A Comparison of Constraint-based and Sequence-based Generation of Complex Input Data Structures

Rohan Sharma, **Milos Gligoric**, Vilas Jagannath and Darko Marinov

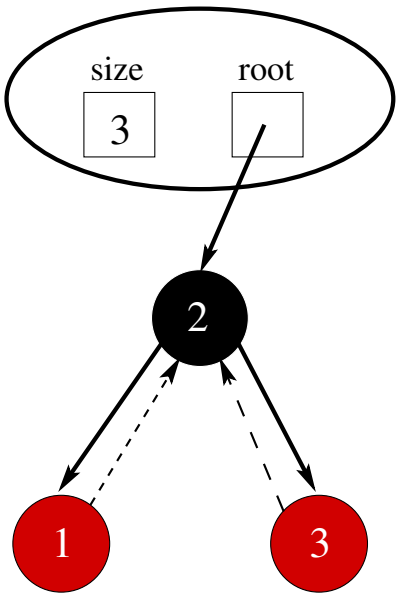
University of Illinois at Urbana-Champaign

April 10th, 2010
CSTVA 2010, Paris, France

Example Complex Data Structures

- Data structures that have linked nodes whose values need to satisfy some complex constraints

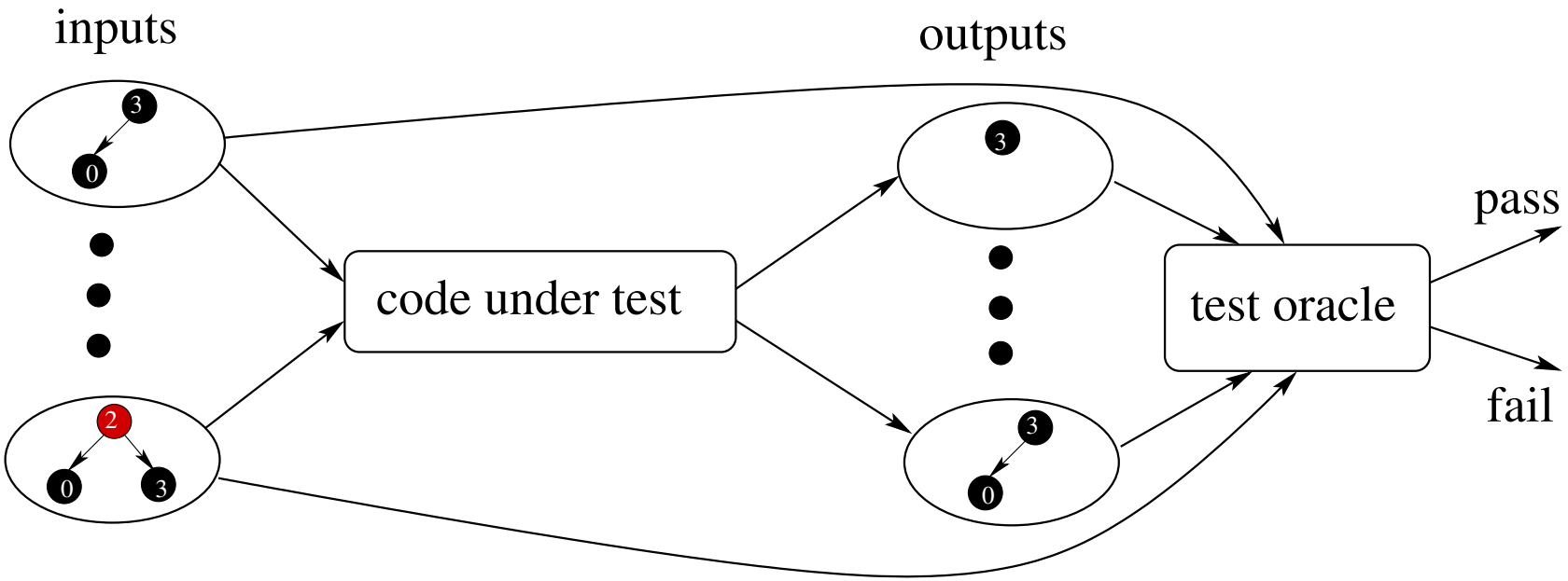
RedBlackTree



AST of Java Program

```
class A implements B {
  public A m() {
    A a = null;
    return a;
  }
}

interface B extends C {
  public B m();
}

interface C {
  public C m();
}
```

Testing Setup



- Example scenarios:
  - Testing "remove" on "RedBlackTree" (input: RBTs)
  - Testing compiler or refactoring engine (input: programs)
- Our focus: Generation of test inputs

Bounded-Exhaustive Testing

Automated testing approach that checks a system under test for all inputs within given bounds

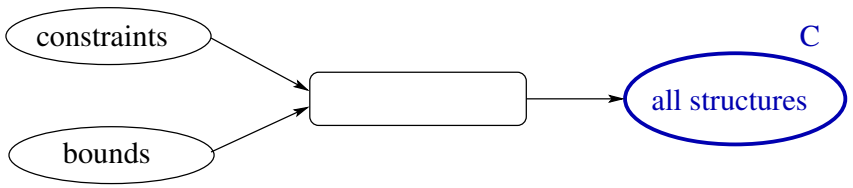
- Rationale: small-scope hypothesis [Jackson & Damon ISSTA 96]
- Exhaustive testing within bounds catches "corner cases"

Used in academia and industry to test real-world applications

- Refactoring Engines - Eclipse & NetBeans [Daniel et al. FSE 07]
- Web Traversal Agent from Google [Misailovic et al. FSE 07]
- XPath Compiler at Microsoft [Stobie ENTCS 05]
- Constraint Analyzer [Khurshid & Marinov J-ASE 04]
- Fault-Tree Analyzer for NASA [Sullivan et al. ISSTA 04]
- Protocol for Dynamic Networks [Khurshid & Marinov ENTCS 01]

