

# Lecture 19: Unit Test Generation

CS 5150, Spring 2025

# Administrative Reminders

- Project Report #3 due on April 10
- Project Report #4 due on April 25
  - Focus on Testing and Integration!
- Assignment A4 coming soon!

# Lecture Goals

- Understand unit-test generation techniques
- Learn about coverage and mutation testing techniques

# Unit Test Generation

# Outline

- Previously: Random testing (Fuzzing)
  - Security, mobile apps, ...
- Feedback-directed random testing: Randoop
  - Classes and libraries
- Systematic testing: Korat
  - Linked data structures

# Leveraging the Specifications

# Automated Test Generation: Key Idea

Leverage the specifications to guide test generation:

- Types
- Invariants
- Pre- and Post- Conditions

# Example: Leveraging Types

```
void remove(BinaryTree bt, Node n) {  
    ... // remove node n from binary tree bt  
}
```

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

- Helps to avoid testing the `remove` method on arbitrary byte arrays



# Example: Leveraging Invariants

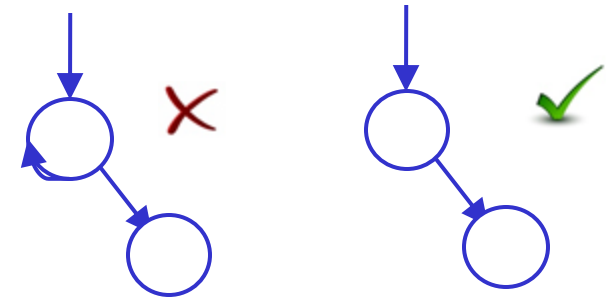
- Root may be null
- If root is not null:
  - No cycles
  - Each node (except root) has one parent
  - Root has no parent

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

# Example: Leveraging Invariants

```
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: Leveraging Invariants

```
@invariant repOk(bt)
```

```
void remove(BinaryTree bt, Node n) {  
    ... // remove node n from binary tree bt  
}
```

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

- Helps to avoid testing the `remove` method on non-tree structures
- Also serves as a contract to check at the end of `remove` method

# Example: Leveraging Pre- and Post-Conditions

```
@invariant repOk(bt)
@requires contains(bt, n) // pre condition
@ensures !contains(bt, n) // post condition

void remove(BinaryTree bt, Node n) {
    ... // remove node n from binary tree bt
}
```

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

- Helps to test even richer states on entry and exit of `remove` method

# Testing Classes and Libraries

Key Ideas of Randoop

# Randoop: 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));  
}
```

public TreeSet(Collection c): Constructs a new, empty tree set, sorted according to the specified comparator. **All elements inserted into the set must be *mutually comparable*** by the specified comparator: comparator.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the set. **If the user attempts to add an element to the set that violates this constraint, the add call will throw a ClassCastException.** <https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>

# 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



# SEGMENT

## The Randoop Algorithm

# Randoop Algorithm

components = { `int i = 0;` `boolean b = false;` ... } // seed components

Repeat until **time limit expires**:

- Create a new sequence
  - Randomly pick a method call  $T_{\text{ret}} \ m(T_1, \dots, T_n)$
  - For each argument of type  $T_i$ , randomly pick sequence  $S_i$  from components that constructs an object  $v_i$  of that type
  - Create  $S_{\text{new}} = S_1; \dots; S_n; T_{\text{ret}} \ v_{\text{new}} = m(v_1, \dots, v_n);$
- Classify new sequence  $S_{\text{new}}$ : discard / output as test / add to components

- - - ➔ Method
- - - ➔ Parameter
- - - ➔ Receiver object

# Randoop: example

## Program under test:

```
public class A{  
    public A() {...}  
    public B m1(A a1) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

## Test1:

```
B b1=new B(0);
```

## Components:

```
S1: B b1=new B(0);
```

```
{0} 1, null, "hi", ...}
```

- - - → Method
- - - → Parameter
- - - → Receiver object

# Randoop: example

## Program under test:

```
public class A{  
    public A() { ... }  
    public B m1(A a1) { ... }  
}  
public class B{  
    public B(int i) { ... }  
    public void m2(B b, A a) { ... }  
}
```

## Test1:

```
B b1=new B(0);
```

## Test2:

```
A a1=new A();
```

## Components:

```
S2: A a1=new A();
```

```
S1: B b1=new B(0);
```

```
{0, 1, null, "hi", ...}
```

- - - → Method
- - - → Parameter
- - - → Receiver object

# Randoop: example

## Program under test:

```
public class A{  
    public A() {...}  
    public B m1(A a1) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

