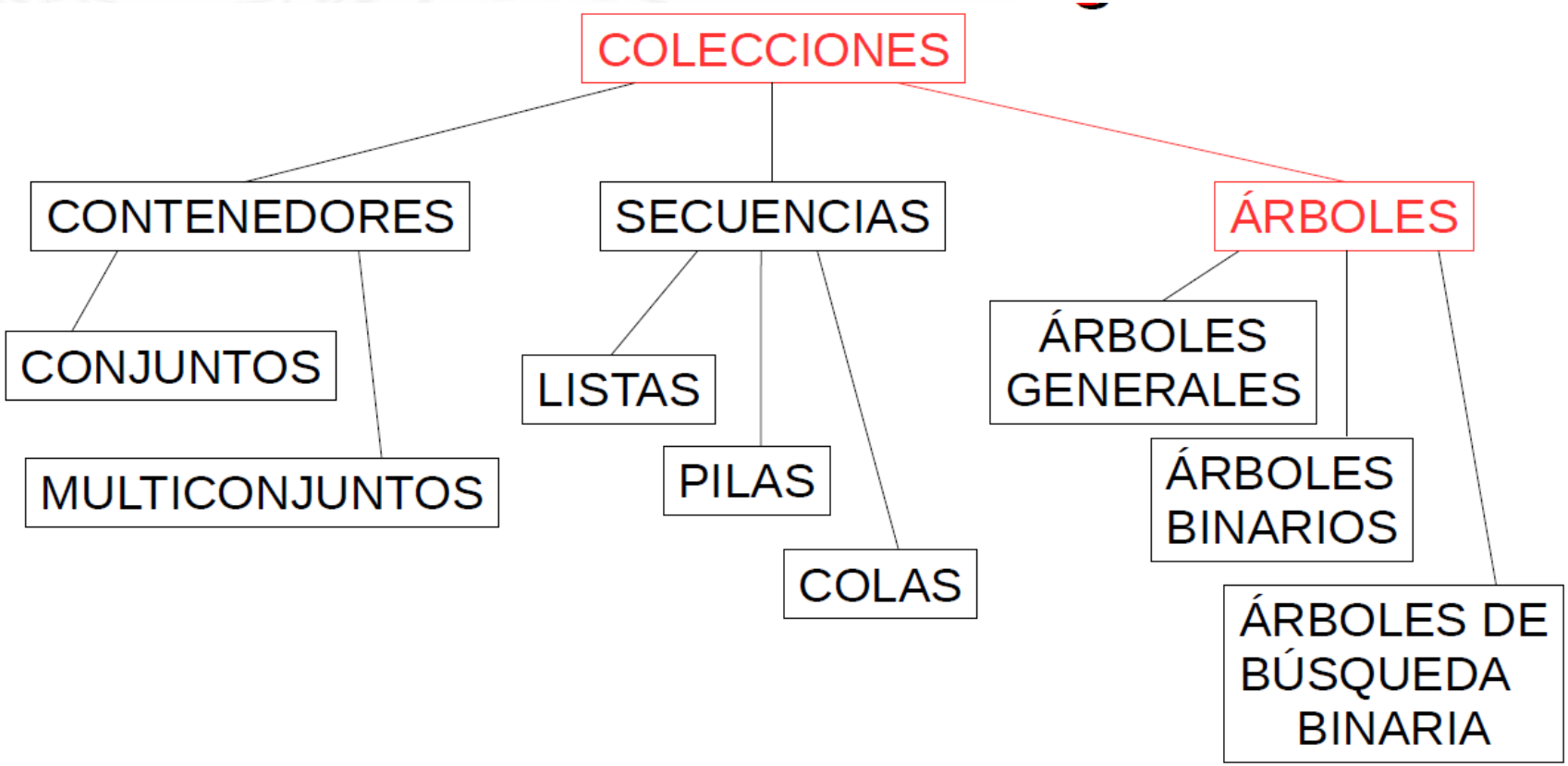
Tema 6 Árboles

ESTRATEGIAS DE PROGRAMACION Y ESTRUCTURAS DE DATOS

CA Guadalajara (UNED)

Índice

Implementación de Árboles



Estructuras Lineales vs Jerárquicas

Estructuras lineales:

