Automated Test Generation

CS 6340

Outline

- Previously: Random testing (Fuzzing)
  - Security, mobile apps, concurrency

- Systematic testing: Korat
  - Linked data structures

- Feedback-directed random testing: Randoop
  - Classes, libraries

Korat

- A test-generation research project
- Idea
  - Leverage pre-conditions and post-conditions to generate tests automatically
- But how?

The Problem

- There are infinitely many tests
  - Which finite subset should we choose?

- And even finite subsets can be huge

- Need a subset which is:
  - concise: Avoids illegal and redundant tests
  - diverse: Gives good coverage

An Insight

- Often can do a good job by systematically testing all inputs up to a small size

- Small Test Case Hypothesis:
  - If there is any test that causes the program to fail, there is a small such test

- If a list function works for lists of length 0 through 3, probably works for all lists
  - E.g., because the function is oblivious to the length

How Do We Generate Test Inputs?

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

- Use the types
- The class declaration shows what values (or null) can fill each field
- Simply enumerate all possible shapes with a fixed set of Nodes

Scheme for Representing Shapes

- Order all possible values of each field
- Order all fields into a vector
- Each shape == vector of field values

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

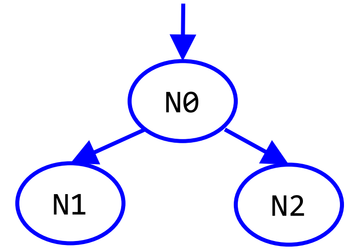
e.g.: BinaryTree of up to 3 Nodes:

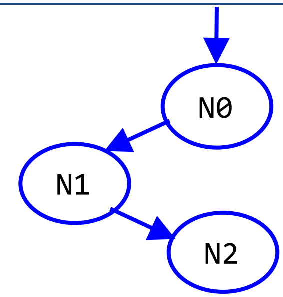
	N0		N1		N2	
root	left	right	left	right	left	right

QUIZ: Representing Shapes

Fill in the field values in each vector to represent the depicted shape:

	N0		N1		N2	
root	left	right	left	right	left	right

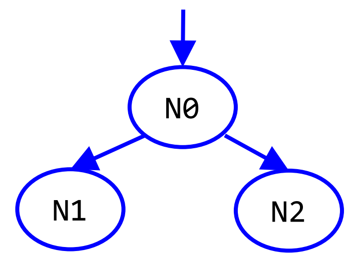


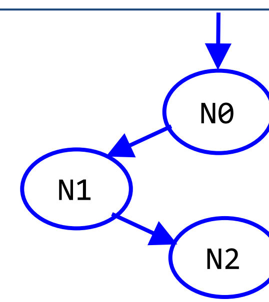


QUIZ: Representing Shapes

Fill in the field values in each vector to represent the depicted shape:

	N0		N1		N2	
root	left	right	left	right	left	right
N0	N1	N2	null	null	null	null
N0	N1	null	null	N2	null	null





A Simple Algorithm

- User selects maximum input size k
- Generate all possible inputs up to size k
- Discard inputs where pre-condition is false
- Run program on remaining inputs
- Check results using post-condition

QUIZ: Enumerating Shapes

Korat represents each input shape as a vector of the following form:

	N0		N1		N2	
root	left	right	left	right	left	right

What is the total number of vectors of the above form?

QUIZ: Enumerating Shapes

Korat represents each input shape as a vector of the following form:

	N0		N1		N2	
root	left	right	left	right	left	right

What is the total number of vectors of the above form?

16384

The General Case for Binary Trees

- How many binary trees are there of size <= k?

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

- Calculation:
  - A `BinaryTree` object, `bt`
  - k `Node` objects, `n0`, `n1`, `n2`, …
  - 2k+1 `Node` pointers
    - `root` (for `bt`)
    - `left`, `right` (for each `Node` object)
  - k+1 possible values (`n0`, `n1`, `n2`, … or `null`) per pointer

- $(k+1)^{(2k+1)}$ possible "binary trees"

A Lot of “Trees” !

