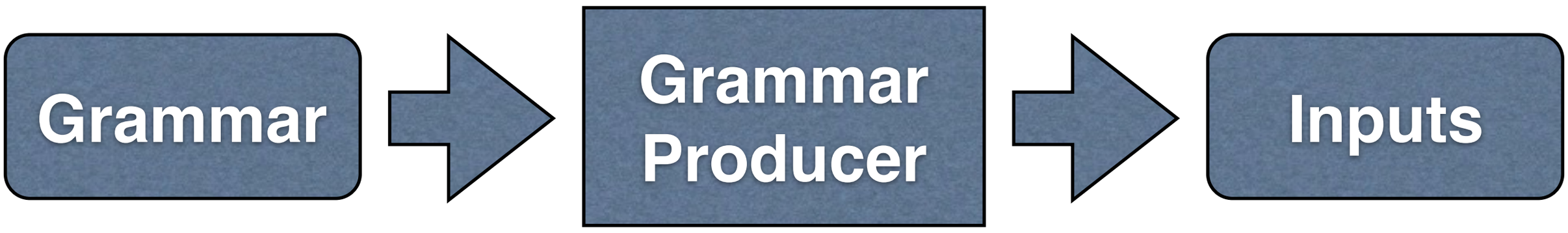
Taller #8 - Korat

Generación Automática de Tests - 2016

Grammar Producer



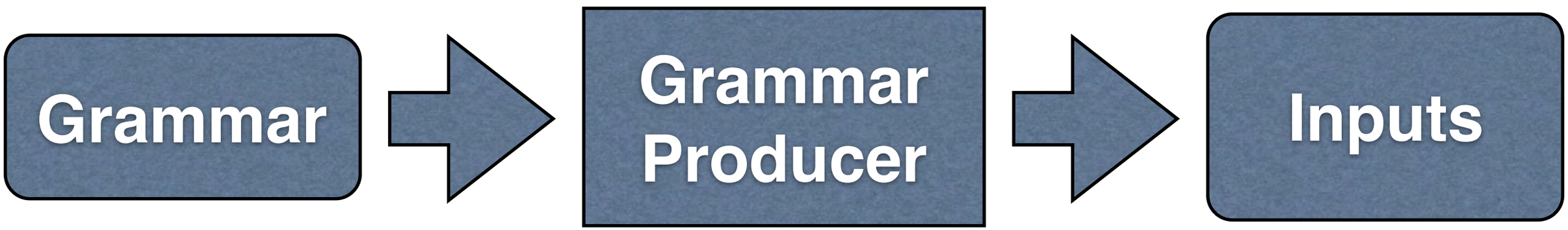

```
grammar = [
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]
```

```
x
(x x)
((x x) x)
(x (x x))
((x x) (x x))
(((x x) x) x)
((x (x x)) x)
(x ((x x) x))
(x (x (x x)))
```

