CIS 547
Software Analysis

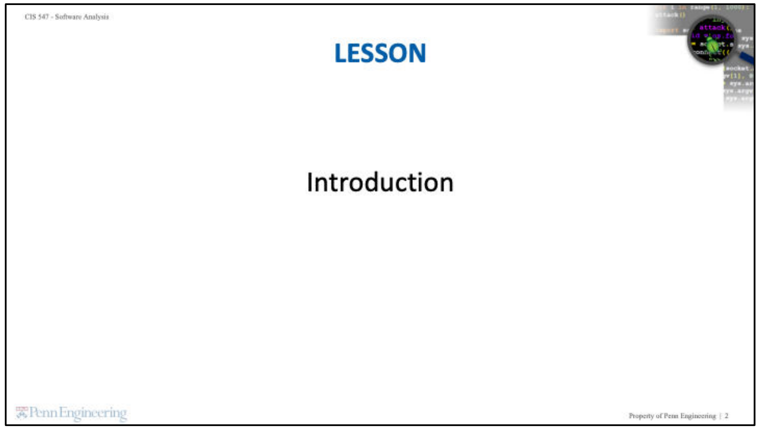
Automated Test Generation

Mayur Naik

Writing and maintaining tests is a tedious and error-prone process. Modern technology has come a long way to help automate parts of this process.

In this lesson, we will learn about techniques for automated test generation. Since automated testing is impractical for entire systems, we will focus on techniques for automatically testing the functionality of the smallest testable parts of an application, called units. This approach, called unit testing, constitutes a software development process which is important in its own right.

The techniques we will learn in this lesson are more directed compared to random testing: they observe how the program under test behaves on past tests in order to guide how to generate future tests. By being more directed, these techniques not only help find bugs more efficiently, but they also help to create a concise test suite that can be used for regression testing.

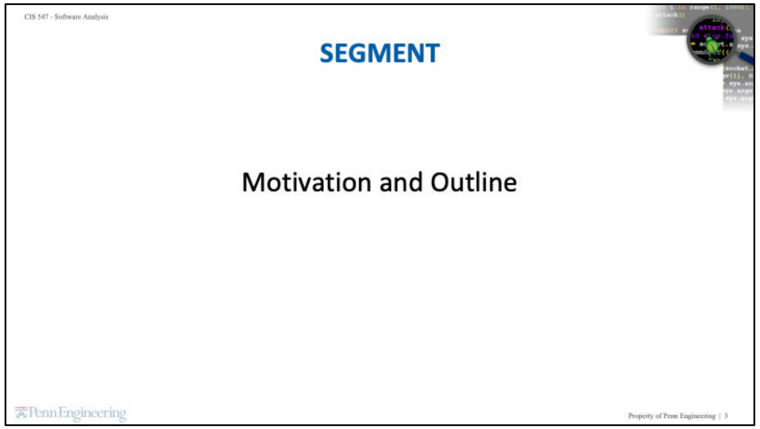


CIS 547 - Software Analysis

LESSON

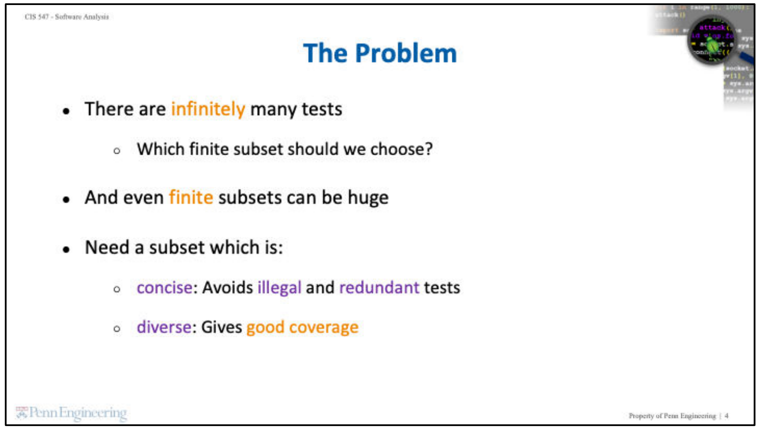
Introduction

Property of Penn Engineering | 2



SEGMENT

Motivation and Outline



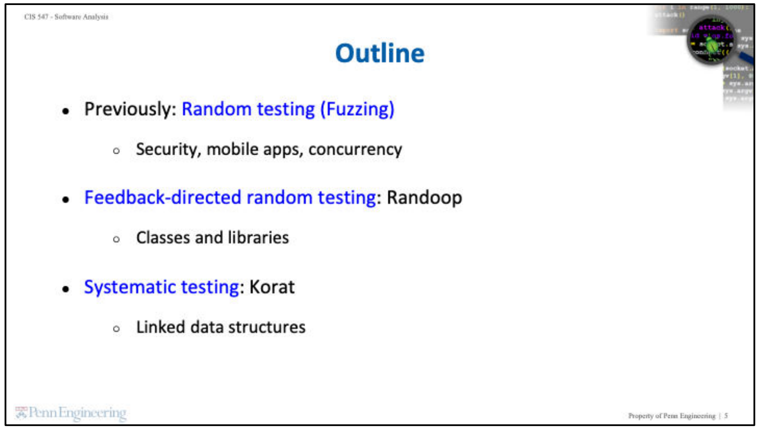
The Problem

- 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

We alluded to this problem previously in the course: there are potentially infinitely many tests we could run on a given piece of software, but we only have the ability to run a finite number of tests.

And even if we restrict ourselves to tests of a finite size, the space of possible tests can become astronomically large very quickly.

We need to choose a subset that is concise and diverse. By concise, we mean that it avoids two kinds of test inputs: illegal ones that do not exercise interesting functionality of the software, and redundant ones that exercise the same facet of the software. By diverse, we mean that it gives good coverage of the software (as measured by some number of code coverage metrics).