- The number of “trees” explodes rapidly
  - $k = 3$: over 16,000 “trees”
  - $k = 4$: over 1,900,000 “trees”
  - $k = 5$: over 360,000,000 “trees”

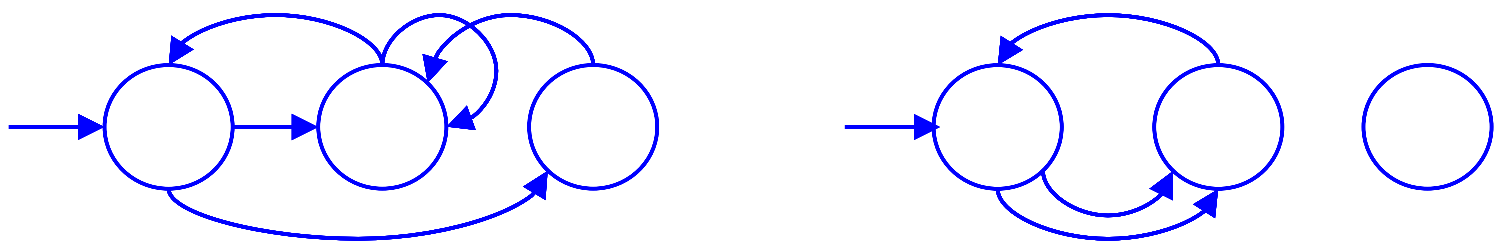
- Limits us to testing only very small input sizes

- Can we do better?

An Overestimate

- (k+1)^(2k+1) trees is a gross overestimate!

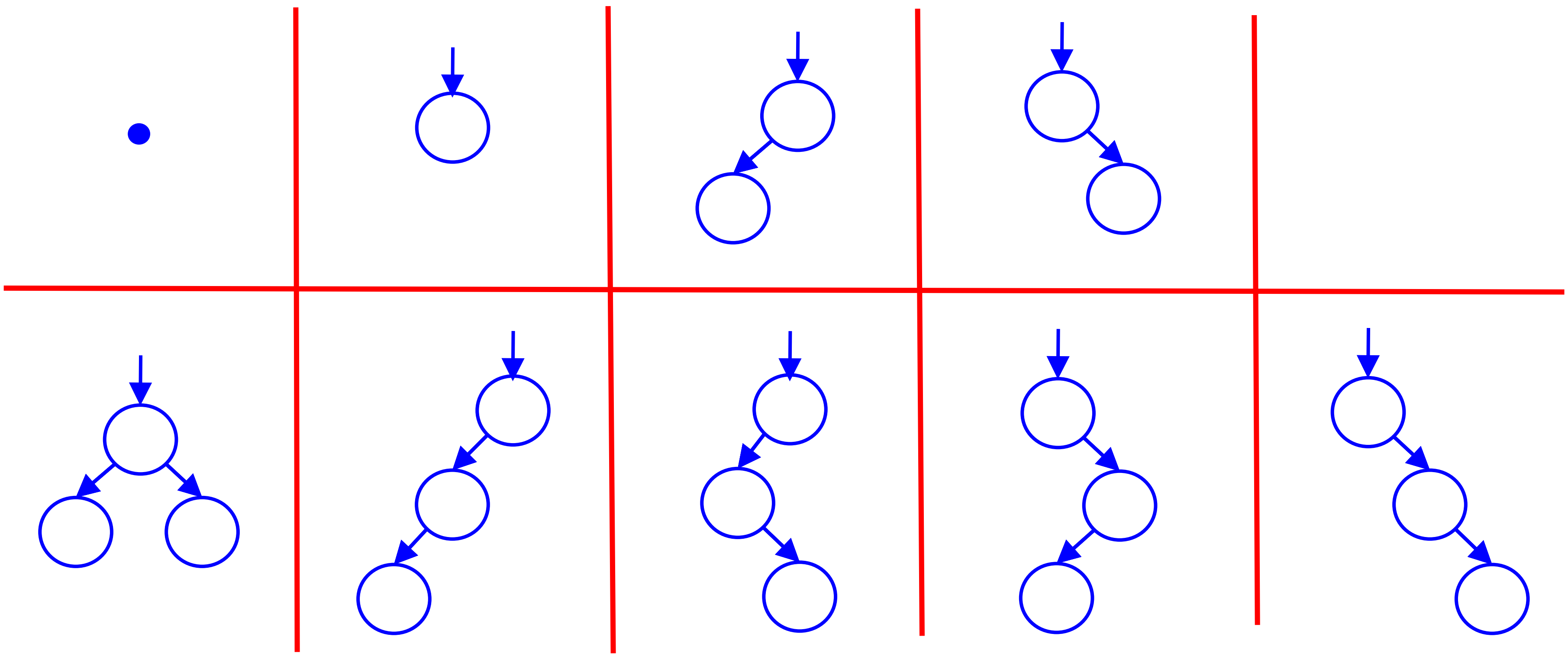
- Many of the shapes are not even trees:



- And many are isomorphic:

How Many Trees?