Two Approaches for Automated Generation

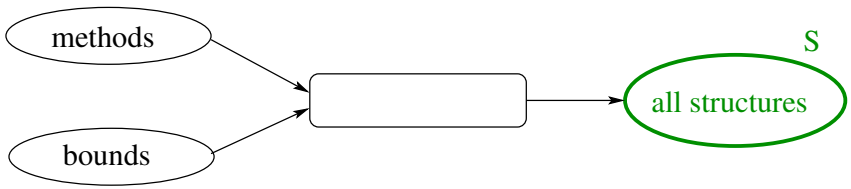
- Constraint-based Generation
  - Describe data structure constraints and provide bounds
  - Solve constraints
  - Generate structures at the concrete representation level



Two Approaches for Automated Generation

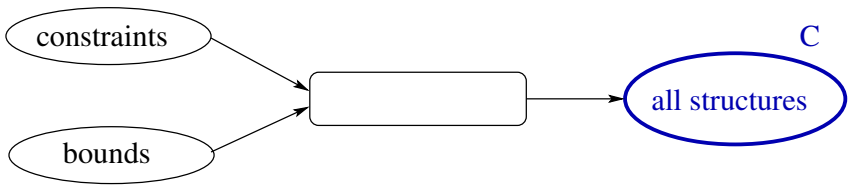
- Constraint-based Generation
  - Describe data structure constraints and provide bounds
  - Solve constraints
  - Generate structures at the concrete representation level

- Sequence-based Generation
  - Use sequences of methods
  - Provide length of sequence and parameter values
  - Generate structures at the abstract level

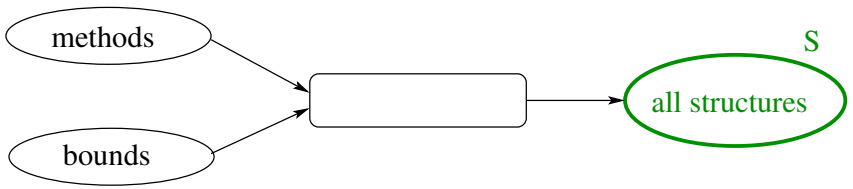


Two Approaches for Automated Generation

- Constraint-based Generation
  - Describe data structure constraints and provide bounds
  - Solve constraints
  - Generate structures at the concrete representation level



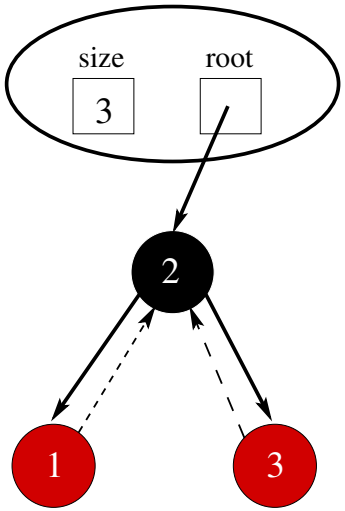
- Sequence-based Generation
  - Use sequences of methods
  - Provide length of sequence and parameter values
  - Generate structures at the abstract level



- Compare structures in C and S

Example - RedBlackTree

```
class RedBlackTree {
  int size;
  Node root;

  static boolean RED = false;
  static boolean BLACK = true;

  static class Node {
    Node left, right, parent;
    boolean color;
    int value;
  }

  RedBlackTree() { /* empty tree */ }

  void insert(int value) { ... }
  void remove(int value) { ... }
}
```



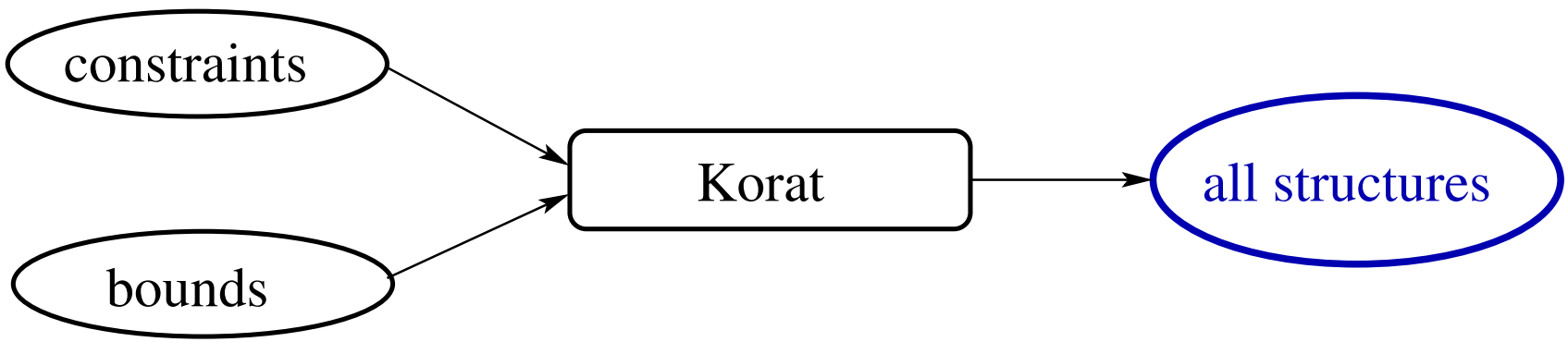
Example - Generated Structure

```
// constratint based

RedBlackTree rbt = new RedBlackTree();

Node n0 = new Node();
Node n1 = new Node();
Node n2 = new Node();

n0.color = BLACK;
n0.value = 2;
n0.left = n1;
n0.right = n2;
n0.parent = null;

n1.color = RED;
n1.value = 1;
n1.left = null;
n1.right = null;
n1.parent = n0;

n2.color = RED;
n2.value = 3;
n2.left = null;
n2.right = null;
n2.parent = n0;
```

```
// sequence based

RedBlackTree rbt = new RedBlackTree();

rbt.insert(2);
rbt.insert(1);
rbt.insert(3);
```

