Árvores

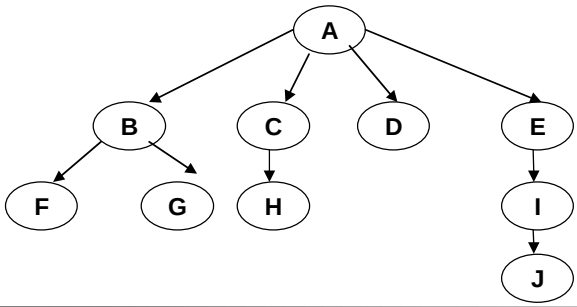
Sumário

- Definições
- Árvore binária
  - Nó de árvore
  - Implementação
- Iteradores de árvore
  - pré-ordem, pós-ordem, in-ordem
- Árvore de pesquisa binária
  - Nó de árvore
  - Implementação
- Árvore com nível

ARV

Árvores

- **Conjunto de nós e conjunto de arestas que ligam pares de nós**
  - Um nó é a *raiz*
  - Com excepção da raiz, todo o nó está ligado por uma aresta a 1 e 1 só nó (o pai)
  - Há um caminho único da raiz a cada nó; o *tamanho do caminho* para um nó é o número de arestas a percorrer



Nós sem descendentes: folhas

ARV

Árvores

- **Ramos da árvore**
  - Árvore de N nós tem N-1 ramos
- **Profundidade de um nó**
  - Comprimento do caminho da raiz até ao nó
    - Profundidade da raiz é 0
    - Produndidade de um nó é 1 + a profundidade do seu pai
- **Altura de um nó**
  - Comprimento do caminho do nó até à folha a maior profundidade
    - Altura de uma folha é 0
    - Altura de um nó é 1 + a altura do seu filho de maior altura
  - Altura da árvore: altura da raiz
- **Se existe caminho do nó u para o nó v**
  - u é antepassado de v
  - v é descendente de u
  - Tamanho de um nó: número de descendentes

ARV

Nó de árvore binária

```
// ******************PUBLIC OPERATIONS*********************
// int size( )            --> Return size of subtree at node
// int height( )          --> Return height of subtree at node
// void printPostOrder( ) --> Print a postorder tree traversal
// void printInOrder( )   --> Print an inorder tree traversal
// void printPreOrder( )  --> Print a preorder tree traversal
// BinaryNode duplicate( )--> Return a duplicate tree

/**
 * Binary node class with recursive routines to
 * compute size and height.
 */
final class BinaryNode
{
private Object     element;
    private BinaryNode left;
    private BinaryNode right;
```

ARV

Nó de árvore binária

```
public BinaryNode( )
{
    this( null, null, null ); }

public BinaryNode(Object theElement, BinaryNode lt, BinaryNode rt)
{
    element = theElement;
    left    = lt;
    right   = rt; }

public static int size( BinaryNode t )
{
    if( t == null )
        return 0;
    else
        return 1 + size( t.left ) + size( t.right ); }
```

ARV

Nó de árvore binária

```
public static int height( BinaryNode t )
{
    if( t == null )
        return -1;
    else
        return 1 + Math.max( height( t.left ),
                             height( t.right ) );
}


public void printPreOrder( )
{
    System.out.println( element );       // Node
    if( left != null )
        left.printPreOrder( );           // Left
    if( right != null )
        right.printPreOrder( );          // Right
}
```

ARV

Nó de árvore binária

```
public void printPostOrder( )
{
    if( left != null )
        left.printPostOrder( );          // Left
    if( right != null )
        right.printPostOrder( );         // Right
    System.out.println( element );       // Node
}

public void printInOrder( )
{
    if( left != null )
        left.printInOrder( );            // Left
    System.out.println( element );       // Node
    if( right != null )
        right.printInOrder( );           // Right
}
```

ARV

Nó de árvore binária

```
/**
 * Return a reference to a node that is the root of a
 * duplicate of the binary tree rooted at the current node.
 */
public BinaryNode duplicate( )
{
    BinaryNode root = new BinaryNode( element, null, null );

    if( left != null )            // If there's a left subtree
        root.left = left.duplicate( );    // Duplicate; attach
    if( right != null )          // If there's a right subtree
        root.right = right.duplicate( );  // Duplicate; attach
    return root;                        // Return resulting tree
}
```