CIS 547 - Software Analysis

Outline

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

Penn Engineering

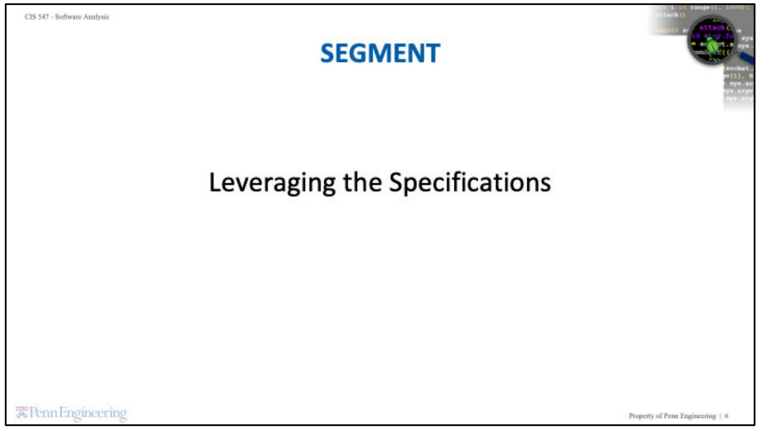
Property of Penn Engineering | 5

Previously, we looked at random testing, or fuzzing, as a technique for testing. Recall that fuzzing is useful for finding possible security bugs as well as for testing mobile apps and multithreaded programs.

In this module, we will focus on more directed forms of testing that overcome a major limitation of purely random testing: they avoid generating illegal and redundant inputs that dominate the space of possible test inputs.

One such testing approach, conceived in a tool called Randoop, is called feedback-directed random testing. It is suited for unit testing of Java program fragments like classes and libraries.

Then we will look at a systematic testing approach, embodied in a tool called Korat, which is suited for testing routines that manipulate linked data structures such as lists and trees.



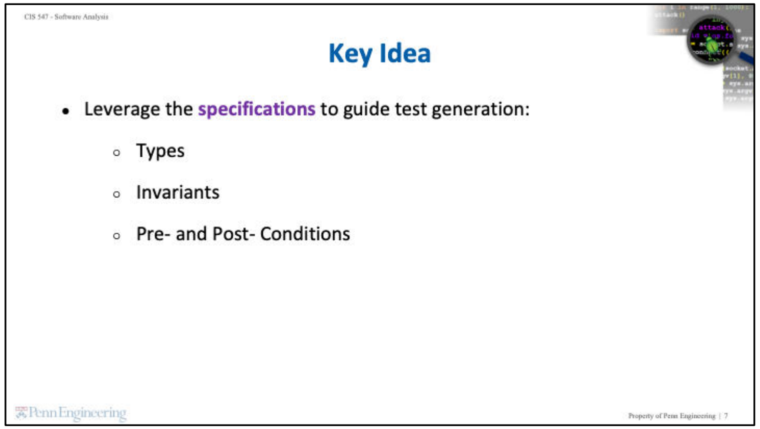
CIS 547 - Software Analysis

SEGMENT

Leveraging the Specifications

Penn Engineering

Property of Penn Engineering | 6



CIS 547 - Software Analysis

Key Idea

- Leverage the **specifications** to guide test generation:
  - Types
  - Invariants
  - Pre- and Post- Conditions

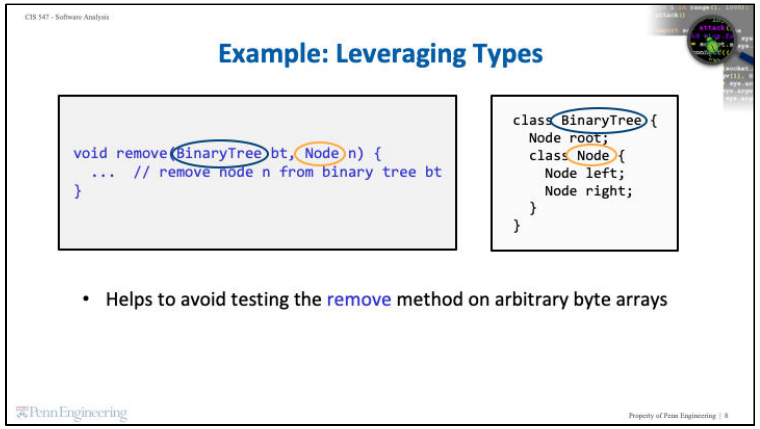
Penn Engineering

Property of Penn Engineering | 7

The key idea behind the techniques covered in this module is to leverage program specifications to guide test generation.

The forms of specifications include the ones we learnt earlier in the course: types, invariants, and pre- and post- conditions.

Let’s look at the example of a Binary Tree data structure to illustrate what each of these forms of specifications look like, and how they can help guide test generation.



CIS 547 - Software Analysis

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;
  }
}
```

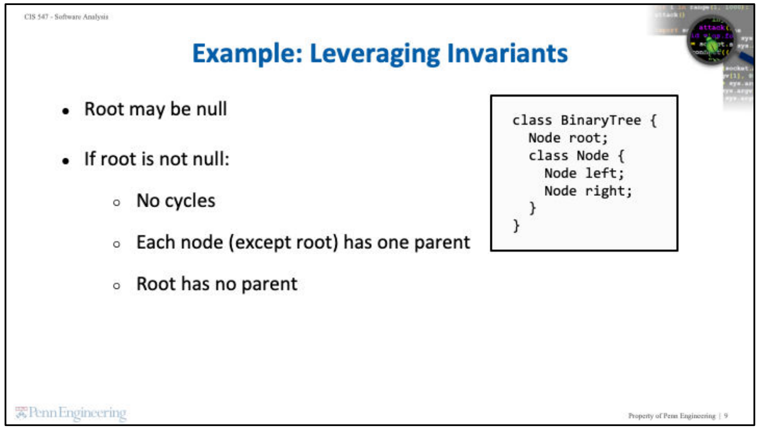
- Helps to avoid testing the remove method on arbitrary byte arrays

Penn Engineering

Property of Penn Engineering | 8

Recall from the module on random testing that a function-level fuzzer like libfuzzer treats the function’s input as a byte array. However, functions typically specify more structured inputs than byte arrays, especially in programming languages with stronger type systems.

Consider this remove method in Java, for instance. It takes two arguments: a binary tree and a node to remove from the tree. The BinaryTree and Node types provide valuable information to guide the generation of test inputs to this function. The space of candidate inputs to this remove method is significantly smaller than the space of candidate inputs if this method were written in, for instance, a language without static type information, such as Python or Javascript.



CIS 547 - Software Analysis

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;
  }
}
```