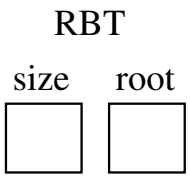
Constraint-based Generation: Korat

- http://mir.cs.illinois.edu/korat
- Input:
  - Constraints (written in standard implementation language)
  - Bound
- Output:
  - All structures where constraints are satisfied



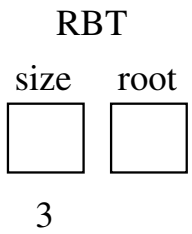
Constraint-based Generation: Bounds

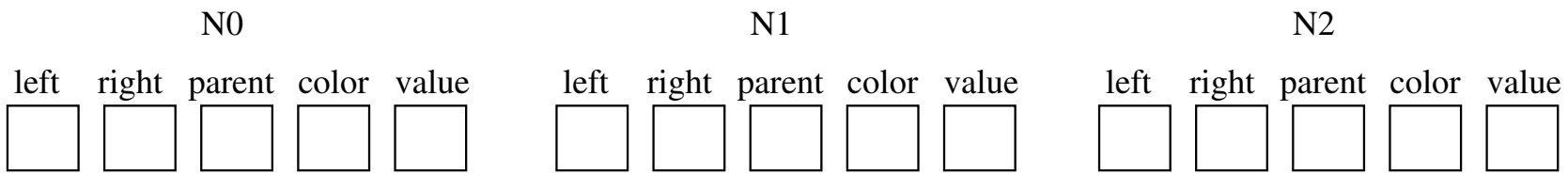
- User provides bounds



Constraint-based Generation: Bounds

- User provides bounds





Constraint-based Generation: Bounds

- User provides bounds

RBT

size	root
3	null
	N0
	N1
	N2

N0

left	right	parent	color	value
null	null	null		
N0	N0	N0		
N1	N1	N1		
N2	N2	N2		

N1

left	right	parent	color	value
null	null	null		
N0	N0	N0		
N1	N1	N1		
N2	N2	N2		

N2

left	right	parent	color	value
null	null	null		
N0	N0	N0		
N1	N1	N1		
N2	N2	N2		

Constraint-based Generation: Bounds

- User provides bounds

RBT

size	root
3	null N0 N1 N2

N0

left	right	parent	color	value
null N0 N1 N2	null N0 N1 N2	null N0 N1 N2	true false	1 2 3

N1

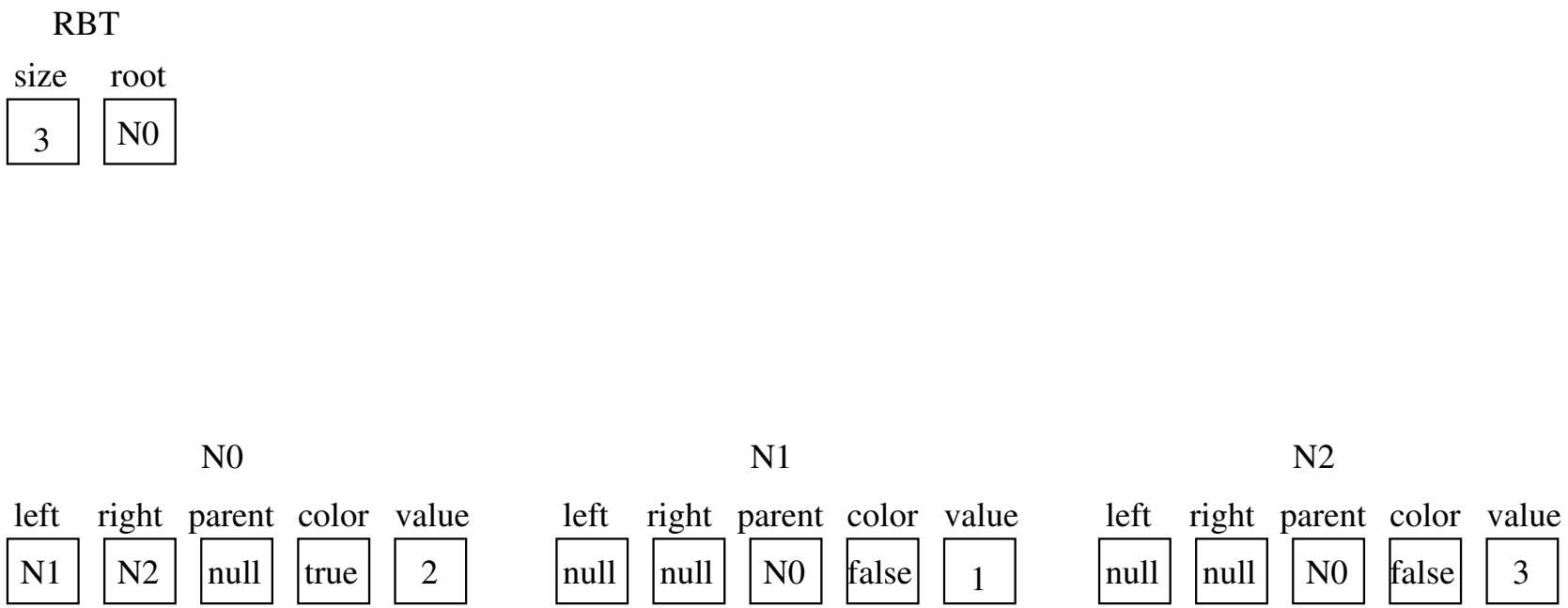
left	right	parent	color	value
null N0 N1 N2	null N0 N1 N2	null N0 N1 N2	true false	1 2 3