- Todo elemento → un único predecesor (salvo el 1º)
- Todo elemento → un único sucesor (salvo el último)

Estructuras jerárquicas:

- Cualquier elemento puede tener
  - varios predecesores y/o varios sucesores
- Árboles → un predecesor, varios sucesores
- Grafos → varios predecesores, varios sucesores

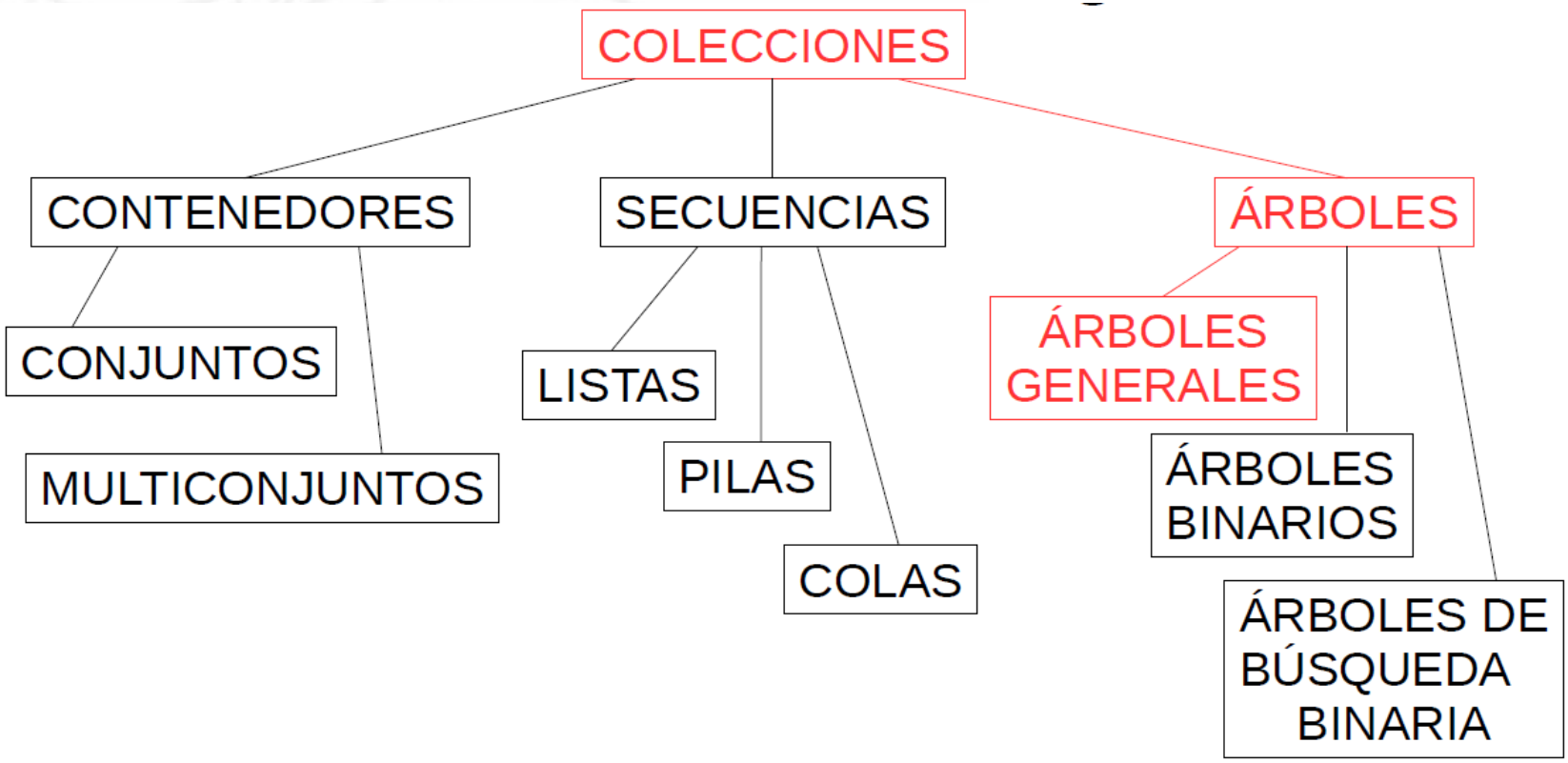
Clase Abstracta Árbol

```
public abstract class Tree<E> extends Collection<E> implements
TreeIF<E> {

  protected E root;

  Tree() { super(); this.root = null; }

  /* Devuelve el elemento situado en la raíz del árbol */
  public E getRoot() {
    return this.root;
  }

  /* Decide si el árbol es una hoja */
  public boolean isLeaf() {
    return this.root!=null && getNumChildren() == 0;
  }
```