Only 9 distinct binary trees with at most 3 nodes



Another Insight

- Avoid generating inputs that don't satisfy the pre-condition in the first place
- Use the pre-condition to guide the generation of tests

The Technique

- Instrument the pre-condition
  - Add code to observe its actions
  - Record fields accessed by the pre-condition

- Observation:
  - If the pre-condition doesn't access a field, then pre-condition doesn't depend on the field.

The Pre-Condition for Binary Trees

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

- Root may be null
- If root is not null:
  - No cycles
  - Each node (except root) has one parent
  - Root has no parent

The Pre-Condition for Binary Trees

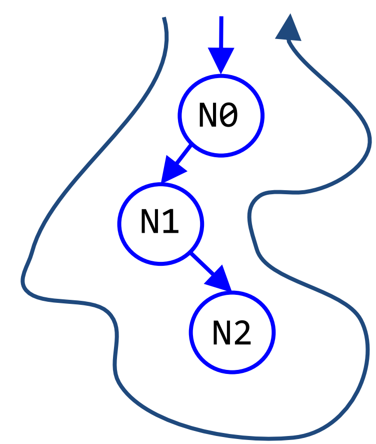
```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
     Node current = workList.removeFirst();
     if (current.left != null) {
        if (!visited.add(current.left)) return false;
        workList.add(current.left);
     }
     ... // similarly for current.right
   }
   return true;
}
```

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

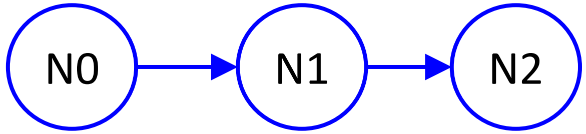
The Pre-Condition for Binary Trees

```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
     Node current = workList.removeFirst();
     if (current.left != null) {
        if (!visited.add(current.left)) return false;
        workList.add(current.left);
     }
     ... // similarly for current.right
   }
   return true;
}
```

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```



Example: Using the Pre-Condition

- Consider the following "tree": 

	N0		N1		N2	
root	left	right	left	right	left	right
null	null	N1	null	N2	null	null

- The pre-condition accesses only the root as it is null

  => Every possible shape for other nodes yields same result

  => This single input eliminates 25% of the tests!

Enumerating Tests

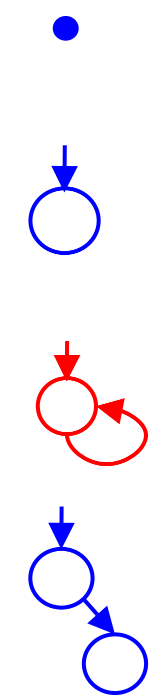
- Shapes are enumerated by their associated vectors
  - Initial candidate vector: all fields null
  - Next shape generated by:
    - Expanding last field accessed in pre-condition
    - Backtracking if all possibilities for a field are exhausted

- Key idea: Never expand parts of input not examined by pre-condition

- Also: Cleverly checks for and discards shapes isomorphic to previously-generated shapes

Example: Enumerating Binary Trees

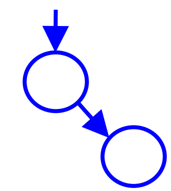
	N0		N1		N2			
root	left	right	left	right	left	right		
null	null	null	null	null	null	null	✓	
1								
N0	null	null	null	null	null	null	✓	
1	2	3						
N0	null	N0	null	null	null	null	✗	
N0	null	N1	null	null	null	null	✓	
1	2	3	4	5				



QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

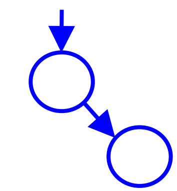
	N0		N1		N2	
root	left	right	left	right	left	right
N0	null	N1	null	null	null	null
1	**2**	**3**	**4**	**5**		