Filtered Grammar-Based Testing

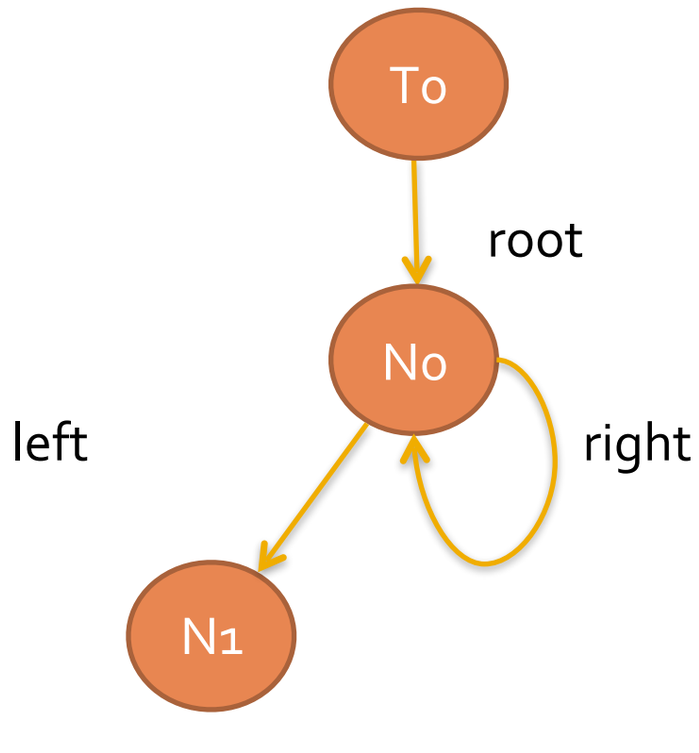


Ej:binary trees

Ej:arboles binarios de búsqueda

Generando Candidatos

Representación Gráfica



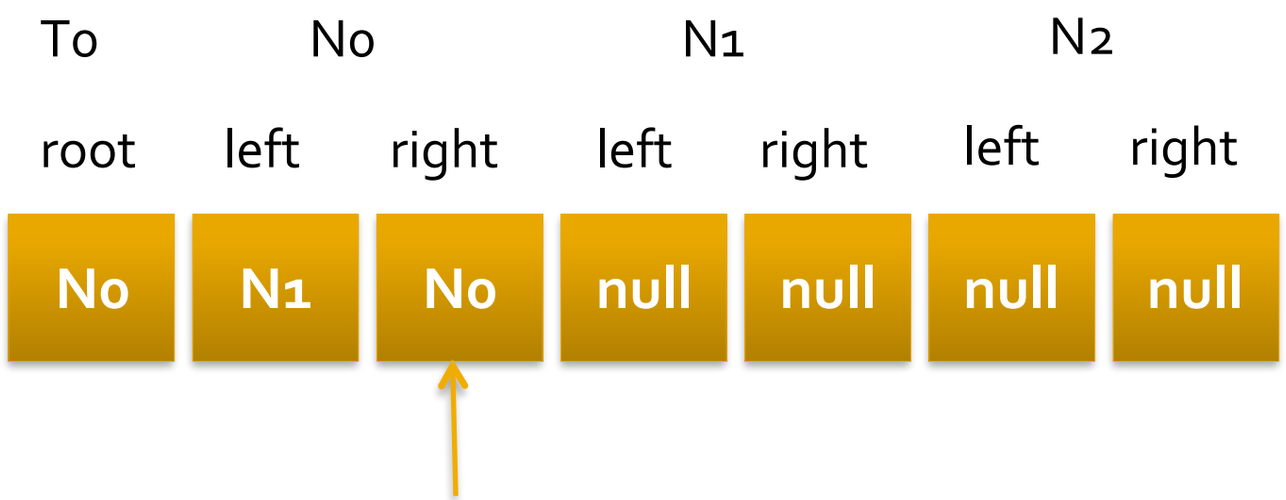
Field Ordering

1) To.root

2) No.left

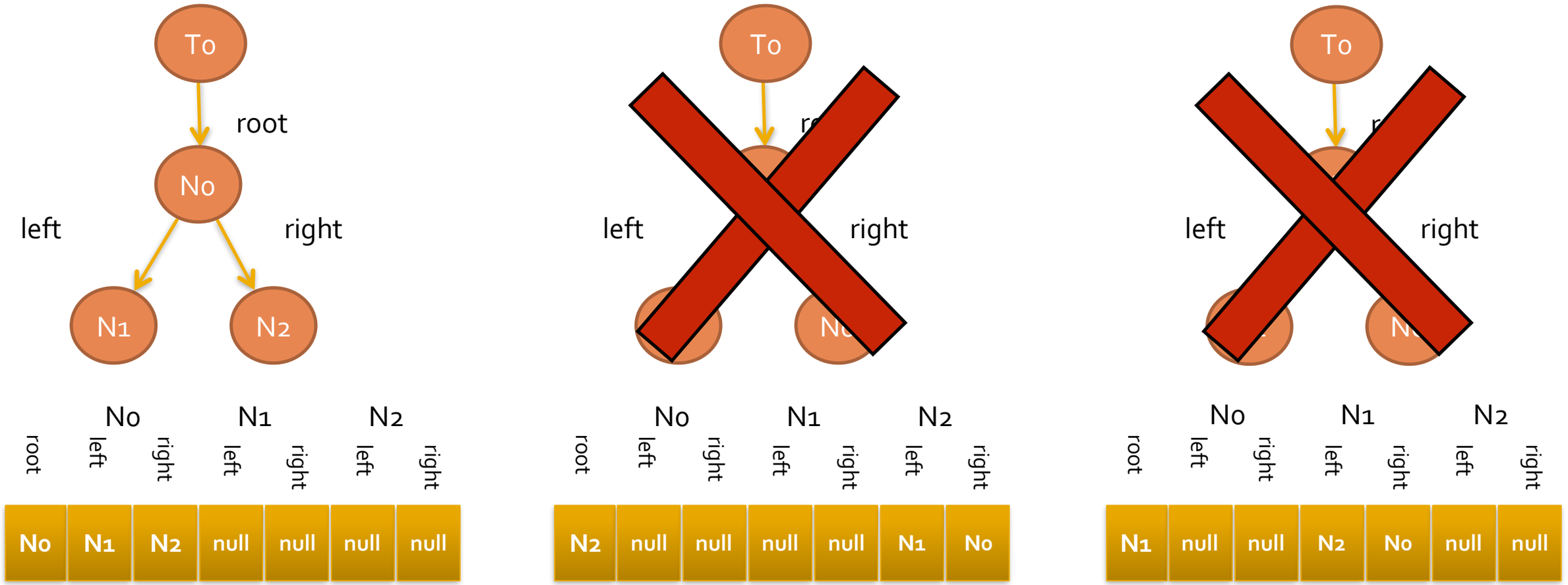
3) No.right

Candidate Vector



this.repOk()==false
(increase index)

Rotura de Simetrías



- El índice de un elemento *e* no puede ser mayor a *k+1*, donde *k* es el mayor de los índices de todos los elementos del mismo tipo que *e*

```
def generator(repOK, scope):

  # inicializamos el vector con el primer candidato
  vector = init(scope)

  while True:
    # ejecutar repOK() y obtener field ordering
    fOrder, ret_val = exec repOK(vector)
    # retorno el candidato si satisface repOK()
    if ret_val==True:
      yield vector

    # Obtengo el próximo candidato
    fIndex=fOrder.removeLast()
    while not incField(vector,fIndex):
      # Si ya no puedo incrementar el field,
      # hacemos backtracking
      if len(fOrder)>0:
        fIndex = fOrder.removeLast()
      else:
        # Si ya no se puede hacer backtracking,
        # no existen más candidatos no isomorfos
        return # causa StopIteration
```

Korat: Automated Testing Based on Java Predicates

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{chandra,khurshid,marinov}@lcs.mit.edu

ABSTRACT

This paper presents Korat, a novel framework for automated testing of Java programs. Given a formal specification for a method, Korat uses the method precondition to automatically generate all (nonisomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output.

To generate test cases for a method, Korat constructs a Java predicate (i.e., a method that returns a boolean) from the method's precondition. The heart of Korat is a technique for automatic test case generation: given a predicate and a bound on the size of its inputs, Korat generates all (nonisomorphic) inputs for which the predicate returns true. Korat exhaustively explores the bounded input space of the predicate but does so efficiently by monitoring the predicate's executions and pruning large portions of the search space.

This paper illustrates the use of Korat for testing several data structures, including some from the Java Collections Framework. The experimental results show that it is feasible to generate test cases from Java predicates, even when the search space for inputs is very large. This paper also compares Korat with a testing framework based on declarative specifications. Contrary to our initial expectation, the experiments show that Korat generates test cases much faster than the declarative framework.

1. INTRODUCTION