Clase Abstracta Árbol

```
  /* Reimplementación de algunos métodos de Collection */

  /* Decide si el árbol es vacío */
  public boolean isEmpty() { return this.root==null; }

  /* Vacía el árbol */
  public void clear() { super.clear(); this.root = null; }

  abstract public int getNumChildren();

  abstract public int getFanOut();

  abstract public int getHeight();

  abstract public IteratorIF<E> iterator(Object mode);

  abstract public boolean contains(E e);
}
```

Implementación de Árboles Generales



Implementación de Árboles Generales

Número de hijos no acotado (puede ser 0)

¿Qué operaciones hacen falta?

- Modificar la raíz del árbol
- Obtener todos los hijos
- Obtener un hijo concreto
- Insertar un hijo concreto
- Eliminar un hijo concreto

Implementación de Árboles Generales

```
public class GTree<E> extends Tree<E> implements GTreeIF<E> {

  private ListIF<GTreeIF<E>> children;

  /* Constructor por defecto: crea un árbol vacío */
  public GTree() { super(); this.children = new List<GTreeIF<E>>(); }

  public void setRoot(E e) { this.root = e; }

  public ListIF<GTreeIF<E>> getChildren() { return this.children; }

  public GTreeIF<E> getChild(int pos) {return this.children.get(pos);}

  public void addChild(int pos, GTreeIF<E> e) {
    this.children.insert(pos, e);}

  public void removeChild(int pos) { this.children.remove(pos); }
```

Implementación de Árboles Generales

```
/* Reimplementación/Especialización de métodos de Collection */

/* Devuelve el número de nodos del árbol */
public int size() {
  if ( isEmpty() ) { return 0; }
  int s = 1;
  IteratorIF<GTreeIF<E>> childIt = this.children.iterator();
  while ( childIt.hasNext() ) {
    s = s + childIt.getNext().size();
  }
  return s;
}

/* Vacía el árbol */
public void clear() { super.clear(); this.children.clear(); }
```

Implementación de Árboles Generales

```
/* Métodos heredados de CollectionIF */

/* Comprueba si el árbol contiene el elemento */
public boolean contains(E e) {
  if ( isEmpty() ) { return false; }
  boolean found = getRoot().equals(e);
  IteratorIF<GTreeIF<E>> childIt = this.children.iterator();
  while ( !found && childIt.hasNext() ) {
    found = childIt.getNext().contains(e);
  }
  return found;
}

/* Métodos heredados de TreeIF */

/* Devuelve el número de hijos del árbol */
public int getNumChildren() { return this.children.size(); }
```

Implementación de Árboles Generales

```
/* Devuelve el fan-out del árbol */
public int getFanOut() {
  if ( isEmpty() ) { return 0; }
  int fOut = getNumChildren();
  IteratorIF<GTreeIF<E>> childIt = this.children.iterator();
  while ( childIt.hasNext() ) {
    int aux = childIt.getNext().getFanOut();
    if ( aux > fOut ) { fOut = aux; }
  }
  return fOut;
}
```

Implementación de Árboles Generales

```
/* Devuelve la altura del árbol */
public int getHeight() {
  if ( isEmpty() ) { return 0; }
  int height = 0;
  IteratorIF<GTreeIF<E>> childIt = this.children.iterator();
  while ( childIt.hasNext() ) {
    int aux = childIt.getNext().getHeight();
    if ( aux > height ) { height = aux; }
  }
  return 1 + height;
}
```

Implementación de Árboles Generales

```
/* Devuelve un iterador sobre el árbol según el recorrido elegido */
public IteratorIF<E> iterator(Object mode) {
  QueueIF<E> queue = new Queue<E>();
  if ( mode instanceof GTree.IteratorModes ) {
    switch ((GTree.IteratorModes) mode) {
      case PREORDER:  preorder(this,queue);  break;
      case POSTORDER: postorder(this,queue); break;
      case BREADTH:   breadthLR(this,queue); break;
    }
  }
  return queue.iterator();
}
```