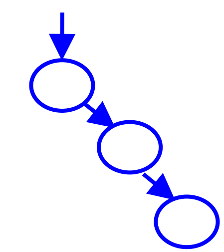


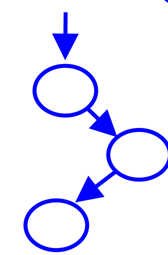
QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

	N0		N1		N2	
root	left	right	left	right	left	right
N0	null	N1	null	null	null	null
1	2	3	4	5		
N0	null	N1	null	N2	null	null
1	2	3	4	5	6	7
N0	null	N1	N2	null	null	null



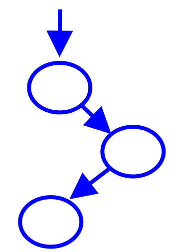




QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

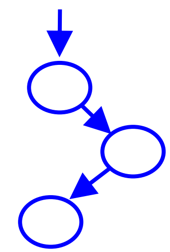
	N0		N1		N2	
root	left	right	left	right	left	right
N0	null	N1	N2	null	null	null

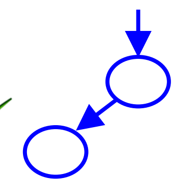


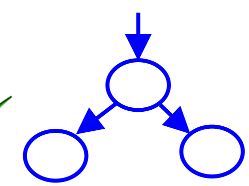
QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

	N0		N1		N2	
root	left	right	left	right	left	right
N0	null	N1	N2	null	null	null
1	2	3	4	7	5	6
N0	N1	null	null	null	null	null
1	2	5	3	4		
N0	N1	N2	null	null	null	null







Experimental Results

benchmark	size	time (sec)	structures generated	candidates considered	state space
BinaryTree	8	1.53	1430	54418	2^{53}
	9	3.97	4862	210444	2^{63}
	10	14.41	16796	815100	2^{72}
	11	56.21	58786	3162018	2^{82}
	12	233.59	208012	12284830	2^{92}
HeapArray	6	1.21	13139	64533	2^{20}
	7	5.21	117562	519968	2^{25}
	8	42.61	1005075	5231385	2^{29}
LinkedList	8	1.32	4140	5455	2^{91}
	9	3.58	21147	26635	2^{105}
	10	16.73	115975	142646	2^{120}
	11	101.75	678570	821255	2^{135}
	12	690.00	4213597	5034894	2^{150}
TreeMap	7	8.81	35	256763	2^{92}
	8	90.93	64	2479398	2^{111}
	9	2148.50	122	50209400	2^{130}

Strengths and Weaknesses

- Strong when we can enumerate all possibilities
  - e.g. Four nodes, two edges per node

  => Good for:
    - Linked data structures
    - Small, easily specified procedures
    - Unit testing
- Weaker when enumeration is weak
  - Integers, Floating-point numbers, Strings

Weaknesses

Only as good as the pre- and post-conditions

```
Pre: is_member(x, list)
List remove(Element x, List list) {
   if (x == head(list))
       return tail(list);
   else
       return cons(head(list),
                   remove(x, tail(list)));
}
Post: !is_member(x, list')
```

Weaknesses

Only as good as the pre- and post-conditions

```
Pre: !is_empty(list)
List remove(Element x, List list) {
   if (x == head(list))
      return tail(list);
   else
      return cons(head(list),
                  remove(x, tail(list)));
}
Post: is_list(list')
```

Feedback-Directed Random Testing

How do we generate a test like this?

```
public static void test() {

  LinkedList l1 = new LinkedList();
  Object o1 = new Object();
  l1.addFirst(o1);
  TreeSet t1 = new TreeSet(l1);
  Set s1 = Collections.unmodifiableSet(t1);

  // This assertion fails
  assert(s1.equals(s1));
 }
```

Overview

Problem with uniform random testing: Creates too many illegal or redundant tests

Idea: Randomly create new test guided by feedback from previously created tests

test == method sequence

Recipe:

- Build new sequences incrementally, extending past sequences
- As soon as a sequence is created, execute it
- Use execution results to guide test generation towards sequences that create new object states

Randoop: Input and Output

Input:

- classes under test
- time limit
- set of contracts

  e.g. “o.hashCode() throws
  no exception”

  e.g. “o.equals(o) == true”

Output:

- contract-violating test cases

```
LinkedList l1 = new LinkedList();
Object o1 = new Object();
l1.addFirst(o1);
TreeSet t1 = new TreeSet(l1);
Set s1 = Collections.unmodifiableSet(t1);

assert(s1.equals(s1));
```

No contract violated up to here

fails when executed

Randoop Algorithm

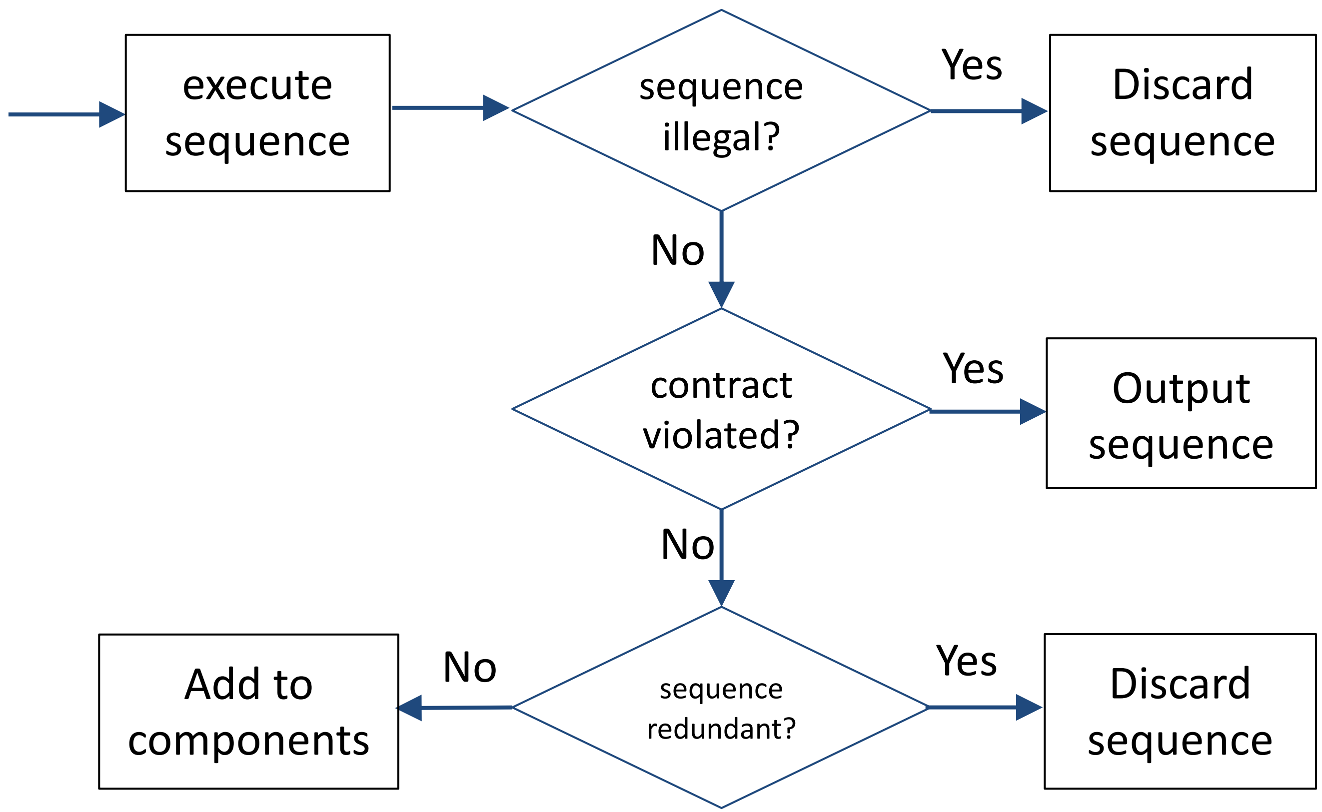
components = { $\boxed{\text{int i = 0;}}$, $\boxed{\text{boolean b = false;}}$ . . . }
// seed components

Repeat until time limit expires:

- Create a new sequence
  - Randomly pick a method call $\texttt{T}_{\texttt{ret}}$ `m(T1,...,Tn)`
  - For each argument of type `Ti`, randomly pick sequence `Si` from components that constructs an object `vi` of that type
  - Create S_{new} = `S1; ...; Sn; Tret vnew = m(v1...vn);`