cate (i.e., a method that returns a boolean) from the method's precondition. One of the key contributions of Korat is a technique for automatic test case generation: given a predicate, and a bound on the size of its inputs, Korat generates all nonisomorphic inputs for which the predicate returns `true`. Korat uses backtracking to systematically explore the bounded input space of the predicate. Korat generates *candidate* inputs and checks their validity by invoking the predicate on them. Korat monitors accesses that the predicate makes to all the fields of the candidate input. If the predicate returns without reading some fields of the candidate, then the validity of the candidate must be independent of the values of those fields—Korat uses this observation to prune large portions of the search space. Korat also uses an optimization to generate only nonisomorphic test cases. (Section 3.4 gives a precise definition of nonisomorphism.) This optimization reduces the search time without compromising the exhaustive nature of the search.

Korat lets programmers write specifications in any language as long as the specifications can be automatically translated into Java predicates. We have implemented a prototype of Korat that uses the Java Modeling Language (JML) [20] for specifications. Programmers can use JML to write method preconditions and postconditions, as well as class invariants. JML uses Java syntax and semantics for expressions, and contains some extensions such as quantifiers. A large subset of JML can be automatically translated into Java predicates. Programmers can thus use Korat without having to learn a specification language much different than Java. Moreover, since JML specifications can call Java methods, programmers can use the

korat.sourceforge.net

Korat - Home Page

- Home
- Manual
- Tutorial
- Downloads
- Screenshots
- Publications
- About us

Welcome to Korat Home Page

Korat is a tool for constraint-based generation of structurally complex test inputs for Java programs. Structurally complex means that the inputs are structural (e.g., represented with linked data structures) and must satisfy complex constraints that relate parts of the structure e.g., invariants for linked data structures).

Korat requires (1) an imperative predicate that specifies the desired structural constraints and (2) a finitization that bounds the desired test input size. Korat generates all predicate inputs (within the bounds) for which the predicate returns true. To do so, Korat performs a systematic search of the predicate's input space. The inputs that Korat generates enable bounded-exhaustive testing for programs ranging from library classes to stand-alone applications.