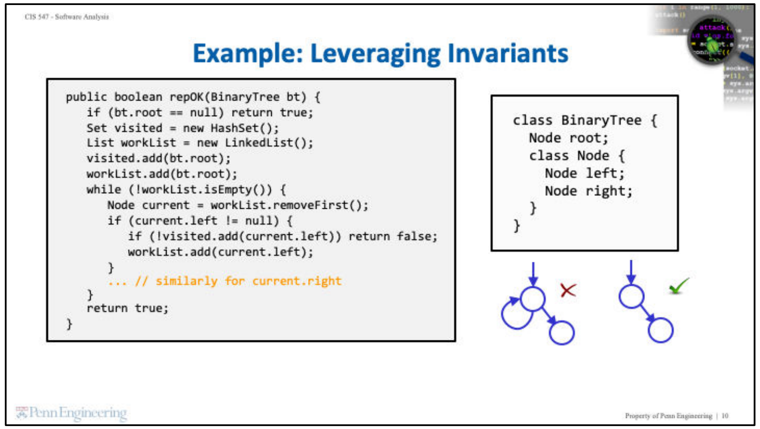
Penn Engineering

Property of Penn Engineering | 9

We can extend this idea to other kinds of specifications such as invariants. Let's look at an invariant for binary trees.

For a data structure of this type [gesture to code] to be a valid binary tree, it should satisfy one of two different cases:

the root may be null
if the root is not null, then the structure should be a directed tree. That is:
It should have no cycles
Every node except the root should have one parent
And the root should have no parent



CIS 547 - Software Analysis

Example: Leveraging Invariants

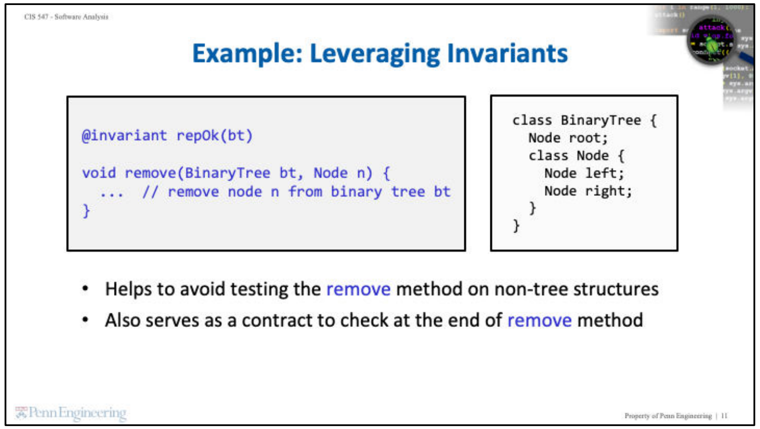
```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left !=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

Penn Engineering

Property of Penn Engineering | 10

Here is a concrete implementation of the invariant we just outlined for binary trees. It is a function called repOK in the terminology of Korat, one of the tools we will study later in this module. The repOK function takes as input a BinaryTree object bt, and returns true if it is a valid binary tree, and false otherwise. Let's not delve into the details of this function for now; we will return to it later when we talk about Korat. For now, you should convince yourself that both the structures shown here respect the type of class BinaryTree, but only one of them satisfies the shown invariant and therefore is a valid binary tree.



CIS 547 - Software Analysis

Example: Leveraging Invariants

```
@invariant repOk(bt)

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

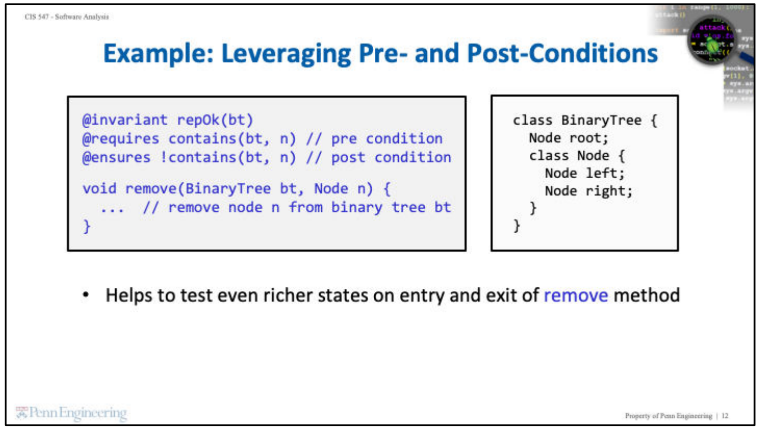
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

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

Penn Engineering

Property of Penn Engineering | 11

We can now leverage the repOK function implementing the invariant on binary trees to further prune the space of candidate inputs to the remove method. In particular, we can avoid generating structures that respect the type of class BinaryTree but that violate the invariant. Furthermore, we can use the invariant to check that the remove method never results in an invalid binary tree upon completion, on each generated input binary tree that satisfies the invariant. A programmer can specify such invariants for use by an automated test generation tool via annotations in the source code such as this [gesture to @invariant].