N2

left	right	parent	color	value
null N0 N1 N2	null N0 N1 N2	null N0 N1 N2	true false	1 2 3

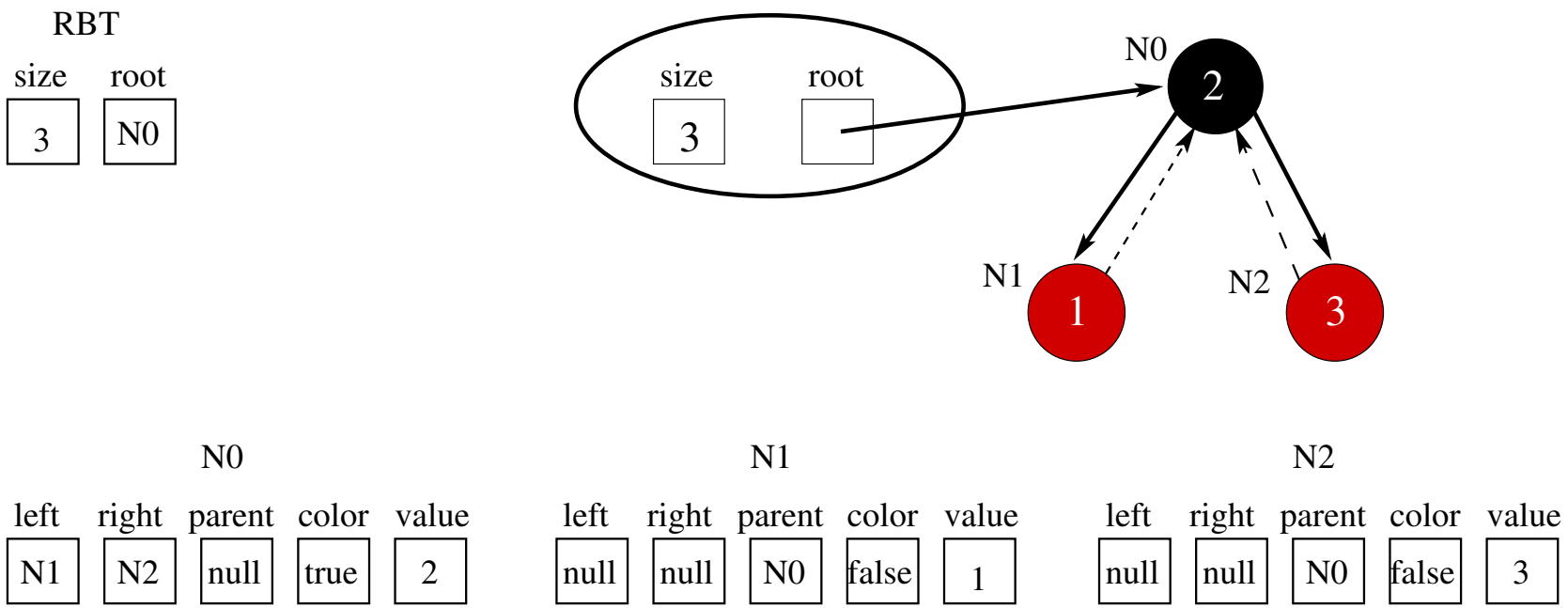
Constraint-based Generation: Bounds

- Korat automatically generates the structures



Constraint-based Generation: Bounds

- Korat automatically generates the structures

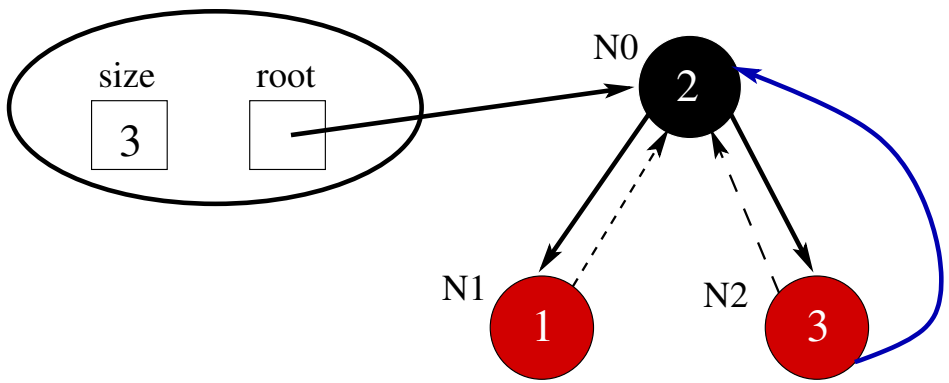


Constraint-based Generation: Bounds

- Korat automatically generates the structures

RBT

size	root
3	N0



N0

left	right	parent	color	value
N1	N2	null	true	2

N1

left	right	parent	color	value
null	null	N0	false	1

N2

left	right	parent	color	value
null	N0	N0	false	3

Constraint-based Generation: Constraints

- Korat checks the constraints on each candidate structure

```
boolean repOk() {
  return treeness() && coloring() && ordering();
}
```

- Treeness
  - No cycles
  - No two incoming edges to the same node
  - All nodes must be reachable from the root
- Coloring
  - Red node cannot have red children
  - Paths have the same number of black nodes
- Ordering
  - Nodes value ordered for binary search

Constraint-based Generation: Constraints

```
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new LinkedList<Node>();
  workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false;
      if (cl.parent != current) return false;
      workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false;
      if (cr.parent != current) return false;
      workList.add(cr);
    }
  }
  return size == visited.size();
}
```