Implementación de Árboles Generales

```
/* Recorre el árbol en preorden */
private void preorder(GTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    q.enqueue(t.getRoot());
    IteratorIF<GTreeIF<E>> childIt = t.getChildren().iterator();
    while ( childIt.hasNext() ) {
      preorder(childIt.getNext(),q);
    }
  }
}
```

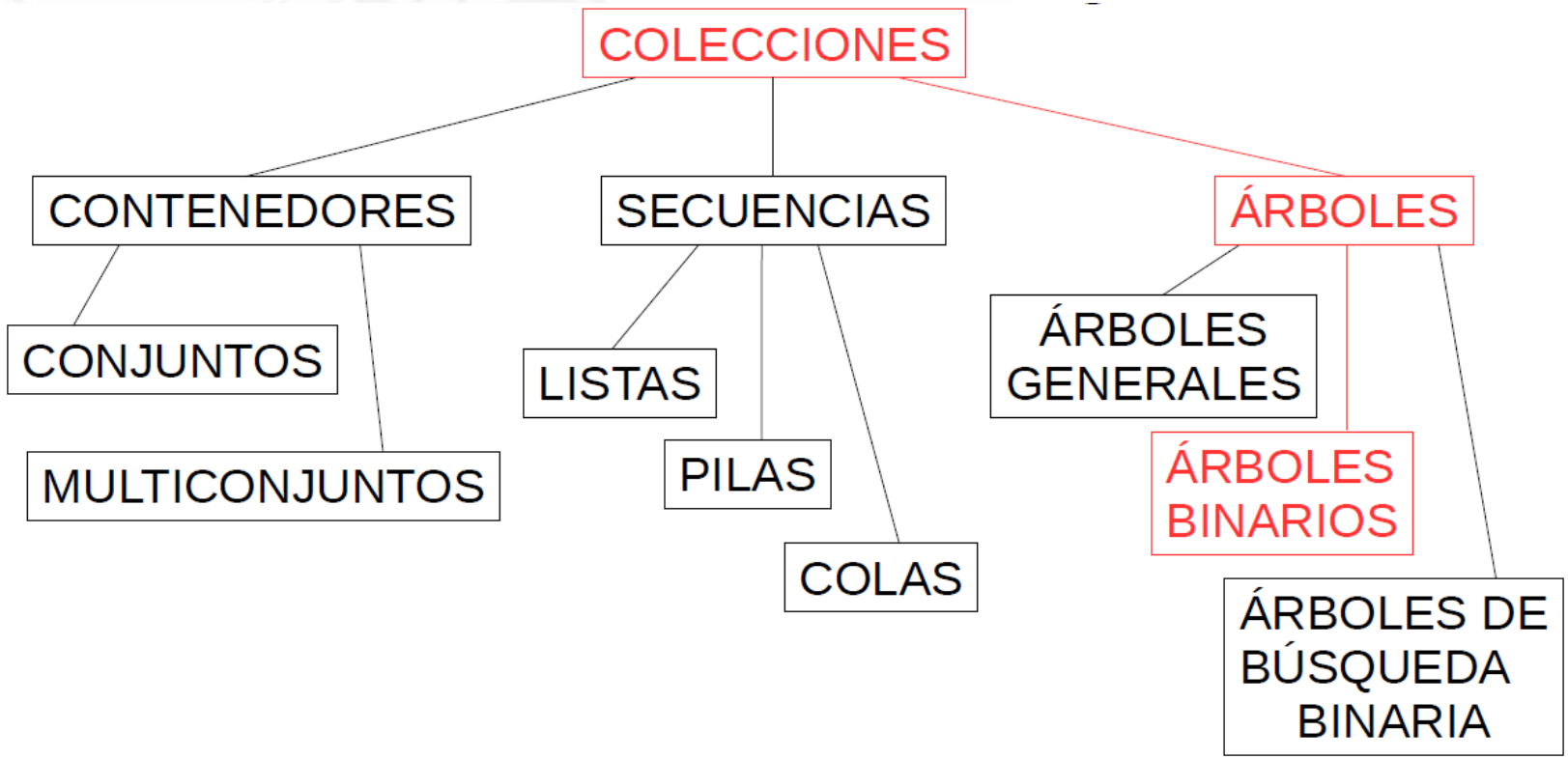
Implementación de Árboles Generales

```
/* Recorre el árbol en postorden */
private void postorder(GTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    IteratorIF<GTreeIF<E>> childIt = t.getChildren().iterator();
    while ( childIt.hasNext() ) {
      postorder(childIt.getNext(),q);
    }
    q.enqueue(t.getRoot());
  }
}
```