## Test1:

```
B b1=new B(0);
```

## Test2:

```
A a1=new A();
```

## Test3:

```
A a1=new A(); //reused from s2  
B b2=a1.m1(a1);
```

## Components:

```
S3: A a1=new A();  
    B b2=a1.m1(a1);
```

```
S2: A a1=new A();
```

```
S1: B b1=new B(0);
```

```
{0, 1, null, "hi", ...}
```

- - - → Method
- - - → Parameter
- - - → Receiver object

# Randoop: example

## Program under test:

```
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

## Components:

S3: A a1=new A();  
B b2=a1.m1(a1);

S2: A a1=new A();

S1: B b1=new B(0);

{0, 1, null, "hi", ...}

S4: ...

## Test1:

B b1=new B(0);

## Test2:

A a1=new A();

## Test3:

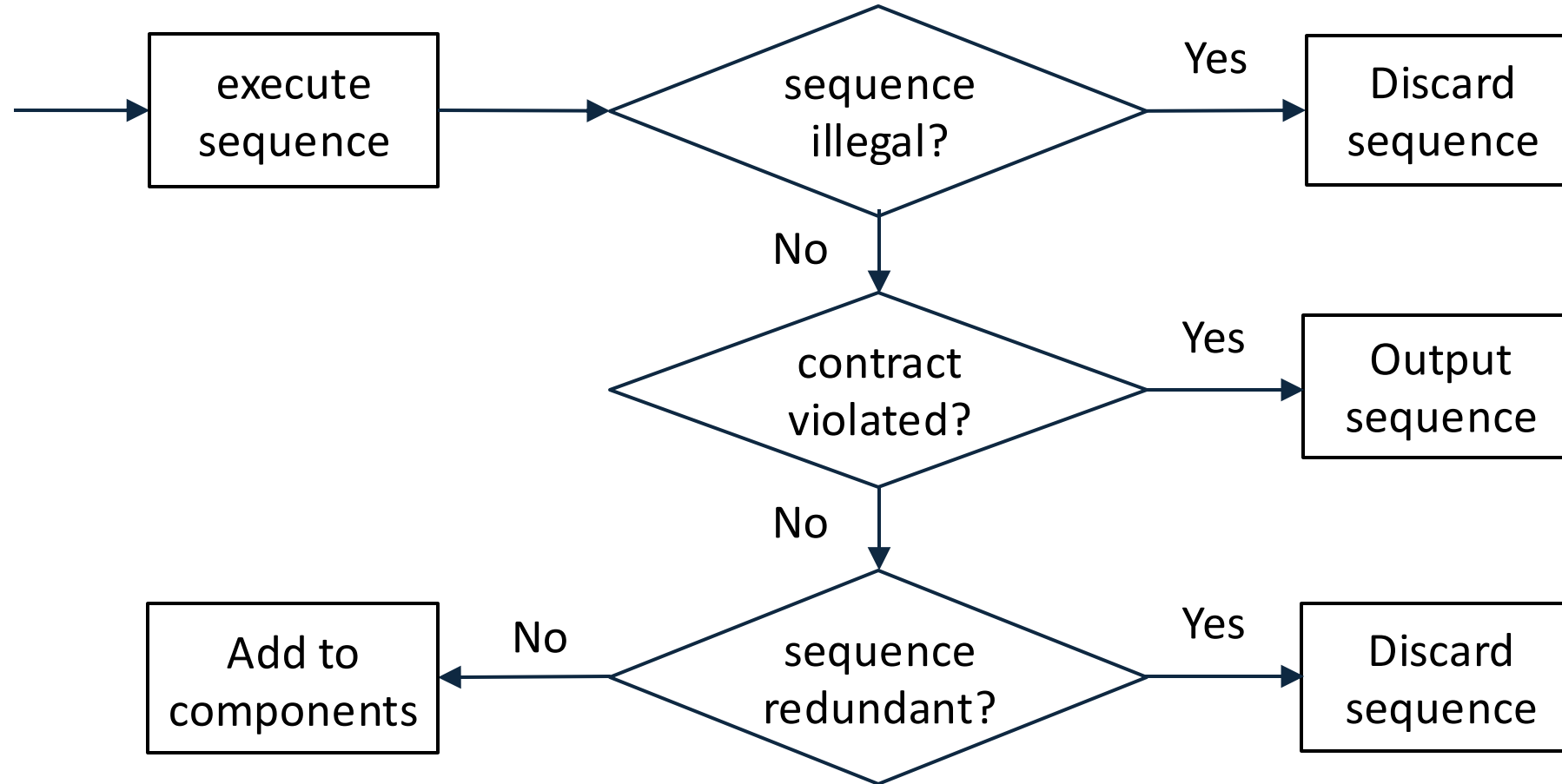
A a1=new A();  
B b2=a1.m1(a1);

## Test4:

B b1=new B(0); //reused from s1  
A a1=new A();  
B b2=a1.m1(a1); //reused from s3  
b1.m2(b2, a1);

...

# 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));
```