Constraint-based Generation: Constraints

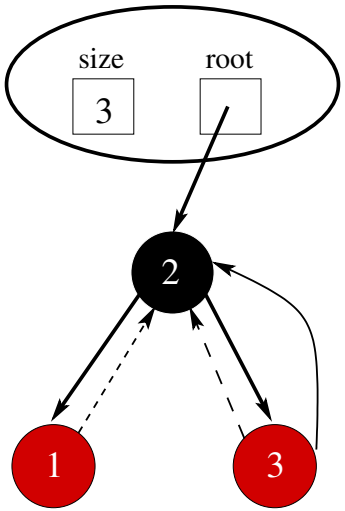
```
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new LinkedList<Node>();
  workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false; // no cycles & no two incoming edges
      if (cl.parent != current) return false;
      workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false; // no cycles & no two incoming edges
      if (cr.parent != current) return false;
      workList.add(cr);
    }
  }
  return size == visited.size();
}
```

Constraint-based Generation: Constraints

```
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new LinkedList<Node>();
  workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false;
      if (cl.parent != current) return false;
      workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false;
      if (cr.parent != current) return false;
      workList.add(cr);
    }
  }
  return size == visited.size(); // all nodes reachable
}
```

Constraint-based Generation: Constraints

```
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new LinkedList<Node>();
  workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false;
      if (cl.parent != current) return false;
      workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false;
      if (cr.parent != current) return false;
      workList.add(cr);
    }
  }
  return size == visited.size();
}
```



Constraint-based Generation

- Pros:
  - Does not require all methods on data structure
  - Can test one method in isolation
- Cons:
  - Relies on the user to provide constraints
  - Can generate false alarms or miss bugs

Sequence-based Generation

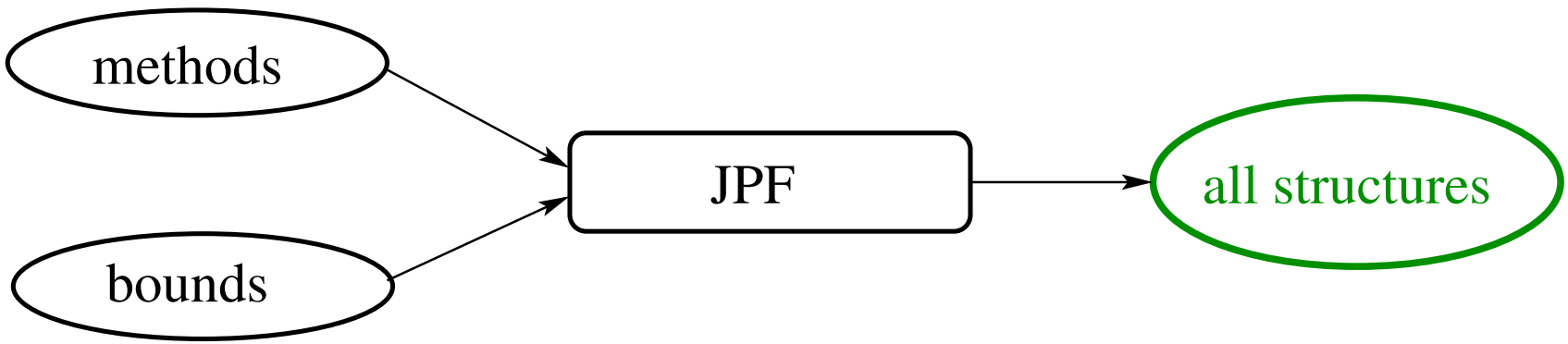
- Structures are generated from sequences of methods
- Structures are created at the abstract level
- Concrete representation is considered to prune generation

```
RedBlackTree rbt = new RedBlackTree();

rbt.insert(1);
rbt.remove(1);
```

```
RedBlackTree rbt = new RedBlackTree();

rbt.insert(2);
rbt.remove(2);
```

JavaPath Finder (JPF)

- Stateful model checker for Java programs
- JPF is JVM with backtracking capabilities
- Verify.getInt(int min, int max)



Sequence-based Generation

```
class RedBlackTreeDriver {
  static void main(String[] args) {



    RedBlackTree rbt = new RedBlackTree();

      switch (Verify.getInt(0, 1)) {
      case 0:
        rbt.insert(/* select value */);
        break;
      case 1:
        rbt.remove(/* select value */);
        break;
      }



  }
}
```

Sequence-based Generation

```
class RedBlackTreeDriver {
  static void main(String[] args) {

    // length of sequence (and method parameters)
    int lenOfSeq = ...;

    RedBlackTree rbt = new RedBlackTree();
    for (int i = 0; i < lenOfSeq; i++) {
      switch (Verify.getInt(0, 1)) {
      case 0:
        rbt.insert(/* select value */);
        break;
      case 1:
        rbt.remove(/* select value */);
        break;
      }
    }



  }
}
```

Sequence-based Generation

```
class RedBlackTreeDriver {
  static void main(String[] args) {

    // length of sequence (and method parameters)
    int lenOfSeq = ...;

    RedBlackTree rbt = new RedBlackTree();
    for (int i = 0; i < lenOfSeq; i++) {
      switch (Verify.getInt(0, 1)) {
      case 0:
        rbt.insert(Verify.getInt(0, lenOfSeq - 1));
        break;
      case 1:
        rbt.remove(Verify.getInt(0, lenOfSeq - 1));
        break;
      }
    }


  }
}
```