CIS 547 - Software Analysis

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
}
```

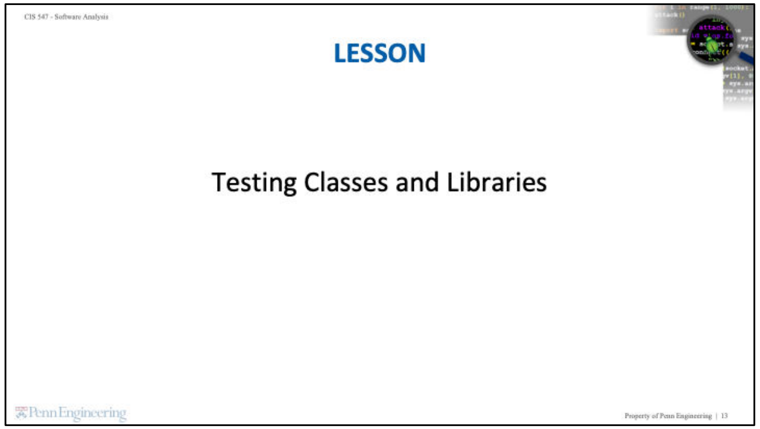
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

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

Penn Engineering

Property of Penn Engineering | 12

Going even further, the programmer can specify pre and post conditions, denoted in our example program via requires and ensures annotations, respectively. In this case, the pre-condition states that the input node n must be contained in the input binary tree bt. Such a pre-condition could further prune the space of candidate inputs by avoiding generating inputs where the node n is not contained in the binary tree bt. Furthermore, the post-condition could be used to check that the remove method indeed removes the given node from the given binary tree, on each generated test input. You should by now have a good sense of how far specifications can go in helping to guide test generation.



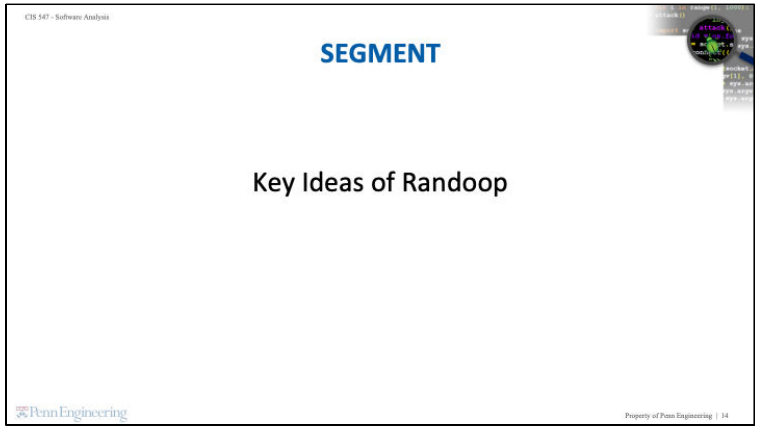
CIS 547 - Software Analysis

LESSON

Testing Classes and Libraries

Penn Engineering

Property of Penn Engineering | 13



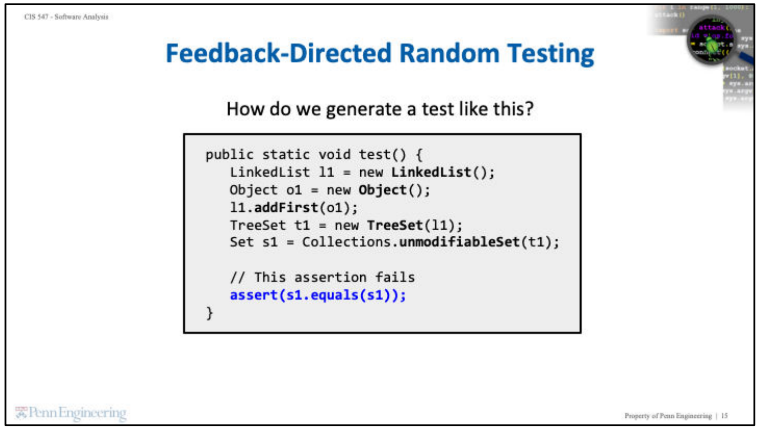
CIS 547 - Software Analysis

SEGMENT

Key Ideas of Randoop

Penn Engineering

Property of Penn Engineering | 14



CIS 547 - Software Analysis

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));
}
```

Property of Penn Engineering | 15

In this lesson, we will study a technique called feedback-directed random testing, first conceived in a tool called Randoop, which stands for Random tester for Object-Oriented Programs.

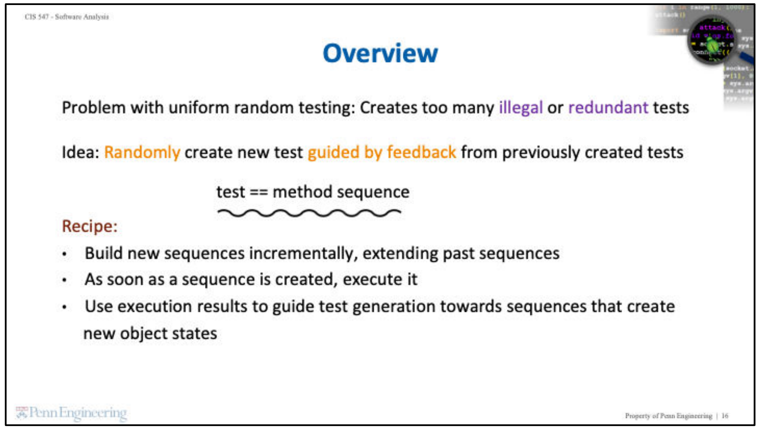
Let’s look at a concrete example that involves testing the API of various container classes in Java’s standard library, such as LinkedList and TreeSet.