# SEGMENT

## Randoop in Practice

# Code coverage by Randoop

Data structure programs	Time (s)	Branch cov.
Bounded stack (30 LOC)	1	100%
Unbounded stack (59 LOC)	1	100%
BS Tree (91 LOC)	1	96%
Binomial heap (309 LOC)	1	84%
Linked list (253 LOC)	1	100%
Tree map (370 LOC)	1	81%
Heap array (71 LOC)	1	100%

# Bug detection by Randoop: subjects

Subjects	LOC	Classes
JDK (2 libraries) (java.util, javax.xml)	53K	272
Apache commons (6 libraries) (logging, primitives, chain, jelly, math, collections)	114K	974
.Net libraries (6 libraries)	615K	3455

# Bug detection by Randoop: subjects

<b>Subjects</b>	<b>Failed tests</b>	<b>Unique failed tests</b>	<b>Error-revealing tests</b>	<b>Distinct errors</b>
JDK	613	32	29	8
Apache commons	3,044	187	29	6
.Net framework	543	205	196	196
Total	4,200	424	254	210

# Some Bugs 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;
    }
}
```

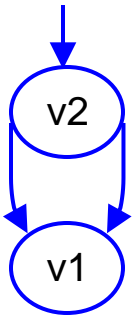
# Example: Leveraging Invariants

- Root may be null
- If root is not null:
  - No cycles
  - Each node (except root) has one parent
  - Root has no parent
- RepOk method checks if the binary tree is valid

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

# 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

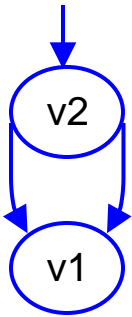
☐ No

```
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: 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;
    }
}
```

# How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

# Code Coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., (provably) unreachable code
  - Often required for safety-critical applications



# Types of Code Coverage

- **Function coverage:** which **functions** were called?
- **Statement coverage:** which **statements** were executed?
- **Branch coverage:** which **branches** were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, ...

# QUIZ: Code Coverage Metrics

Test Suite: { foo(1, 0) }

Statement Coverage:  %

Branch Coverage:  %

Give arguments for another call to foo(x, y) to add to the test suite to increase both coverages to 100%.

x =       y =

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

# QUIZ: Code Coverage Metrics

Test Suite: { foo(1, 0) }

Statement Coverage:  %

Branch Coverage:  %

Give arguments for another call to foo(x, y) to add to the test suite to increase both coverages to 100%.

x =       y =

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

# Mutation Testing/Analysis

- Founded on “competent programmer assumption”:  
*The program is close to correct to begin with*
- Key idea: Test variations (mutants) of the program
  - Replace  $x > 0$  by  $x < 0$
  - Replace  $w$  by  $w + 1$ ,  $w - 1$
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

# A Problem

- What if a **mutant** is equivalent to the **original**?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
  - Try to prove **program equivalence** automatically (undecidable)
  - Often requires manual intervention

# QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input type="checkbox"/>	<input type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Is the test suite adequate with respect to both mutants?



Yes



No

# QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Is the test suite adequate with respect to both mutants?



Yes



No

# QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:  
foo(, ) ==



# QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 $x \leq y \rightarrow x > y$	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Mutant 2 $x \leq y \rightarrow x \neq y$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

```
int foo(int x, int y) {  
    int z = 0;  
    if (x <= y) {  
        z = x;  
    } else {  
        z = y;  
    }  
    return z;  
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:  
foo(, ) ==



# LESSON

## Testing Data Structures

# SEGMENT

Key Ideas of Korat

# Korat

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

# 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 smaller 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?

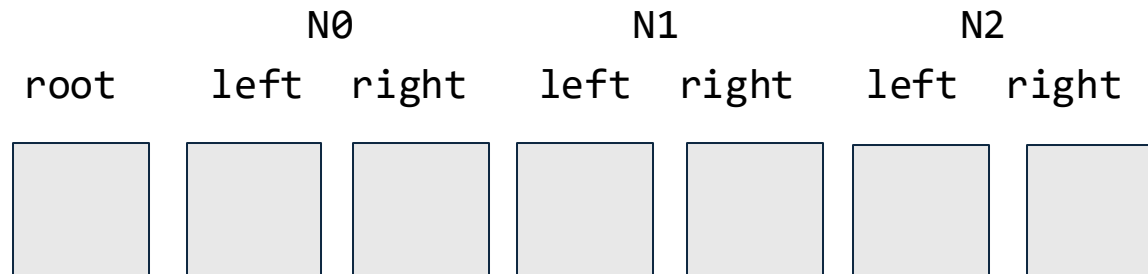
- Use the **types**
- The class declaration shows what values (or null) can fill each field
- Simply enumerate all possible shapes with a fixed set of **Nodes**

