//Complejidad: O(n)

**private void findLevel(BinaryTree<E> tree, Position<E> pos, int level, int midLevel, List<E> output)** {

if (level == midLevel)

output.add(pos.getElement());

if (level < midLevel)

for (Position<E> child : tree.children(pos))

findLevel(tree, child, level + 1, midLevel, output);

}

//Complejidad: O(n)

**private int height(BinaryTree<E> tree, Position<E> node)** {

int left = 0, right = 0;

if (tree.hasLeft(node))

left = height(tree, tree.left(node));

if (tree.hasRight(node))

right = height(tree, tree.right(node));

return 1 + Math.max(left, right);

}

***//No se puede modificar el árbol mientras se recorre con SU iterador.***

Algorithm inOrder(v)

if hasLeft(v) then

inOrder(left(v))

visit(v)

if hasRight(v) then

inOrder(right(v))

Algorithm preOrder(v)

visit(v)

for each child w of v

preOrder(w)

Algorithm postOrder(v)

for each child w of v

postOrder(w)

visit(v)

orden\_por\_nivel(raiz)

nueva cola

cola.encola(raíz)

while not cola.vacía do

nodo := cola.desencola()

visita(nodo)

cola.encola(nodo.hijos)

Arboles Rojo-Negro

Raiz = negra; hojas = negras; hijos de rojo = negro.

Inserciones: evitar rojo – rojo

Si tio es negro => reestructuración trinodo (nodo, padre, abuelo); b negro, a c rojo

Si tio es rojo => recoloreado arbol; padre tio negro, abuelo rojo. CUIDADO PROPAGACION

Borrado.

Si es hoja, fácil. Si es interno => se cambia por el menor de los mayores.