Here is a real unit test generated by Randoop when run on the Java standard library version 1.5. This test shows a violation of the equals contract: a set s1 returned by the unmodifiableSet() library method is not deemed equal to itself, which violates the reflexivity property of the equals() method as specified in the Java standard library API’s documentation.

This test actually reveals two errors: one in the equals() method and another in the TreeSet() constructor method, which failed to throw ClassCastException as required by its specification.

Formally speaking, Randoop generates object-oriented unit tests consisting of a sequence of method calls that set up state (such as creating and mutating objects) and an assertion about the result of the final call.

How does Radoop generate such tests? Let’s take a look.



CIS 547 - Software Analysis

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

Property of Penn Engineering | 16

In this setting, a test is a sequence of public library methods that an application using the library can call.

While uniform random testing was suitable for the applications in the previous lesson, such as security testing and testing of entire systems like mobile apps or concurrent programs, here we want to generate a concise yet diverse test suite for testing a library.

A major problem with the uniform random testing approach here is that it would create too many illegal and redundant method sequences. (We will see examples of such sequences shortly.)

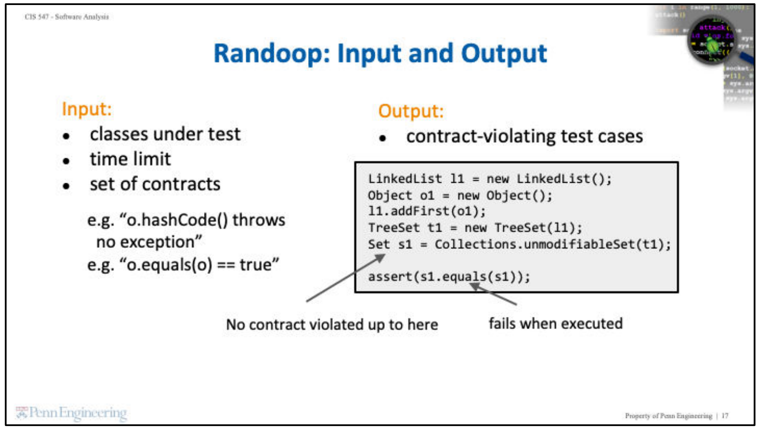
This motivates Randoop’s variant of random testing called feedback-directed random testing, in which each new test is guided by feedback from running previously created tests. In what follows, we will use the terms test and method sequence interchangeably.

Randoop uses the following recipe:

It builds new method sequences incrementally by extending past method sequences.
As soon as a method sequence is created, it executes the sequence.
It then uses the result of this execution to determine whether it is an illegal or a redundant sequence, in which case it discards the sequence.

Otherwise, it remembers the sequence to extend it when generating future sequences.

It thereby guides test generation towards sequences that create new object states instead of ones it has already seen.



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

Let's begin by looking at Randoop's input and output.

Randoop's input consists of three things:

a set of classes making up the library under test,
a time limit that allows the user to control how many tests Randoop generates, and
a set of contracts, which are properties that the object generated in the final call of a Randoop-generated test is required to satisfy.

Generally, these properties are specified in the library API's documentation. An example contract that must hold for any Java object o is that calling the hashCode method on it does not throw any exception. Another example contract is that using the equals method to test any Java object o's equality to itself must always return true. While these are generic contracts defined by the java.lang.Object class, which is the parent class of all Java objects, one can also provide class-specific contracts for Randoop to check.

Randoop's output consists of contract-violating test cases. Here is the example such test case that we saw earlier. It consists of two parts:

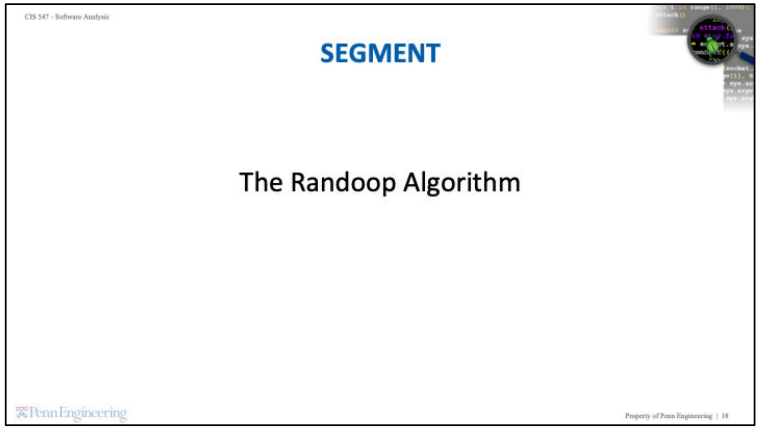
a sequence of calls to public methods in the input classes, which set up state (such as creating and mutating objects), and
a contract stated as an assertion about the result of the final call in the sequence, in

this case s1.

In order for this test case to be valid, two conditions must hold.

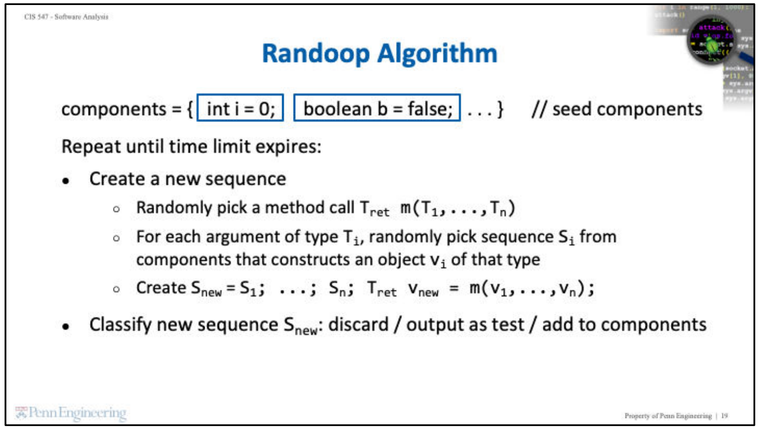
First, no contract should be violated up to the end of the execution of the sequence; for instance, no exception should be thrown during the execution of the sequence.
Second, the final assertion should fail when executed.