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

# Scheme for Representing Shapes

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

**Example:** BinaryTree of up to 3 Nodes:

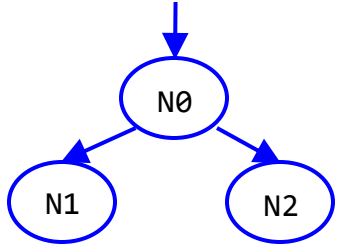
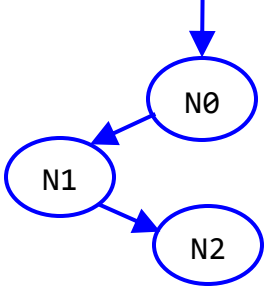


```
class BinaryTree {  
    Node root;  
    class Node {  
        Node left;  
        Node right;  
    }  
}
```




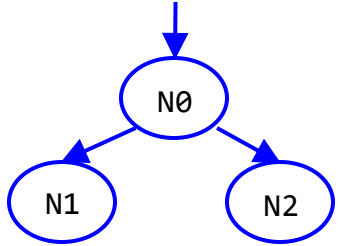
# QUIZ: Representing Shapes

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

	N0		N1		N2		
	root	left	right	left	right	left	right
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
							
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
							

# QUIZ: Representing Shapes

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

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

# SEGMENT

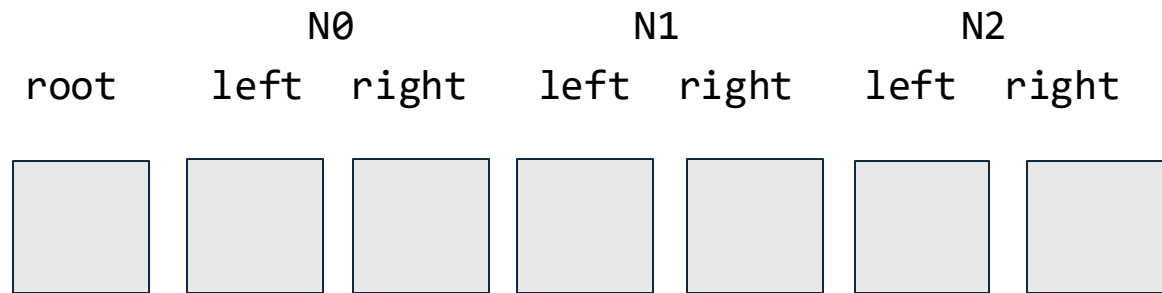
## A Simple Algorithm

# A Simple Algorithm

- User selects some 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

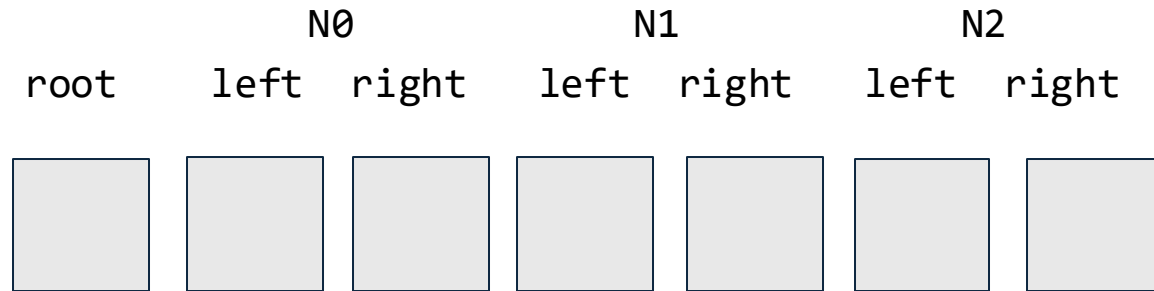
Korat represents each input shape as a vector of the following form:



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:



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  $\leq k$ ?
- Calculation:
  - A BinaryTree object, bt
  - $k$  Node objects,  $n_0, n_1, n_2, \dots$
  - $2k+1$  Node pointers
    - root (for bt)
    - left, right (for each Node object)
    - $k+1$  possible values ( $n_0, n_1, n_2, \dots$  or null) per pointer
- $(k+1)^{(2k+1)}$  possible “binary trees”