Korat can graphically show the structures it generates. The visualization in Korat was inspired by Alloy, and our current Korat implementation uses the Alloy Analyzer's visualization facility, which provides a fully customizable display that allows users to specify desired views on the underlying structures. Korat automatically translates object graphs into the Alloy representation.

Korat

- Desarrollado originalmente en MIT
- Enumera inputs usando el algoritmo de generación de instancias basado en predicados descripto en la clase anterior
- Open-source (korat.sourceforge.net)
- Java 1.6 (Lamentablemente algunos features no funcionan bien Java >= 1.7)

Korat - Software Under Test

```
public class BinaryTree {
    public static class Node {
        Node left;
        Node right;
    }
    private Node root;
    private int size;
}
```

- Primero debemos declarar el SUT sobre el cual deseamos generar instancias

Korat - predicado repOk()

```
public boolean repOK() {
    if (root == null)
        return size == 0;
    // checks that tree has no cycle
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    // checks that size is consistent
    return (visited.size() == size);
}
```

- El Predicado debe retornar **true** sii la instancia es válida
- Debe ser **determinístico** (dada la misma estructura retornar el mismo valor)
- El orden de acceso debe ser el mismo

Korat - Scope

```
public static IFinitization finBinaryTree(int nodesNum, int minSize,
        int maxSize) {
    IFinitization f = FinitizationFactory.create(BinaryTree.class);
    IObjSet nodes = f.createObjSet(Node.class, nodesNum, true);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    IIntSet sizes = f.createIntSet(minSize, maxSize);
    f.set("size", sizes);
    return f;
}
```

- Necesitamos indiciar explícitamente el scope sobre el cual se realizará la generación

- Naming convention fin<<CLASSNAME>>(args)

Korat - Scope

- Javadoc:
  - korat.sourceforge.net/docs/korat_api/index.html
- Package: korat.finitization

Korat - Ejecución

```
$ java -cp classpath korat.Korat
  --visualize
  --class org.autotest.BinaryTree
  --args 3,3,3
```

Testing usando Korat

- Podemos usar Korat de al menos 2 maneras:
  - Generar instancias, serializarlas, y luego usarlas en JUnit Test Cases
  - Proveer a Korat de un método que es invocado cada vez que se genera una nueva instancia válida (i.e. un “parameterized unit test”)
    - Este es el **“observer”** design pattern.

ITestCaseListener

```
import korat.testing.ITestCaseListener;

class BinTreeListener implements ITestCaseListener {

  public void notifyNewTestCase(Object o) { … }

  public void notifyTestFinished(long numOfExplored, long
numOfGenerated) {…}

}
```

ITestCaseListener - Ejemplo

```
class CountNodesTestListener implements ITestCaseListener
{

  public void notifyNewTestCase(Object o) {
    BinaryTree t = (BinaryTree) o;

    int nodes = countNodes(t);

    assert t.size == nodes.size();

}
```

Korat - Ejecución

```
$ java -cp classpath korat.Korat

 --visualize

--class org.autotest.BinaryTree

--args 3,3,3

--listeners org.autotest.CountNodesTestListener
```

Requerimientos

- Java 1.6
- graphviz (http://www.graphviz.org/Download.php)
- Korat:
  - http://korat.sourceforge.net/downloads.html

Enunciado - Ejercicio #1

Considere la clase **BinaryTree** provista con su **RepOK** y su método de finitización. Genere utilizando **Korat** todos los árboles binarios de tamaño a lo sumo 3, usando valores enteros en el rango (0..3).

¿Cuántas instancias válidas se generan?

Enunciado - Ejercicio #2

Considere la clase **AvlTree** provista. Implemente para esta clase el invariante de representación imperativo, i.e., la rutina **repOK** (tenga en cuenta para esto el comentario que acompaña a esta rutina). Genere utilizando **Korat** todos los **AVL** de tamaño a lo sumo 4, conteniendo valores enteros en el rango (1..4).

*Un **AVL** es un árbol ordenado que además cumple que "... the heights of the two child subtrees of any node differ by at most one"*

¿Cuántas instancias válidas se generan?

Enunciado - Ejercicio #3

Ejecutar **Korat** con un parameterized unit test que inserte el valor 0 y compruebe el nuevo tamaño del árbol. Documente todos los bugs descubiertos mediante el uso de **Korat** (Sólo registre los bugs descubiertos, pero no los corrija).

¿Cuántas instancias se exploran en total?