Lecture 20: Unit Test Generation II

CS 5150, Spring 2025

Administrative Reminders

Project Report #3 Due Today! No Extensions allowed.

Previously…

- **Randoop**:
    - Generating unit tests by generating API call sequences and incorporating execution feedback
    - Use API contracts as assertions

- Coverage/Mutation Analysis.

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, …

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

Poll: PollEv.com/cs5150sp25

- Which of the statements are **not** true about code coverage and mutation analysis?

Lecture Goals

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

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?

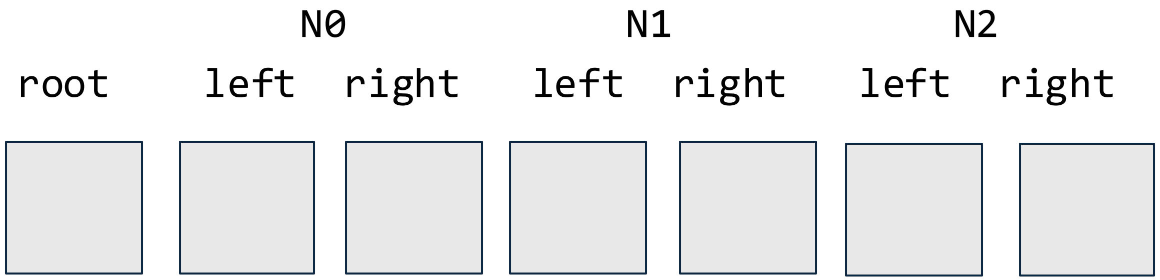
- Use the types
- The class declaration shows what values (or null) can fill each field
- Simply enumerate all possible shapes with a fixed set of Nodes

```
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:

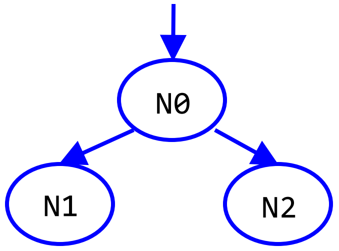


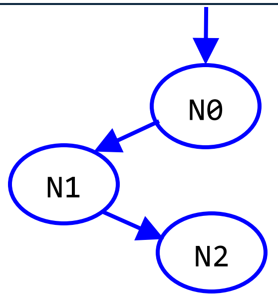

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

Activity: 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

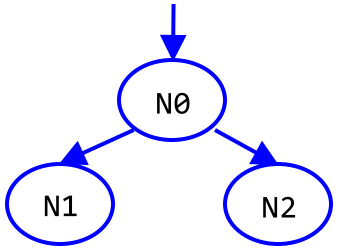