Sequence-based Generation

```
class RedBlackTreeDriver {
  static void main(String[] args) {
    int numOfNodes = ...;
    // length of sequence (and method parameters)
    int lenOfSeq = ...;

    RedBlackTree rbt = new RedBlackTree();
    for (int i = 0; i < lenOfSeq; i++) {
      switch (Verify.getInt(0, 1)) {
      case 0:
        rbt.insert(Verify.getInt(0, lenOfSeq - 1));
        break;
      case 1:
        rbt.remove(Verify.getInt(0, lenOfSeq - 1));
        break;
      }
    }
    if (rbt.size == numOfNodes) {
      printStructure(rbt);
    }
  }
}
```

Sequence-based Generation

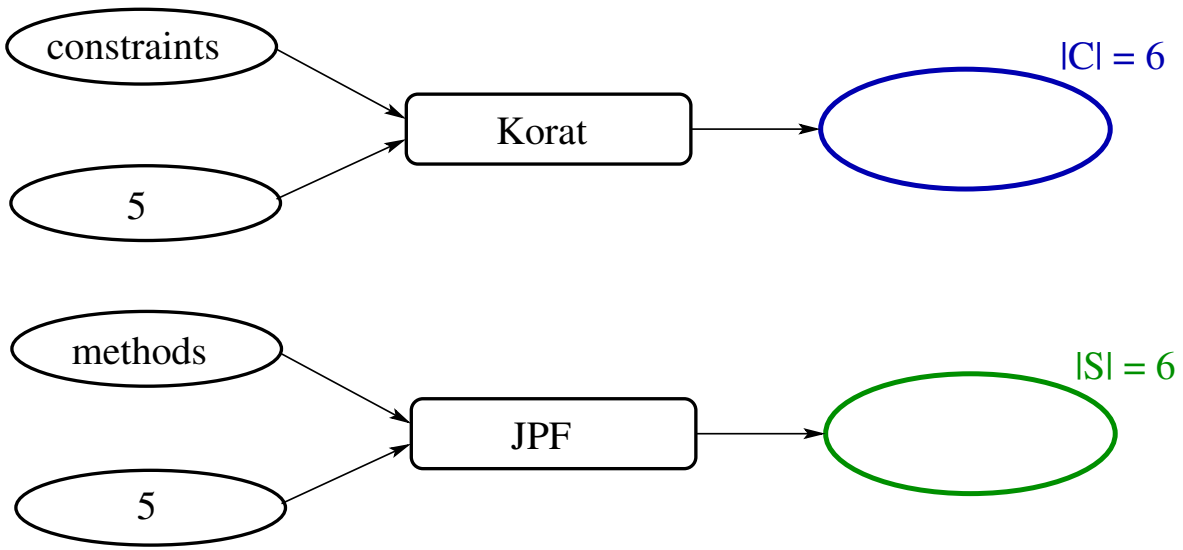
- Pros:
  - Generates only the states reachable by sequences of methods
  - Cannot generate false alarms
- Cons:
  - Requires testing several methods at once
  - Test the method that we use to generate the structure

Experiments for Comparison of Generated Structures

- Compare structures in C and S
- Five data structures with complex constraints
  - AVL Tree (put, remove)
  - HeapArray (insert, remove)
  - FibonacciHeap (insert, remove)
  - RedBlackTree (insert, remove)
  - DisjSet (find, union)
- Constraints (repOk) written "naturally"

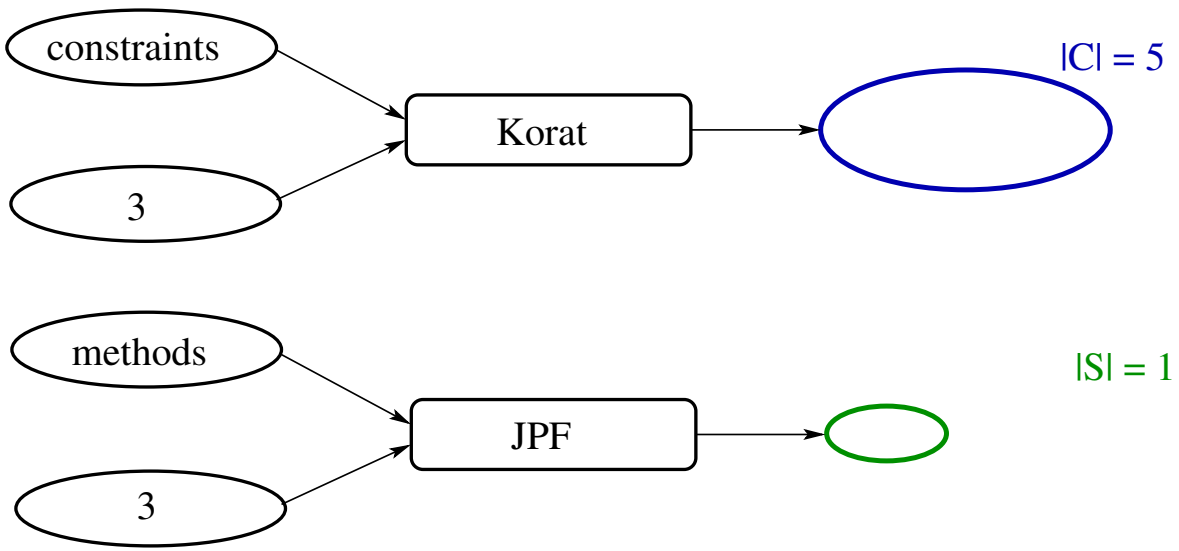
Case 1: Same Structures

- Same structures of size n with sequences of length n
- AVL Tree & HeapArray
- $C = S$