ARV

Nó de árvore binária

```
    public Object getElement( )
    {return element; }

    public BinaryNode getLeft( )
    { return left; }

    public BinaryNode getRight( )
    { return right; }

    public void setElement( Object x )
    { element = x; }

    public void setLeft( BinaryNode t )
    { left = t;  }

    public void setRight( BinaryNode t )
    { right = t; }}
```

ARV

Árvore binária

```
public class BinaryTree
{

  private BinaryNode root;


    public BinaryTree( )
    {
        root = null;
    }

    public BinaryTree( Object rootItem )
    {
        root = new BinaryNode( rootItem, null, null );
    }
```

ARV

Árvore binária

```
public void printPreOrder( )
{
    if( root != null )
        root.printPreOrder( );
}

public void printInOrder( )
{
    if( root != null )
       root.printInOrder( );
}

public void printPostOrder( )
{
    if( root != null )
       root.printPostOrder( );
}
```

ARV

Árvore binária

```
public void makeEmpty( )
{
    root = null; }

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

public int size( )
{
    return BinaryNode.size( root ); }

public int height( )
{
    return BinaryNode.height( root ); }

public BinaryNode getRoot( )
{
    return root; }
```

ARV

Árvore binária

```
    public void merge(Object rootItem, BinaryTree t1, BinaryTree t2)
    {
        if( t1.root == t2.root && t1.root != null )
        {
            System.err.println("leftTree==rightTree; merge aborted");
            return;
        }

            // Allocate new node
        root = new BinaryNode( rootItem, t1.root, t2.root );

            // Ensure that every node is in one tree
        if( this != t1 )
            t1.root = null;
        if( this != t2 )
            t2.root = null;
    }
}
```

ARV

Iteradores em Árvore binária

```
import java.util.NoSuchElementException;

import weiss.nonstandard.Stack;
import weiss.nonstandard.Queue;
import weiss.nonstandard.ArrayStack;
import weiss.nonstandard.ArrayQueue;

// TreeIterator class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS*********************
//     first and advance are abstract; others are final
// boolean isValid( )   --> True if at valid position in tree
// Object retrieve( )   --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance (prefix)
// ******************ERRORS********************************
// Exceptions thrown for illegal access or advance
```

ARV

Iteradores em Árvore binária

```
abstract class TreeIterator
{
        /** The tree. */
    protected BinaryTree t;          // Tree
        /** Current position. */
    protected BinaryNode current;    // Current position

    public TreeIterator( BinaryTree theTree )
    { t = theTree; current = null; }

    abstract public void first( );

    final public boolean isValid( )
    { return current != null; }

    final public Object retrieve( )
    { if( current == null )
            throw new NoSuchElementException("TreeIterator retrieve");
        return current.getElement( ); }

    abstract public void advance( ); }
```

ARV

Iterador em pós-ordem

```
class PostOrder extends TreeIterator
{
     protected Stack s;    // The stack of StNode objects

    public PostOrder( BinaryTree theTree )
    {
        super( theTree );
        s = new ArrayStack( );
        s.push( new StNode( t.getRoot( ) ) );
    }

    public void first( )
    {
        s.makeEmpty( );
        if( t.getRoot( ) != null )
            s.push( new StNode( t.getRoot( ) ) );
        try
          { advance( ); }
        catch( NoSuchElementException e ) { }  // Empty tree
    }
```

ARV

Iterador em pós-ordem

```
public void advance( )
{
    if( s.isEmpty( ) )
    {
        if( current == null )
            throw new NoSuchElementException("PostOrder Advance");
        current = null;
        return;
    }

    StNode cnode;

    for( ; ; )
    {
        cnode = ( StNode ) s.topAndPop( );
```


Iterador em pós-ordem

```
        if( ++cnode.timesPopped == 3 )
        {
            current = cnode.node;
            return;
        }

        s.push( cnode );
        if( cnode.timesPopped == 1 )
        {
            if( cnode.node.getLeft( ) != null )
                s.push( new StNode( cnode.node.getLeft( ) ) );
        }
        else  // cnode.timesPopped == 2
        {
            if( cnode.node.getRight( ) != null )
                s.push( new StNode( cnode.node.getRight( ) ) );
        }
    }
}
```

Iterador em pós-ordem

```
      // An internal class for tree iterators
    protected static class StNode
    {
        BinaryNode node;
        int timesPopped;

        StNode( BinaryNode n )
        {
           node = n;
           timesPopped = 0;
        }
    }
}
```