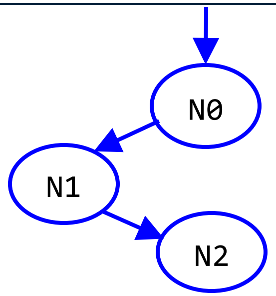


Activity: Representing Shapes

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





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

Activity: 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?

Activity: 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?
- 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”

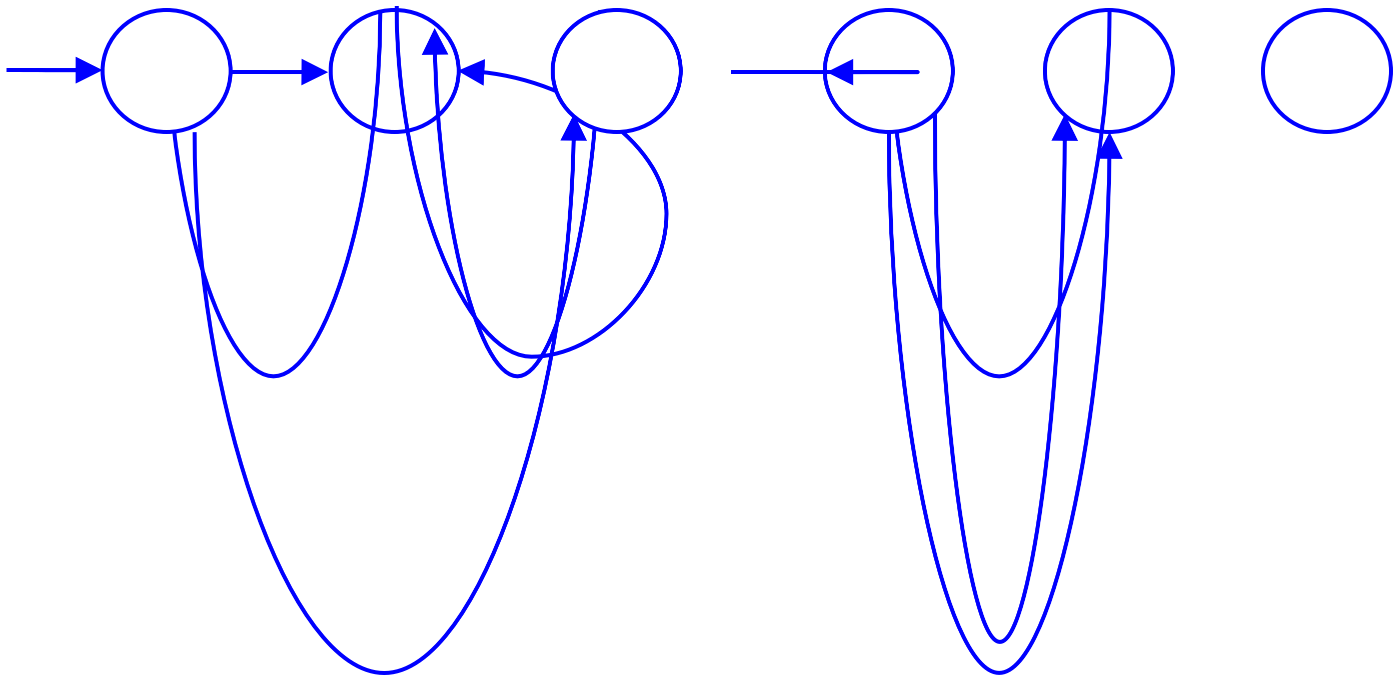
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

A Lot of “Trees”!

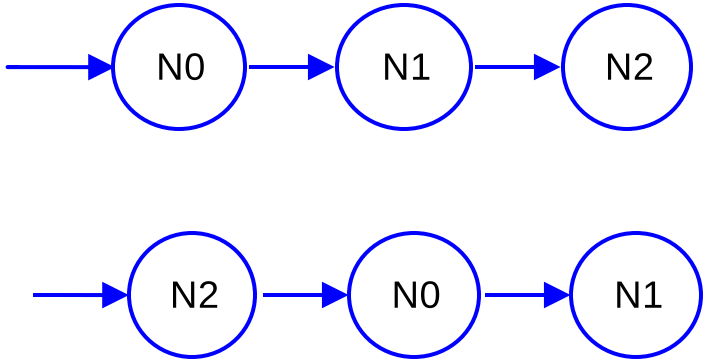
- The number of “trees” explodes rapidly
  - k = 3: over 16,000 “trees”
  - k = 4: over 1,900,000 “trees”
  - k = 5: over 360,000,000 “trees”
- Limits us to testing only very small input sizes
- Can we do better?

An Overestimate

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

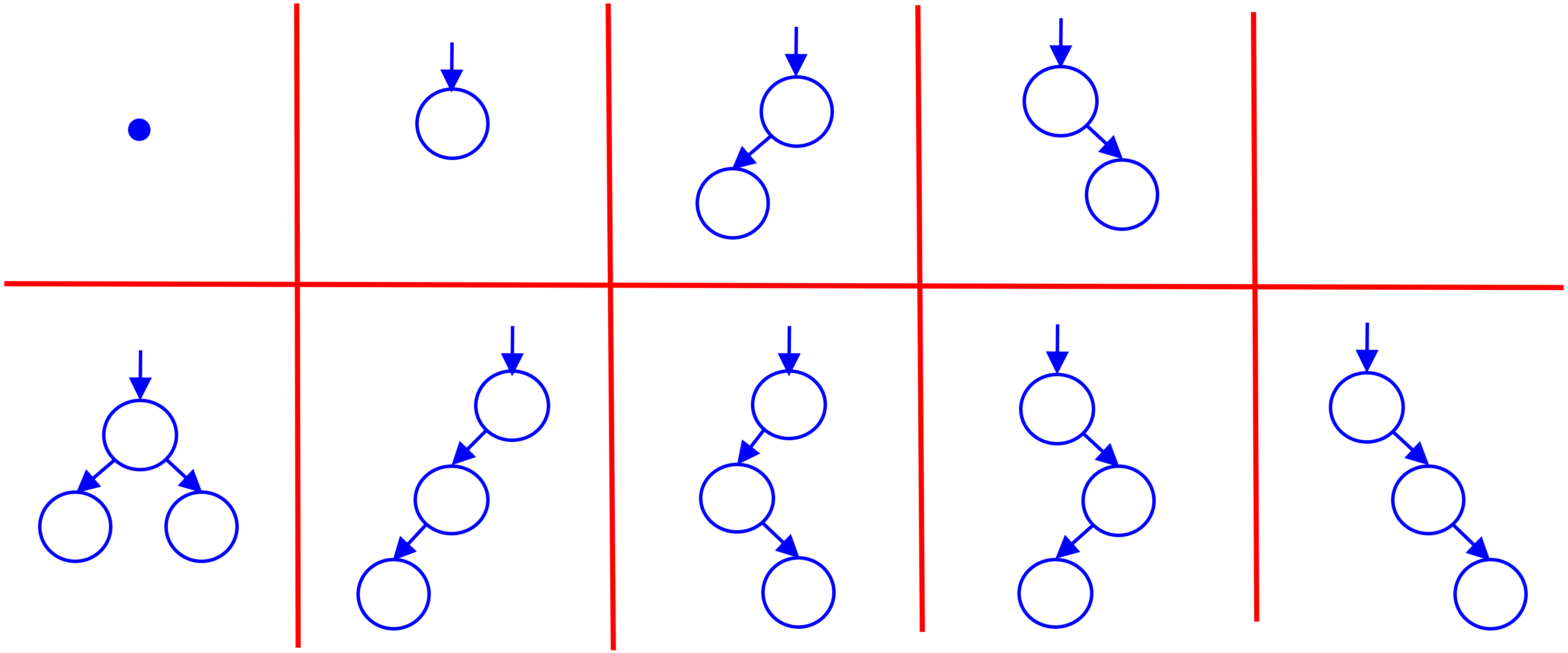


- And many are isomorphic:



How Many Trees?

There are only 9 distinct (non-isomorphic) 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

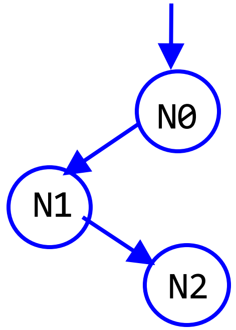
```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left !=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