These two conditions will be important in deciding what Randoop does with each test case it generates: whether to discard the test case, whether to output the test case, or whether to retain it for extending in future test cases.



SEGMENT

The Randoop Algorithm



CIS 547 - Software Analysis

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_{ret} `m(`T_1`,...,`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_{new} = S_1`; ...; `S_n`; `T_{ret} v_{new} ` = m(`v_1`,...,`v_n`);`
- Classify new sequence S_{new}: discard / output as test / add to components

Penn Engineering

Property of Penn Engineering | 19

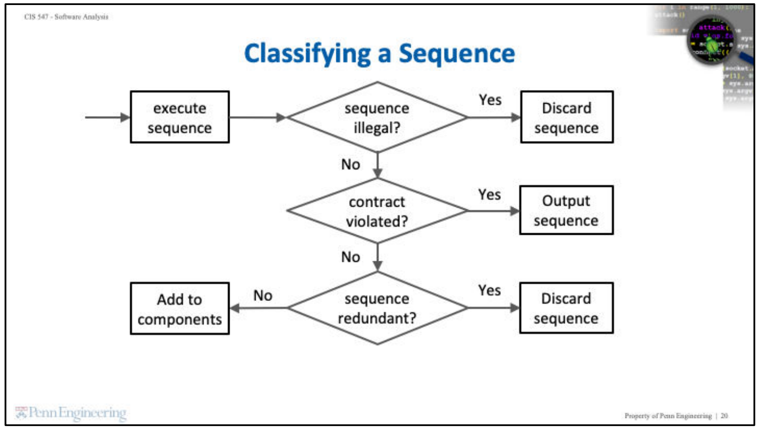
Now that we’ve looked at the input and output of Randoop, let’s look at the algorithm underlying Randoop. Throughout, the algorithm maintains a set of components. Each component is a sequence of code statements that computes values for arguments of the methods that Randoop decides to call in the sequence it generates. This set of components starts out with singleton sequences called seed components. For example, we have shown two seed components here. If Randoop decides to call a method with an integer-typed argument, it can use this first component which computes the integer i. If Randoop decides to call a method with an boolean-typed argument, it can use this second component which computes the boolean b. And so on. This set of components grows with progressively longer sequences as the algorithm progresses. These sequences will compute increasingly complex objects such as linked lists.

Next, the algorithm repeats the following process until the specified time limit expires. It first creates a new sequence of method calls. For this purpose, it does three things in order.

First, it randomly picks a public method from the input classes to call. Let’s say this method is named m, and that it has n arguments of types T1 through Tn, and that it returns a result of type T_ret. m is the final method that the algorithm will call in this sequence. Calling this method alone will result in an invalid test case that might not even compile. In particular, the algorithm must assign a type-compatible value to each of the n arguments before calling this method.
For each argument of type Ti, Randoop randomly picks a sequence Si already generated from the components that constructs an object vi of that type.
Finally, it assembles the n randomly picked sequences, one for each of the n arguments, into a new sequence, and appends the chosen method m at the end of the sequence. This becomes a new sequence denoted S_new. Just like each of the sub-sequences S1 through Sn produced a result stored in variables v1 through vn respectively, the new sequence S_new produces a result that is stored in variable v_new.

Once a new sequence is created, the algorithm has three options to dispose of the sequence: it can discard the sequence, it can output the sequence as a test, or it can add the sequence to the set of components. This third option is what allows the algorithm to produce increasingly richer method sequences over time: each new method sequence that is added to the set of components itself becomes available for extension in future method sequences that the algorithm generates.

Let’s look next at how Randoop decides how to classify each sequence.



CIS 547 - Software Analysis

Classifying a Sequence

Penn Engineering

Property of Penn Engineering | 20

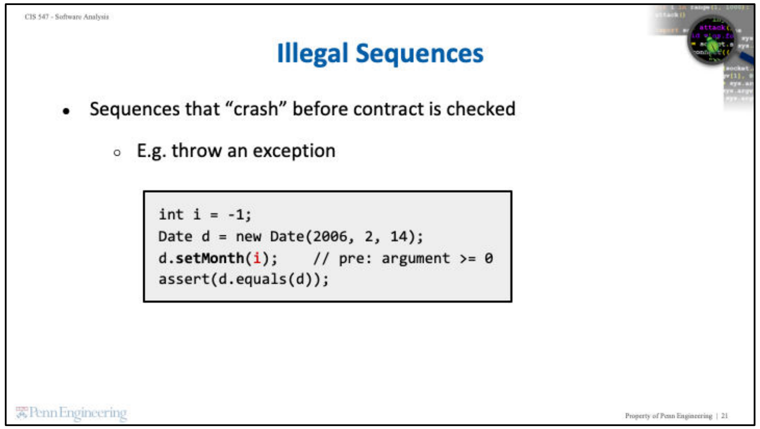
Randoop uses the following decision process to classify each sequence that it creates in the algorithm we just saw.

It executes the sequence and makes three checks on the result. The first check determines whether the sequence is illegal. Intuitively, a sequence is illegal if it violates a precondition, for example, if an exception is thrown during the execution of the sequence. If so, the sequence is discarded.

Otherwise, it checks whether the sequence violates a contract. Recall that a contract is analogous to a post-condition, an assertion on the final result of the sequence. If a contract is violated, it outputs the sequence as a contract-violating test case. Otherwise, it checks whether the sequence is redundant. If so, it discards the sequence. Otherwise, it adds the sequence to the set of components.