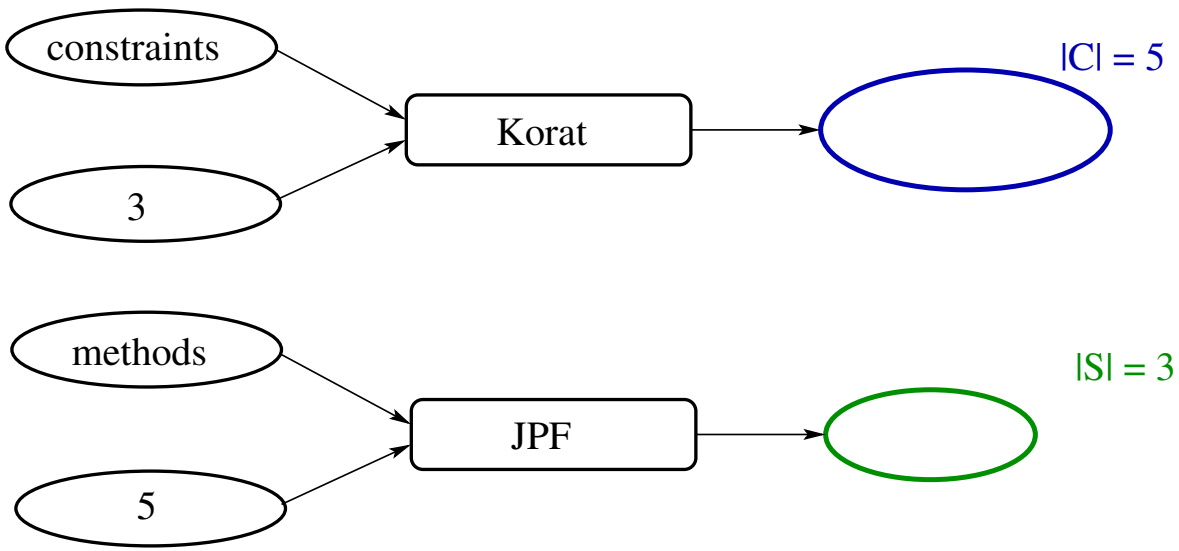
Case 2: Longer Sequences

- Same structures but longer sequences are required
- FibonacciHeap
- $C \supset S$



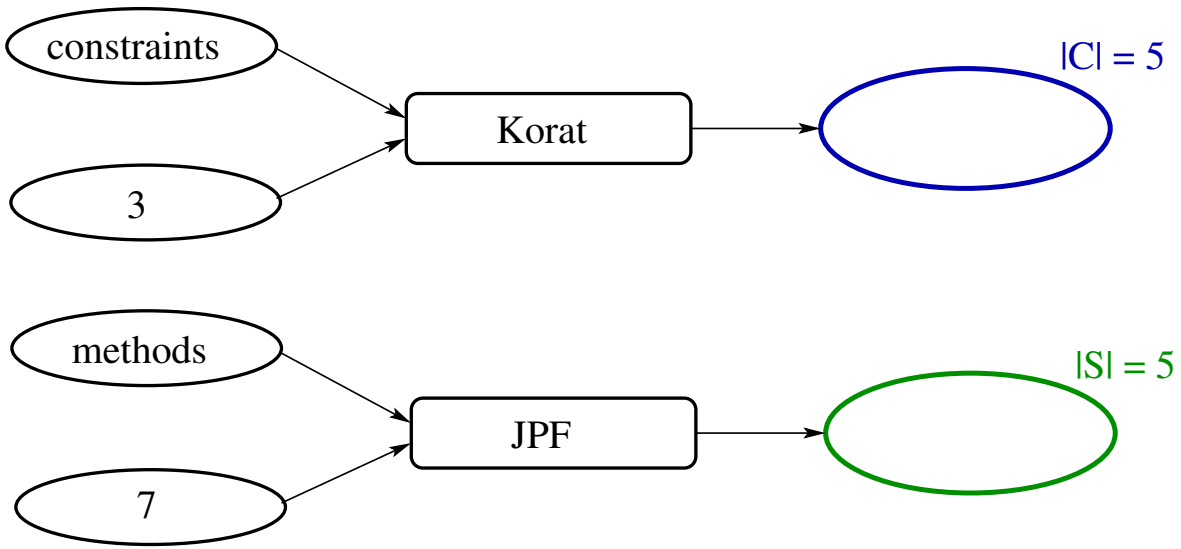
Case 2: Longer Sequences

- Same structures but longer sequences are required
- FibonacciHeap
- $C \supset S$