Implementación de Árboles Generales

```
/* Recorre el árbol en anchura de izquierda a derecha */
private void breadthLR(GTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    QueueIF<GTreeIF<E>> auxQ = new Queue<GTreeIF<E>>();
    auxQ.enqueue(t);
    while ( ! auxQ.isEmpty() ) {
      GTreeIF<E> cGT = auxQ.getFirst();
      q.enqueue(cGT.getRoot());
      IteratorIF<GTreeIF<E>> childIt = cGT.getChildren().iterator();
      while ( childIt.hasNext() ) {
        auxQ.enqueue(childIt.getNext());
      }
      auxQ.dequeue();
    }
  }
}
```

Implementación de Árboles Binarios



Implementación de Árboles Binarios

Número de hijos acotado: máximo 2

Normalmente: “hijo izquierdo” e “hijo derecho”

¿Qué operaciones hacen falta?

- Modificar la raíz
- Acceder, modificar y eliminar ambos hijos

Implementación de Árboles Binarios

```
public class BTree<E> extends Tree<E> implements BTreeIF<E> {

  private BTreeIF<E> leftChild; private BTreeIF<E> rightChild;

  /* Constructor por defecto: crea un árbol binario vacío */
  public Btree(){ super();this.leftChild = null;this.rightChild = null;}

  /* Devuelve el hijo izquierdo del árbol */
  public BTreeIF<E> getLeftChild() { return this.leftChild; }
  /* Devuelve el hijo derecho del árbol */
  public BTreeIF<E> getRightChild() { return this.rightChild; }
  /* Modifica la raíz */
  public void setRoot(E e) { this.root = e; }
  /* Modifica el hijo izquierdo */
  public void setLeftChild(BTreeIF<E> child) {this.leftChild = child;}
  /* Modifica el hijo derecho */
  public void setRightChild(BTreeIF<E> child) {this.rightChild = child;}
```

Implementación de Árboles Binarios

```
  /* Elimina el hijo izquierdo */
  public void removeLeftChild() { this.leftChild = null; }
  /* Elimina el hijo derecho */
  public void removeRightChild() { this.rightChild = null; }

/* Reimplementación/Especialización de algunos métodos de Collection */

  /* Devuelve el número de nodos del árbol */
  public int size() {
    if ( isEmpty() ) { return 0; }
    int s = 1;
    if ( this.leftChild != null ) { s = s + this.leftChild.size(); }
    if ( this.rightChild != null ) { s = s + this.rightChild.size(); }
    return s;
  }
  /* Vacía el árbol binario */
  public void clear() {super.clear();this.leftChild = null;
       this.rightChild = null;}
```

Implementación de Árboles Binarios

```
/* Métodos heredados de CollectionIF */

/* Comprueba si el árbol binario contiene el elemento */
public boolean contains(E e) {
  return ( !isEmpty() && ( this.root.equals(e) ||
     ( this.leftChild != null && this.leftChild.contains(e) ) ||
     ( this.rightChild != null && this.rightChild.contains(e) ) ) );
}

/* Métodos heredados de TreeIF */

/* Devuelve el número de hijos del árbol */
public int getNumChildren() {
  int nC = 0;
  if ( this.leftChild != null ) { nC++; }
  if ( this.rightChild != null ) { nC++; }
  return nC;
}
```