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

# A Lot of “Trees”!

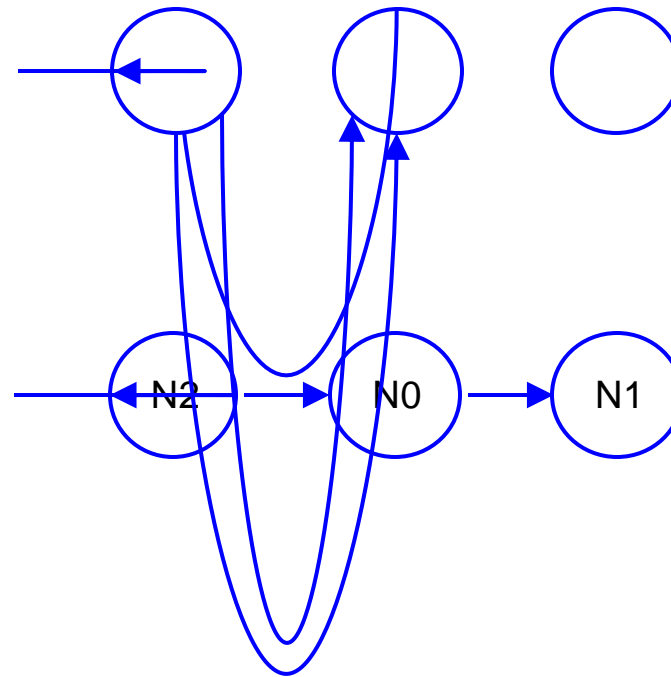
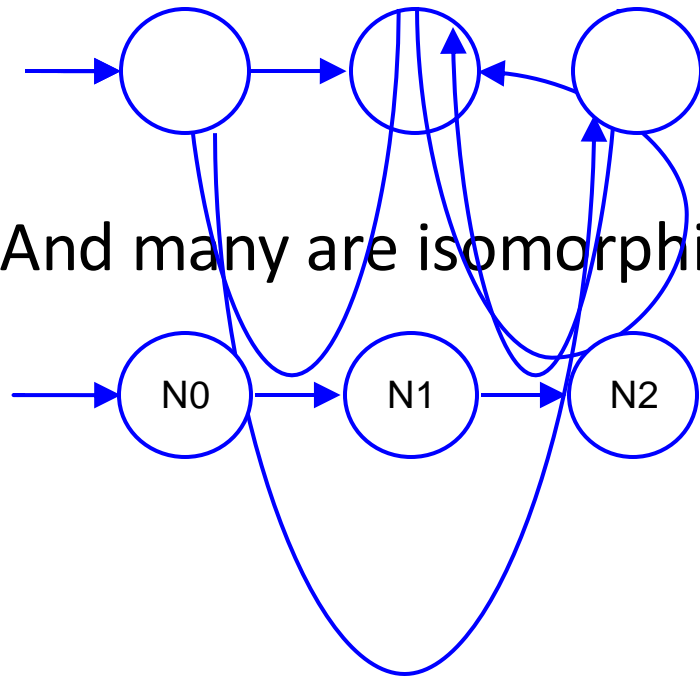
- The number of “trees” explodes rapidly
  - $k = 3$ : over 16,000 “trees”
  - $k = 4$ : over 1,900,000 “trees”
  - $k = 5$ : over 360,000,000 “trees”
- Limits us to testing only very small input sizes
- Can we do better?