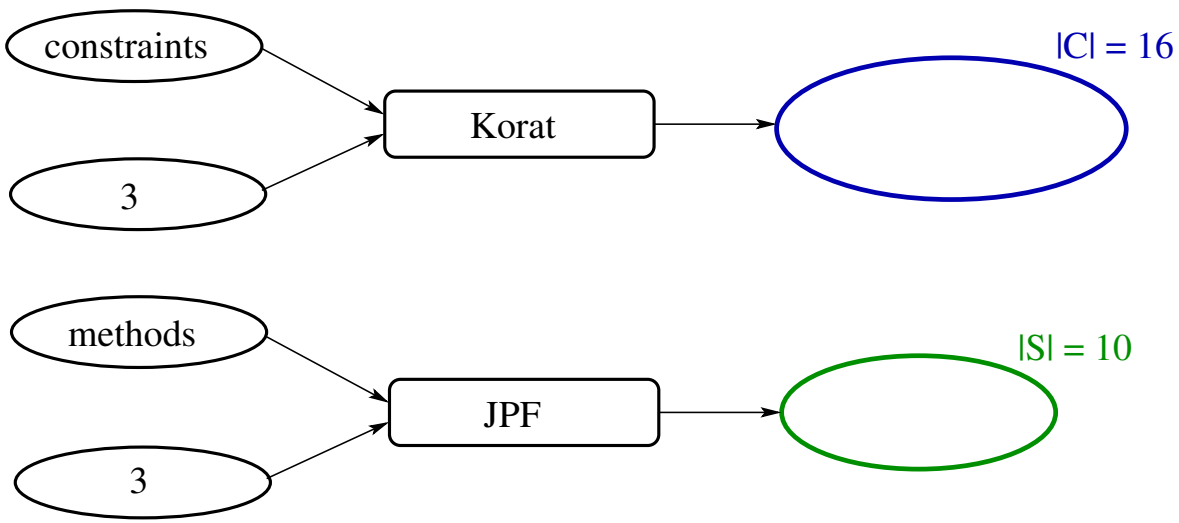
Case 2: Longer Sequences

- Same structures but longer sequences are required
- FibonacciHeap
- $C = S$



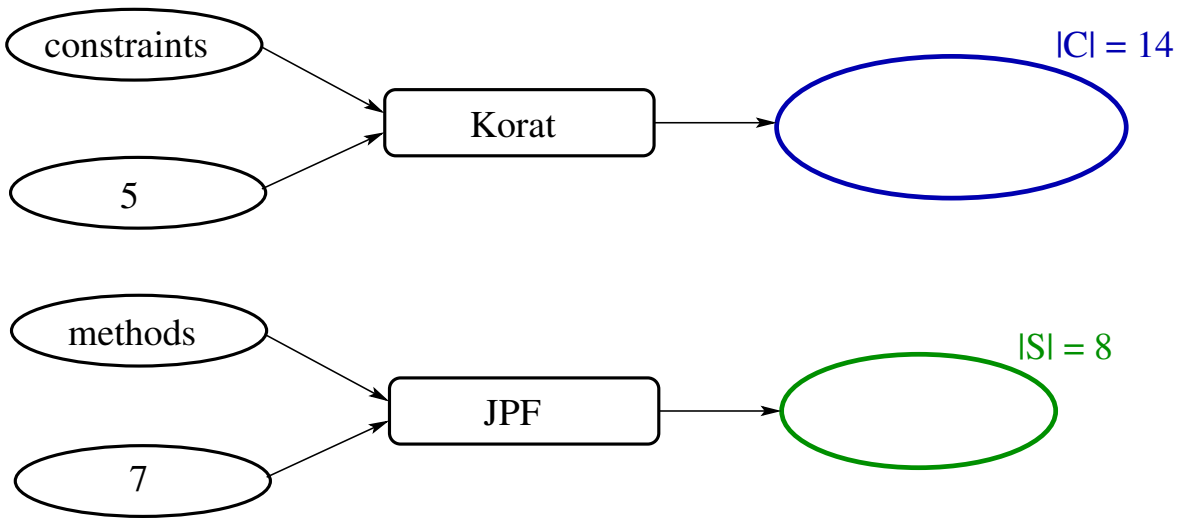
Case 3: More Structures, No False Alarms

- Constraint-based generates more structures but code under test works
- DisjSet
- $C \supset S$



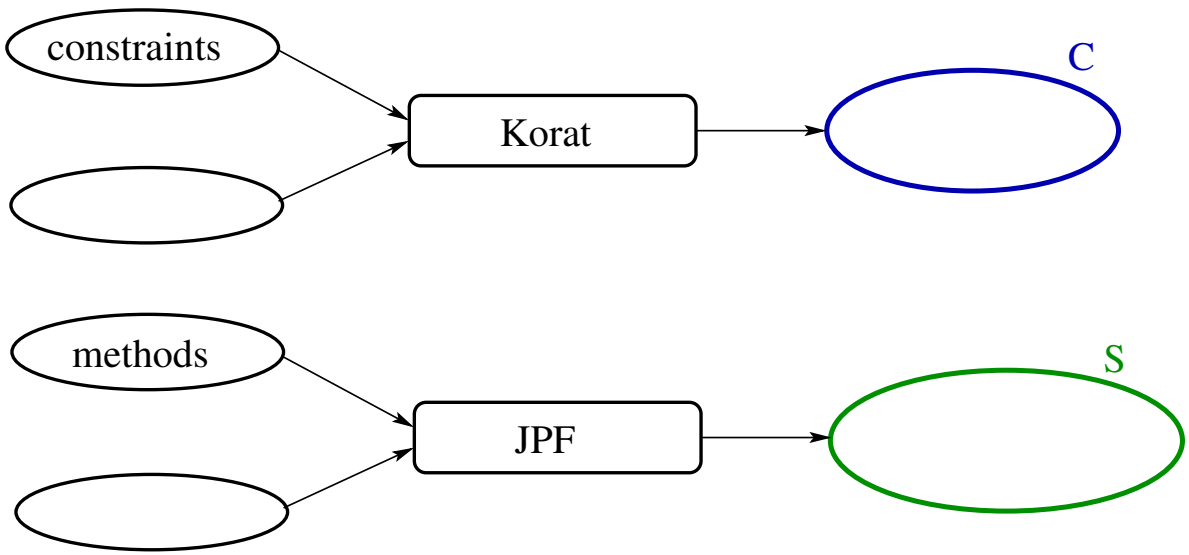
Case 4: More Structures, False Alarms

- Constraint-based generates more structures that lead to false alarm(s)
- RedBlackTree
- Slightly different implementation of the same data structure can behave differently in testing
- $C \supset S$



Case 5: Fewer Structures

- Constraint-based generates fewer structures, which could lead to missed bugs
- The case not encountered in our experiments
- $S - C \neq \{\}$



Conclusion

- Compared constraint-based and sequence-based generation
  - Five data structures with "natural" constraints
  - Four different cases
- Constraint-based generation is valuable for bounded-exhaustive generation but can result in false alarms
- Sequence-based generation does not require writing constraints but makes it hard to provide guarantees about bounded-exhaustive testing