The first two checks ensure that such a sequence does not violate any pre-condition or any post-condition; there is no point in extending such a sequence further. The third check ensures that such a sequence results in a program state that has not been seen before; it is not necessary to extend such a sequence further as its effect has already been captured by some other sequence already in the set of components.

Let's elaborate a bit more on how Randoop decides whether a sequence is illegal and whether a sequence is redundant.



CIS 547 - Software Analysis

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));
```

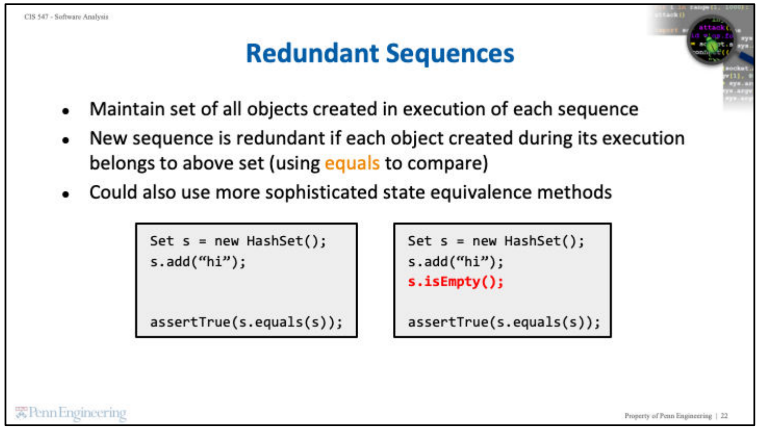
Penn Engineering

Property of Penn Engineering | 21

An illegal sequence is one that crashes before the contract at the end of the sequence is checked, for example, by throwing an exception. Intuitively, an illegal sequence is one which violates some pre-condition during execution.

Here is an example of an illegal sequence. In this test case generated by Randoop, the contract d.equals(d) is not even reached, because the call to setMonth() throws an exception. The reason is that a pre-condition of the setMonth method is that the argument i denoting a month must be non-negative, whereas in this test case, it is -1.

As an aside, you might be wondering how Randoop generates arguments like 2006, 2, and 14 used in the call to Date's constructor method. The answer is that the set of seed components in Randoop is highly configurable: it allows a wide range of possible constants.



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));
```

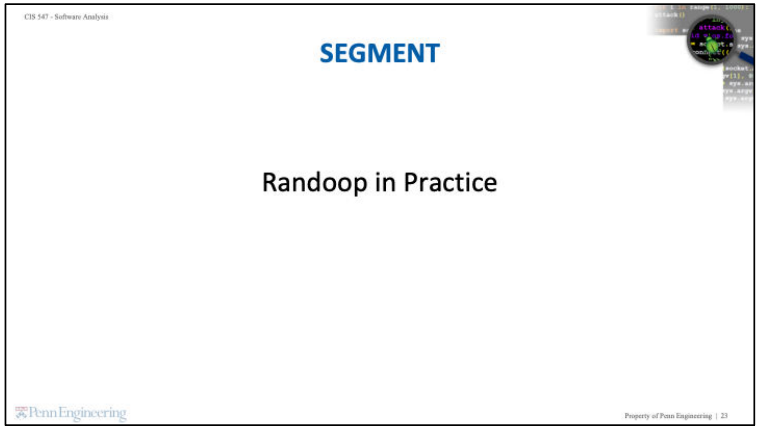
Randoop uses a simple heuristic to decide whether a sequence is redundant.

It maintains the set of all objects created in the execution of each sequence. A new sequence is considered redundant if each object created during its execution belongs to this set. Randoop uses the equals() method to do this comparison check for each object.

Let’s look at an example. Consider this sequence which creates a hashSet and adds a single element, the string “hi”, to it. Suppose we then extend this sequence with a call to s.isEmpty(). Is this new sequence redundant compared to this sequence? The answer, according to Randoop, is yes. Because the equals method states that the set object s in the new sequence is logically equivalent to the set object s in the old sequence.

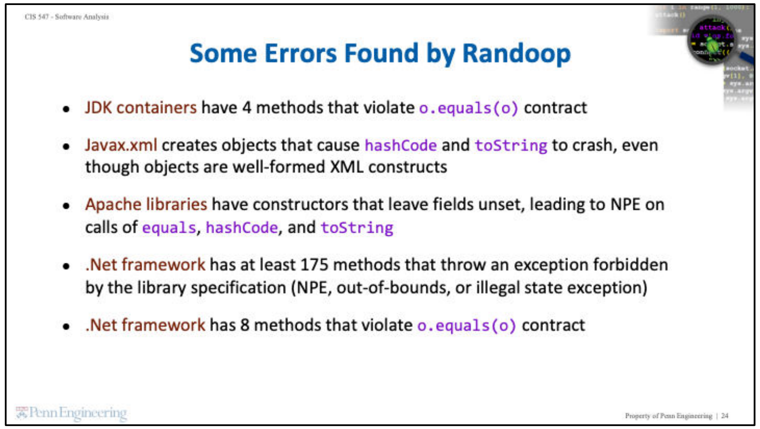
This is not a fool-proof comparison. On one hand, this comparison may mistakenly deem a new object as equal to a previously created object, causing Randoop to conclude that a sequence is redundant, and thereby miss bugs that can be triggered only by extending that sequence. On the other hand, this comparison may mistakenly deem a new object as not equal to a previously created object, causing Randoop to conclude that a sequence as non-redundant. While this case is less problematic -- in that it at least won’t cause Randoop to miss bugs -- it does bloat the set of components and reduce the chance of creating new object states in future sequences.

Nevertheless, Randoop’s heuristic works well in practice, and it can be disabled or refined by the user. For instance, it is straightforward to use reflection to write a method that determines whether two objects have the same concrete representation (that is, the same values for all their fields), or the user can specify more sophisticated methods to determine object equality.