# An Overestimate

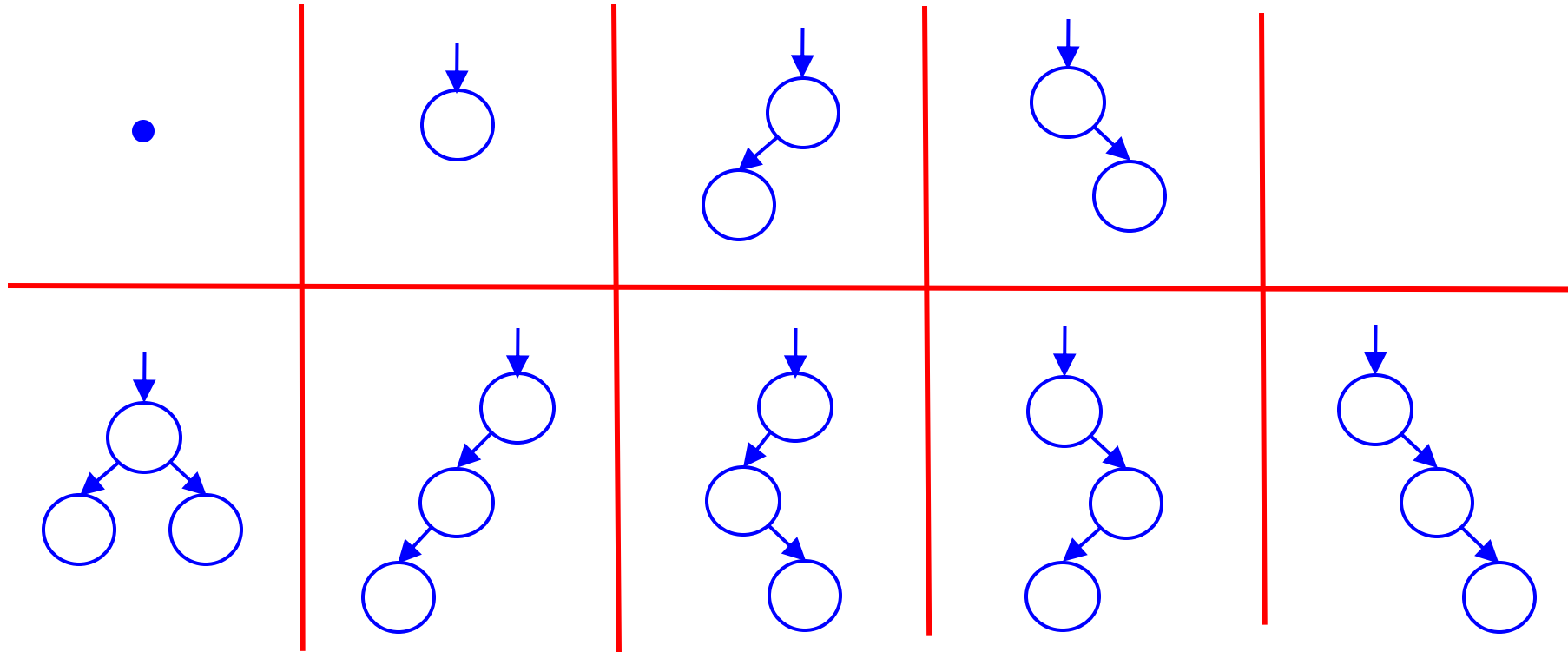
- $(k+1)^{(2k+1)}$  trees is a gross overestimate!
- Many of the shapes are not even trees:

- And many are isomorphic:



# How Many Trees?

There are only 9 distinct binary trees with at most 3 nodes:



# SEGMENT

Using the Invariant

# Another Insight

- Avoid generating inputs that don't satisfy the **invariant** in the first place
- Leverage the **invariant** to guide the generation of tests

# The Technique

- Instrument the **invariant**
  - Add code to record fields accessed by the **invariant**
- Observation:
  - If the **invariant** doesn't access a field, then it doesn't depend on the field