ARV

Iterador em in-ordem

```
class InOrder extends PostOrder
{
    public InOrder( BinaryTree theTree )
    {
        super( theTree );
    }

    public void advance( )
    {
        if( s.isEmpty( ) )
        {
            if( current == null )
                throw new NoSuchElementException( "InOrder advance" );
            current = null;
            return;
        }

        StNode cnode;

        for( ; ; )
        {
            cnode = ( StNode ) s.topAndPop( );
```

ARV

Iterador em in-ordem

```
            if( ++cnode.timesPopped == 2 )
            {
                current = cnode.node;
                if( cnode.node.getRight( ) != null )
                    s.push( new StNode( cnode.node.getRight( ) ) );
                return;
            }

                // First time through
            s.push( cnode );
            if( cnode.node.getLeft( ) != null )
                s.push( new StNode( cnode.node.getLeft( ) ) );
        }
    }
}
```

ARV

Iterador em pré-ordem

```
class PreOrder extends TreeIterator
{
    private Stack s;

    public PreOrder( BinaryTree theTree )
    {
        super( theTree );
        s = new ArrayStack( );
        s.push( t.getRoot( ) );
    }

    public void first( )
    {
        s.makeEmpty( );
        if( t.getRoot( ) != null )
            s.push( t.getRoot( ) );
        try
          { advance( ); }
        catch( NoSuchElementException e ) { } }
```

ARV

Iterador em pré-ordem

```
public void advance( )
    {
        if( s.isEmpty( ) )
        {
            if( current == null )
                throw new NoSuchElementException("PreOrder Advance");
            current = null;
            return;
        }

        current = ( BinaryNode ) s.topAndPop( );

        if( current.getRight( ) != null )
            s.push( current.getRight( ) );
        if( current.getLeft( ) != null )
            s.push( current.getLeft( ) );
    }
}
```

ARV

Iterador em ordem de nível

```
class LevelOrder extends TreeIterator
{
    private Queue q;

    public LevelOrder( BinaryTree theTree )
    {
        super( theTree );
        q = new ArrayQueue( );
        q.enqueue( t.getRoot( ) );
    }

    public void first( )
    {
        q.makeEmpty( );
        if( t.getRoot( ) != null )
            q.enqueue( t.getRoot( ) );
        try
          { advance( ); }
        catch( NoSuchElementException e ) { } // Empty tree
    }
```

ARV

Iterador em ordem de nível

```
    public void advance( )
    {
        if( q.isEmpty( ) )
        {
            if( current == null )
               throw new NoSuchElementException("LevelOrder advance");
            current = null;
            return;
        }

        current = ( BinaryNode ) q.dequeue( );

        if( current.getLeft( ) != null )
            q.enqueue( current.getLeft( ) );
        if( current.getRight( ) != null )
            q.enqueue( current.getRight( ) );
    }
}
```

ARV

Árvores de pesquisa binária

- **Pesquisa em estrutura linear com elementos ordenados: pode ser feita em O(log n)**
  - ... sem inserção ou remoção de elementos
- **Manter o tempo de acesso logarítmico com inserção e remoção**
  - estrutura em árvore binária
  - mais operações do que árvore básica: pesquisar, inserir, remover
- **Objectos nos nós devem ser comparáveis**
  - interface Comparable: objectos de classes que a implementam têm critério de comparação

ARV

Operações em árvores de pesquisa binária

- **Pesquisa**
  - Usa a propriedade de ordem na árvore para escolher caminho, eliminando 1 subárvore a cada comparação
- **Inserção**
  - Como pesquisa, novo nó inserido one a pesquisa falha
- **Máximo e mínimo**
  - Procura escolhendo sempre a árvore direita ou sempre a árvore esquerda
- **Remoção**
  - Nó folha - apagar nó
  - Nó com 1 filho: filho substitui o pai
  - Nó com 2 filhos: elemento é substituído pelo maior da subárvore esquerda (ou menor da direita); o nó deste tem no máximo 1 filho e é apagado

ARV

Nó de árvore de pesquisa binária

```
package weiss.nonstandard;

// Basic node stored in unbalanced binary search trees
// Note that this class is not accessible outside
// of this package.

class BinaryNode
{
        // Constructors
    BinaryNode( Comparable theElement )
    {
        element = theElement;
        left = right = null;
    }

      // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

ARV

Árvore de pesquisa binária

```
package weiss.nonstandard;

// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// void removeMin( )      --> Remove minimum item
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Exceptions are thrown by insert, remove, and removeMin if warranted
```

ARV

Árvore de pesquisa binária