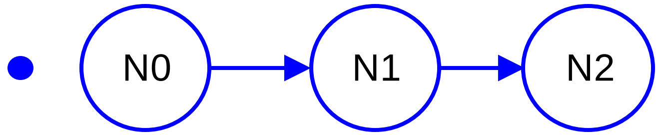
```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left !=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



root	N0 left	N0 right	N1 left	N1 right	N2 left	N2 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!

Example: Generated Test

```
@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;
  }
}
```

Korat will generate a test creating a binary tree that satisfies the **invariant**, and other inputs that satisfy the **pre-condition**

The test will then contain an assertion checking the **post-condition**

SEGMENT

Enumerating Tests

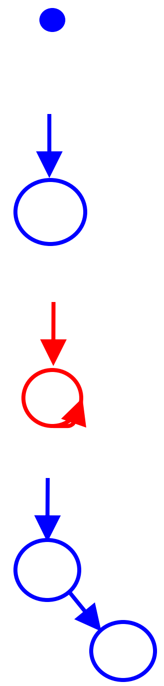
Enumerating Tests

- 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

See paper for details: http://mir.cs.illinois.edu/marinov/publications/BoyapatiETAL02Korat.pdf

Example: Enumerating Binary Trees

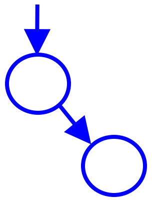
	N0		N1		N2		
root	left	right	left	right	left	right	
null	null	null	null	null	null	null	✓
1							
N0	null	null	null	null	null	null	✓
1	2	3					
N0	null	N0	null	null	null	null	✗
N0	null	N1	null	null	null	null	✓
1	2	3	4	5			



Activity: Enumerating Binary Trees

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

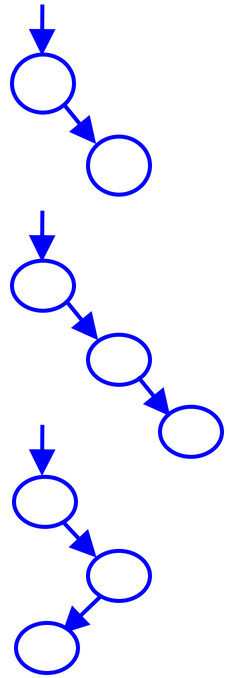
	N0		N1		N2		
root	left	right	left	right	left	right	
N0	null	N1	null	null	null	null	✓
1	2	3	4	5			
							✓
							✓



Activity: Enumerating Binary Trees

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

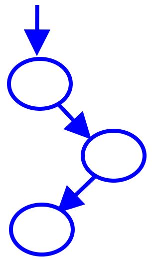
	N0		N1		N2		
root	left	right	left	right	left	right	
N0	null	N1	null	null	null	null	✓
1	2	3	4	5			
N0	null	N1	null	N2	null	null	✓
1	2	3	4	5	6	7	
N0	null	N1	N2	null	null	null	✓



Activity: Enumerating Binary Trees

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

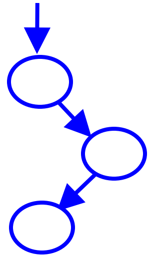
	N0		N1		N2	
root	left	right	left	right	left	right
N0	null	N1	N2	null	null	null

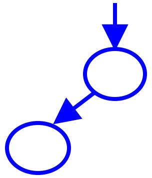


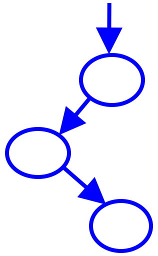
Activity: Enumerating Binary Trees

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	5	6	7
N0	N1	null	null	null	null	null
1	2	3	4	5		
N0	N1	null	null	N2	null	null







Poll: PollEv.com/cs5150sp25

Q: How many Binary trees of max size 2 can be generated by Korat?

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’)
```

QUIZ: Randoop and Korat

Identify which statements are true for each test generation technique:

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

QUIZ: Randoop and Korat

Identify which statements are true for each test generation technique:

	Randoop	Korat
Uses type information to guide test generation.	☑	☑
Each test is generated 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

In Class Exam 1 Discussion?

- 1A:
  6.b: Critical Path
  7.1: static vs dynamic
  8: UML Diagram
  10: Builder pattern
  11: The goal of user testing is to allow test evaluators to determine design choices
  14: Singleton