# The Invariant for Binary Trees

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

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

# The Invariant for Binary Trees

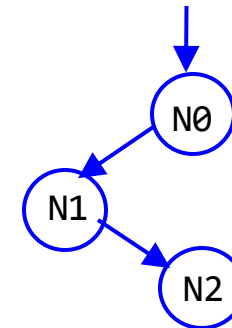
```
public boolean repOK(BinaryTree bt) {  
    if (bt.root == null) return true;  
    Set visited = new HashSet();  
    List workList = new LinkedList();  
    visited.add(bt.root);  
    workList.add(bt.root);  
    while (!workList.isEmpty()) {  
        Node current = workList.removeFirst();  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

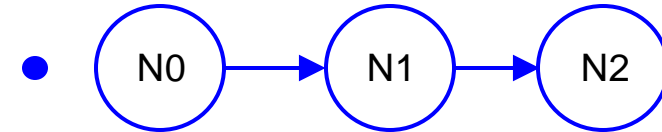
```
class BinaryTree {  
    Node root;  
    class Node {  
        Node left;  
        Node right;  
    }  
}
```





# Example: Using the Invariant

- Consider the following “tree”:



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

- The **invariant** 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!

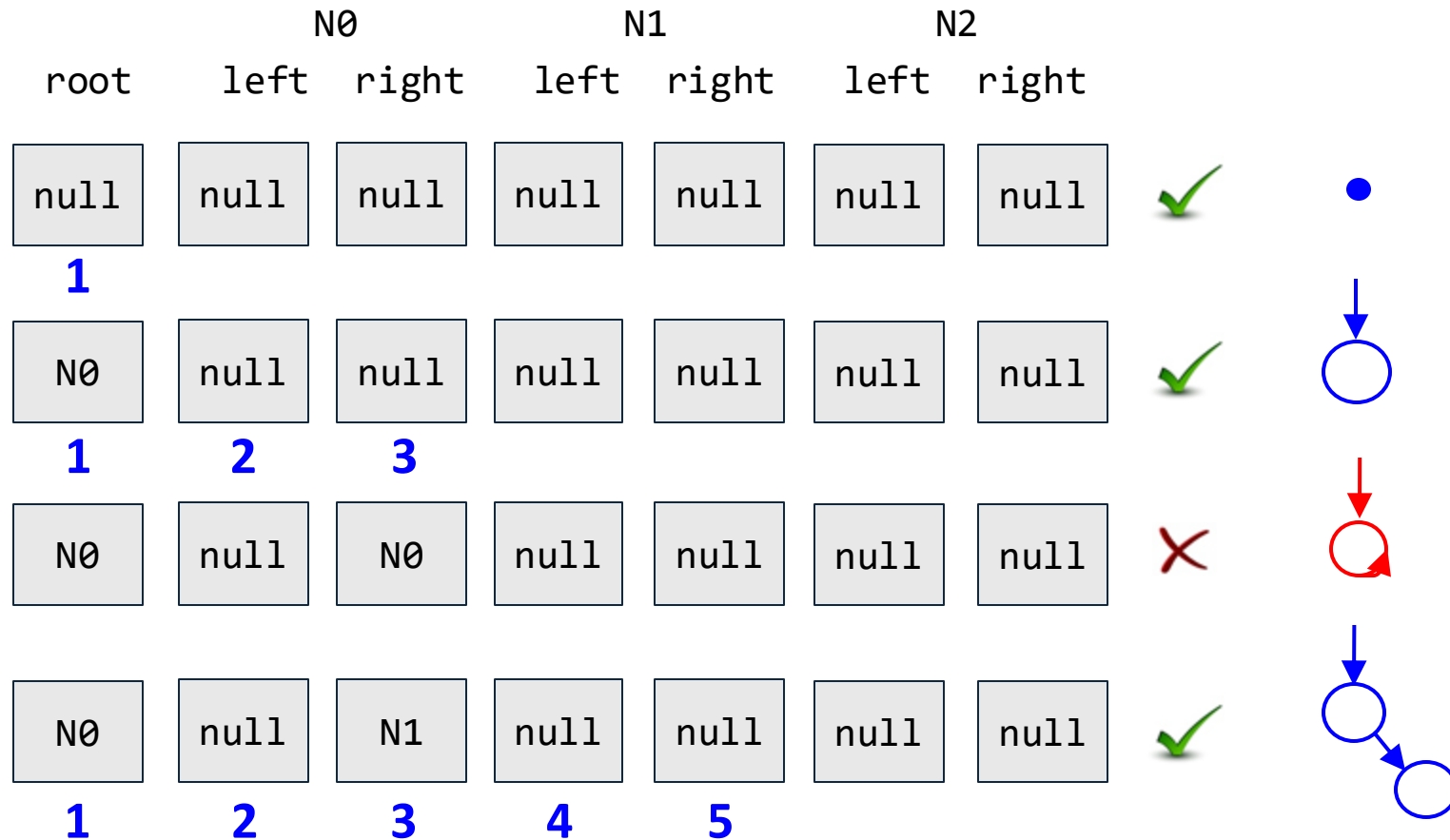
# SEGMENT

## Enumerating Tests

# Enumerating Tests

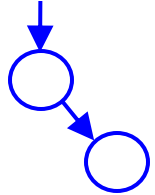
- Shapes are **enumerated** according to their associated vectors
  - Initial candidate vector: all fields null
  - Next shape generated by:
    - **Expanding** last field accessed in invariant
    - **Backtracking** if all possibilities for a field are exhausted
