs an inorder traversal on this binary tree by
 * calling an overloaded, recursive inorder method
 * that starts with the root.
 *
 * @return  an iterator over the binary tree
 */
```

```
public Iterator<T> iteratorInOrder() {
   ArrayUnorderedList<T> templist = new ArrayUnorderedList<T>();
   inorder (0, templist);

   return templist.iterator();
}
```

```
/**
 * Performs a recursive inorder traversal.
 *
 * @param node      the node used in the traversal
 * @param templist  the temporary list used in the traversal
 */

protected void inorder (int node, ArrayUnorderedList<T> templist){

   if (node < tree.length)
      if (tree[node] != null){
         inorder (node*2+1, templist);
         templist.addToRear(tree[node]);
         inorder ((node+1)*2, templist);
      }
}
```