```
public class BinarySearchTree
{
     /** The tree root. */
    protected BinaryNode root;

    public BinarySearchTree( )
    { root = null;}

    public void insert( Comparable x )
    { root = insert( x, root ); }

    public void remove( Comparable x )
    { root = remove( x, root ); }

    public void removeMin( )
    { root = removeMin( root ); }
```

ARV

Árvore de pesquisa binária

```
public Comparable findMin( )
{ return elementAt( findMin( root ) ); }

public Comparable findMax( )
{ return elementAt( findMax( root ) ); }

public Comparable find( Comparable x )
{ return elementAt( find( x, root ) ); }

public void makeEmpty( )
{ root = null;}

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

ARV

Árvore - inserir

```
private Comparable elementAt( BinaryNode t )
{
    return t == null ? null : t.element;
}

protected BinaryNode insert( Comparable x, BinaryNode t )
{
    if( t == null )
        t = new BinaryNode( x );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        throw new DuplicateItemException( x.toString( ) );
    return t;
}
```

ARV

Árvore - remover

```
 protected BinaryNode remove( Comparable x, BinaryNode t )
{
    if( t == null )
        throw new ItemNotFoundException( x.toString( ) );
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null )
    {
        t.element = findMin( t.right ).element;
        t.right = removeMin( t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}
```

ARV

Árvore - remover mínimo

```
protected BinaryNode removeMin( BinaryNode t )
{
    if( t == null )
        throw new ItemNotFoundException( );
    else if( t.left != null )
    {
        t.left = removeMin( t.left );
        return t;
    }
    else
        return t.right; }

protected BinaryNode findMin( BinaryNode t )
{
    if( t != null )
        while( t.left != null )
            t = t.left;

    return t;}
```

ARV

Árvore - máximo

```
    private BinaryNode findMax( BinaryNode t )
    {
        if( t != null )
            while( t.right != null )
                t = t.right;

        return t; }

    private BinaryNode find( Comparable x, BinaryNode t )
    {
        while( t != null )
        {
            if( x.compareTo( t.element ) < 0 )
                t = t.left;
            else if( x.compareTo( t.element ) > 0 )
                t = t.right;
            else
                return t;
        }

        return null; } }
```

ARV

Nó de Árvore com nível

```
package weiss.nonstandard;

class BinaryNodeWithSize extends BinaryNode
{
    BinaryNodeWithSize( Comparable x )
    {
        super( x );
        size = 0;
    }

    int size;
}
```

ARV

Árvore com nível

```
public class BinarySearchTreeWithRank extends BinarySearchTree
{
    public Comparable findKth( int k )
    {
        return findKth( k, root ).element;
    }

    protected BinaryNode findKth( int k, BinaryNode t )
    {
        if( t == null )
            throw new IllegalArgumentException( );
        int leftSize = ( t.left != null ) ?
                       ((BinaryNodeWithSize) t.left).size : 0;

        if( k <= leftSize )
            return findKth( k, t.left );
        if( k == leftSize + 1 )
            return t;
        return findKth( k - leftSize - 1, t.right );
    }
```

ARV

Árvore com nível

```
protected BinaryNode insert( Comparable x, BinaryNode tt )
   {
        BinaryNodeWithSize t = (BinaryNodeWithSize) tt;

        if( t == null )
            t = new BinaryNodeWithSize( x );
        else if( x.compareTo( t.element ) < 0 )
            t.left = insert( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = insert( x, t.right );
        else
            throw new DuplicateItemException( x.toString( ) );
        t.size++;
        return t;
   }
```

ARV

Árvore com nível

```
    protected BinaryNode remove( Comparable x, BinaryNode tt )
    {
        BinaryNodeWithSize t = (BinaryNodeWithSize) tt;

        if( t == null )
            throw new ItemNotFoundException( x.toString( ) );
        if( x.compareTo( t.element ) < 0 )
            t.left = remove( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = remove( x, t.right );
        else if( t.left != null && t.right != null )
        {
            t.element = findMin( t.right ).element;
            t.right = removeMin( t.right );
        }
        else
            return ( t.left != null ) ? t.left : t.right;

        t.size--;
        return t;  }
```

ARV

Árvore com nível

```
    protected BinaryNode removeMin( BinaryNode tt )
    {
        BinaryNodeWithSize t = (BinaryNodeWithSize) tt;

        if( t == null )
            throw new ItemNotFoundException( );
        if( t.left == null )
            return t.right;

        t.left = removeMin( t.left );
        t.size--;
        return t;
    }
}
```

ARV