- Classify new sequence S_{new}: discard / output as test / add to components

Classifying a Sequence



Illegal Sequences

- Sequences that "crash" before contract is checked
  - E.g. throw an exception

```
int i = -1;
Date d = new Date(2006, 2, 14);
d.setMonth(i);    // pre: argument >= 0
assert(d.equals(d));
```

Redundant Sequences

- Maintain set of all objects created in execution of each sequence
- New sequence is redundant if each object created during its execution belongs to above set (using `equals` to compare)
- Could also use more sophisticated state equivalence methods

```
Set s = new HashSet();
s.add("hi");


assertTrue(s.equals(s));
```

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();

assertTrue(s.equals(s));
```

Some Errors Found by Randoop

- JDK containers have 4 methods that violate `o.equals(o)` contract
- Javax.xml creates objects that cause `hashCode` and `toString` to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of `equals`, `hashCode`, and `toString`
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, or illegal state exception)
- .Net framework has 8 methods that violate `o.equals(o)` contract

QUIZ: Randoop Test Generation (Part 1)

Write the smallest sequence that Randoop can possibly generate to create a valid BinaryTree.

Once generated, how does Randoop classify it?

- ◯ Discards it as illegal
- ◯ Outputs it as a bug
- ◯ Adds to components for future extension

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Randoop Test Generation (Part 1)

Write the smallest sequence that Randoop can possibly generate to create a valid BinaryTree.

```
BinaryTree bt = new BinaryTree(null);
```

Once generated, how does Randoop classify it?

- ○ Discards it as illegal
- ○ Outputs it as a bug
- ● Adds to components for future extension

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Randoop Test Generation (Part 2)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in removeRoot().

Once generated, how does Randoop classify it?

- ◯ Discards it as illegal
- ◯ Outputs it as a bug
- ◯ Adds to components for future extension

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Randoop Test Generation (Part 2)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in removeRoot().

```
BinaryTree bt = new BinaryTree(null);
bt.removeRoot();
```

Once generated, how does Randoop classify it?

- ● Discards it as illegal
- ○ Outputs it as a bug
- ○ Adds to components for future extension

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Randoop Test Generation (Part 3)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in BinaryTree's constructor.

Can Randoop create a BinaryTree object with cycles using the given API?

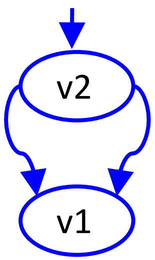
- ◯ Yes
- ◯ No

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Randoop Test Generation (Part 3)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in BinaryTree's constructor.




```
Node v1 = new Node(null, null);
Node v2 = new Node(v1, v1);
BinaryTree bt = new BinaryTree(v2);
```

Can Randoop create a BinaryTree object with cycles using the given API?

- ○ Yes
- ● No

```
class BinaryTree {
  Node root;
  public BinaryTree(Node r) {
     root = r;
     assert(repOk(this));
  }
  public Node removeRoot() {
     assert(root != null);
     ...
  }
}
```

```
class Node {
  Node left;
  Node right;
  public Node(Node l, Node r)
{
     left = l;
     right = r;
  }
}
```

QUIZ: Korat and Randoop

Identify which statements are true for each test generation technique:

	Korat	Randoop
Uses type information to guide test generation.	☐	☐
Each test is generated fully independently of past tests.	☐	☐
Generates tests deterministically.	☐	☐
Suited to test method sequences.	☐	☐
Avoids generating redundant tests.	☐	☐

QUIZ: Korat and Randoop

Identify which statements are true for each test generation technique:

	Korat	Randoop
Uses type information to guide test generation.	☑	☑
Each test is generated fully independently of past tests.	☐	☐
Generates tests deterministically.	☑	☐
Suited to test method sequences.	☐	☑
Avoids generating redundant tests.	☑	☑

Test Generation: The Bigger Picture

- Why didn't automatic test generation become popular decades ago?

- Belief: Weak-type systems
  - Test generation relies heavily on type information
  - C, Lisp just didn't provide the needed types

- Contemporary languages lend themselves better to test generation
  - Java, UML

What Have We Learned?

- Automatic test generation is a good idea
  - Key: avoid generating illegal and redundant tests
- Even better, it is possible to do
  - At least for unit tests in strongly-typed languages
- Being adopted in industry
  - Likely to become widespread