- **Key idea:** Never expand fields not examined by **invariant**
- Also: Cleverly checks for and discards shapes **isomorphic** to previously generated shapes

# Example: Enumerating Binary Trees



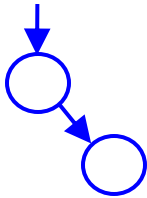
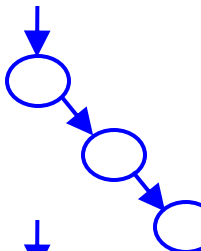
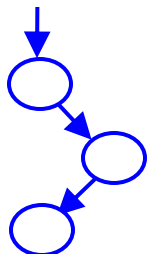
# QUIZ: Enumerating Binary Trees

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

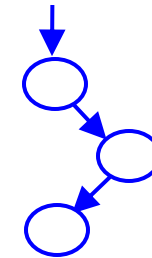
What are the next two legal, non-isomorphic shapes Korat generates?

		N0		N1		N2		
root	left	right	left	right	left	right		
<div>N0</div>	<div>null</div>	<div>N1</div>	<div>null</div>	<div>null</div>	<div>null</div>	<div>null</div>	<div>✓</div>	<div></div>
1	2	3	4	5				
<div>N0</div>	<div>null</div>	<div>N1</div>	<div>null</div>	<div>N2</div>	<div>null</div>	<div>null</div>	<div>✓</div>	<div></div>
1	2	3	4	5	6	7		
<div>N0</div>	<div>null</div>	<div>N1</div>	<div>N2</div>	<div>null</div>	<div>null</div>	<div>null</div>	<div>✓</div>	<div></div>

# QUIZ: Enumerating Binary Trees

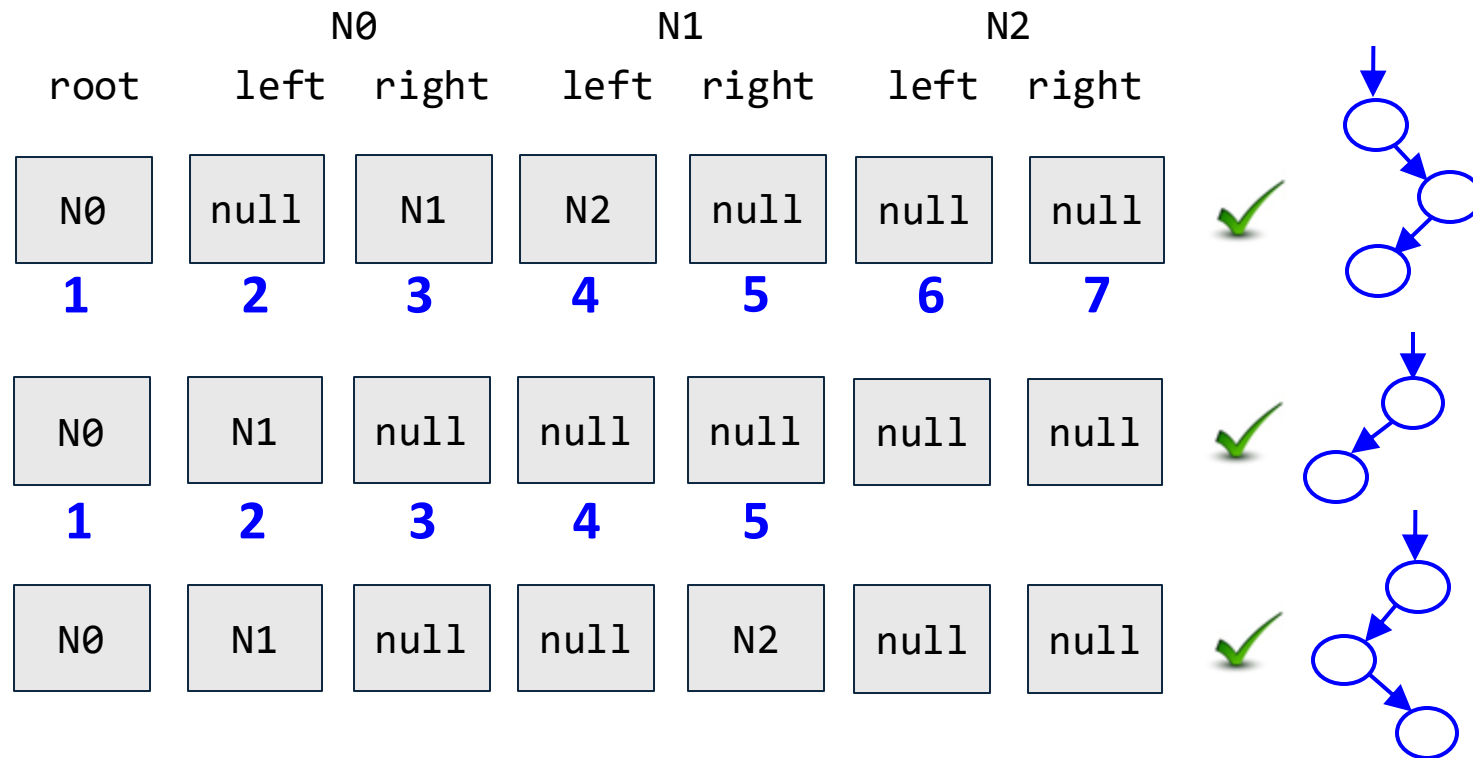
What are the next two legal, non-isomorphic shapes Korat generates?

	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?





# SEGMENT

## Korat in Practice

# 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')
```