Implementación de Árboles Binarios

```java
/* Devuelve el fan-out del árbol */
public int getFanOut() {
  if ( getNumChildren() == 2 ) { return 2; }
  if ( this.leftChild != null ) {
    return Math.max(1,this.leftChild.getFanOut()); }
  if ( this.rightChild != null ) {
    return Math.max(1,this.rightChild.getFanOut()); }
  return 0;
}

/* Devuelve la altura del árbol */
public int getHeight() {
  if ( isEmpty() ) { return 0; }
  int hLC = 0; int hRC = 0;
  if ( this.leftChild != null ) { hLC = this.leftChild.getHeight(); }
  if ( this.rightChild != null ) { hRC = this.rightChild.getHeight();}
  return 1 + ((hLC > hRC)?hLC:hRC);
}
```

Implementación de Árboles Binarios: Recorrido

Recorridos en profundidad:

- ¿Cuándo se visita la raíz?
  - Antes de visitar los hijos → **preorden**
  - Después de visitar todos los hijos → **postorden**
  - Tras visitar H.I. y antes de visitar H.D. → **inorden**

Recorridos en anchura

- Se recorren los nodos de izquierda a derecha
- Se recorren los nodos de derecha a izquierda

Implementación de Árboles Binarios: Recorrido

```
/* Devuelve un iterador sobre el árbol según el recorrido elegido */
public IteratorIF<E> iterator(Object mode) {
  QueueIF<E> queue = new Queue<E>();
  if ( mode instanceof BTreeIF.IteratorModes ) {
    switch ((BTreeIF.IteratorModes) mode) {
      case PREORDER:  preorder(this,queue);  break;
      case INORDER:   inorder(this,queue);   break;
      case POSTORDER: postorder(this,queue); break;
      case BREADTH:   breadthLR(this,queue); break;
      case RLBREADTH: breadthRL(this,queue); break;
    }
  }
  return queue.iterator();
}
```

Implementación de Árboles Binarios: Recorrido

```
/* Recorre el árbol en preorden */
private void preorder(BTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    q.enqueue(t.getRoot());
    if ( t.getLeftChild() != null ) { preorder(t.getLeftChild(),q); }
    if ( t.getRightChild() != null ) { preorder(t.getRightChild(),q);}
  }
}

/* Recorre el árbol en inorden */
private void inorder(BTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    if ( t.getLeftChild() != null ) { inorder(t.getLeftChild(),q); }
    q.enqueue(t.getRoot());
    if ( t.getRightChild() != null ) { inorder(t.getRightChild(),q); }
  }
}
```

Implementación de Árboles Binarios: Recorrido

```
/* Recorre el árbol en postorden */
private void postorder(BTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    if ( t.getLeftChild() != null ) { postorder(t.getLeftChild(),q); }
    if ( t.getRightChild() != null ) {postorder(t.getRightChild(),q);}
    q.enqueue(t.getRoot());
  }
}
```

Implementación de Árboles Binarios: Recorrido

```
/* Recorre el árbol en anchura de izquierda a derecha */
private void breadthLR(BTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    QueueIF<BTreeIF<E>> auxQ = new Queue<BTreeIF<E>>();
    auxQ.enqueue(t);
    while ( ! auxQ.isEmpty() ) {
      BTreeIF<E> cBT = auxQ.getFirst();
      q.enqueue(cBT.getRoot());
      if ( cBT.getLeftChild() != null )
        { auxQ.enqueue(cBT.getLeftChild()); }
      if ( cBT.getRightChild() != null )
        { auxQ.enqueue(cBT.getRightChild()); }
      auxQ.dequeue();
    }
  }
}
```

Implementación de Árboles Binarios: Recorrido

```
/* Recorre el árbol en anchura de derecha a izquierda */
private void breadthRL(BTreeIF<E> t, QueueIF<E> q) {
  if ( !t.isEmpty() ) {
    QueueIF<BTreeIF<E>> auxQ = new Queue<BTreeIF<E>>();
    auxQ.enqueue(t);
    while ( ! auxQ.isEmpty() ) {
      BTreeIF<E> cBT = auxQ.getFirst();
      q.enqueue(cBT.getRoot());
      if ( cBT.getRightChild() != null )
        { auxQ.enqueue(cBT.getRightChild()); }
      if ( cBT.getLeftChild() != null )
        { auxQ.enqueue(cBT.getLeftChild()); }
      auxQ.dequeue();
    }
  }
}
```

Tema 6 Árboles

ESTRATEGIAS DE PROGRAMACION Y ESTRUCTURAS DE DATOS

CA Guadalajara (UNED)