SEGMENT

Randoop in Practice



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

The authors of Randoop applied the tool to three large libraries:

the Java standard library, or JDK;
the Apache commons collection of reusable components, and
Microsoft's .Net framework.

Randoop discovered previously unknown errors in all three libraries. Let's take a look at some of these errors.

In the JDK's containers library, which includes facilities to create containers such as lists, trees, etc., Randoop found four methods that violate the reflexivity property of the equals method.

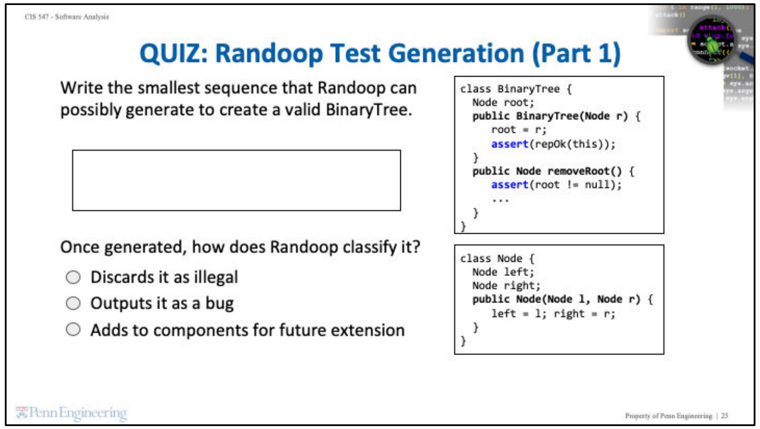
Randoop also found that the JDK's XML library can create objects that cause the hashCode and toString methods to crash even though those objects are well-formed XML constructs.

In the Apache library, Randoop revealed constructors that leave fields uninitialized, leading to null-pointer exceptions on calls to equals, hashCode, and toString methods.

Randoop also found many errors in the .Net framework: this includes at least 175 methods that throw an exception forbidden by the library's specification, such as a null-pointer exception, an array out-of-bounds exception, or an illegal state exception. Randoop also discovered 8 methods in the .Net framework that violate the reflexivity property of the equals method.

You can learn more about Randoop by reading the following technical paper:

http://research.microsoft.com/pubs/76578/randoop-tr.pdf



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 SLIDE}

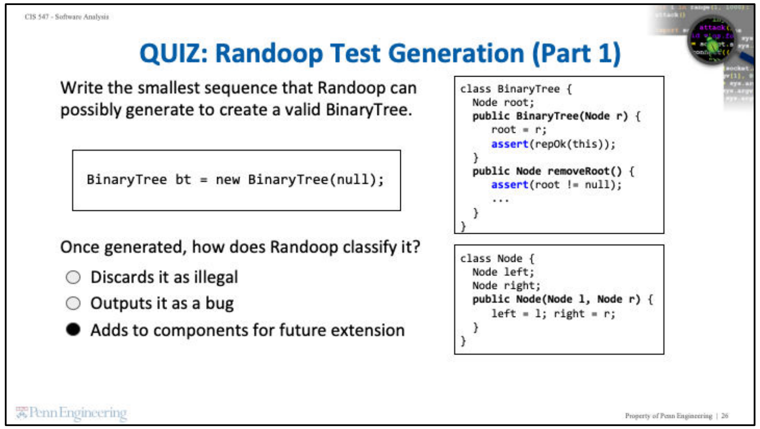
Now let's do a few quizzes so you can check your understanding of the Randoop algorithm.

For this quiz, let's look at the BinaryTree class we considered earlier for Korat. This time, our API consists of three methods:

a constructor for the BinaryTree class which takes as an argument a single Node (which becomes its root),
a removeRoot method for the BinaryTree class which takes no arguments and returns a Node object, and
a constructor for the Node class that takes two Nodes (which will become the left and right children of the newly created Node).

Note that the BinaryTree constructor has an assert statement that ensures that the BinaryTree object is a valid representation of a binary tree (using the repOk method we saw earlier) and that the removeRoot method has an assert statement requiring that the root of the BinaryTree is not null.

For the quiz, write the smallest possible Randoop sequence creating a valid BinaryTree object. Then, decide how Randoop will classify the sequence: will it discard the sequence as illegal, output the sequence as a bug, or add the sequence to components?



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;
  }
}
```

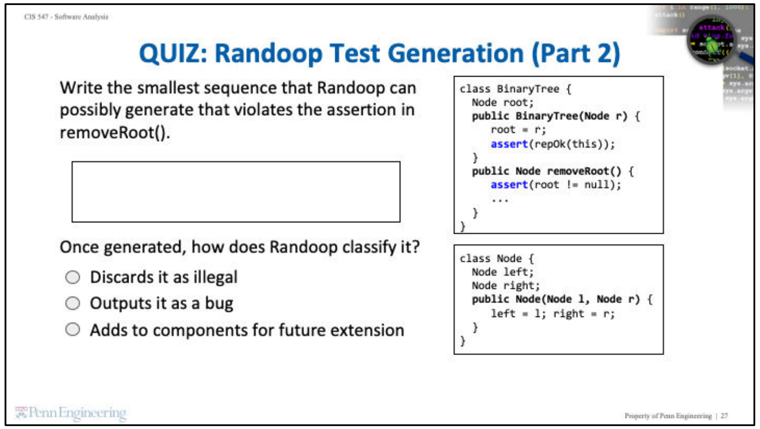
{SOLUTION SLIDE}

The smallest sequence that can generate a BinaryTree object is to call the BinaryTree constructor with a null argument.

In the statement shown here, we directly used the null constant as an argument to the constructor of BinaryTree. However, if you recall the Randoop algorithm, it always uses a variable generated by some sequence. If we were to be pedantic, we would need to add an additional statement at the start of the test case that assigns the null constant to a Node variable v, and then pass v as argument to the constructor of BinaryTree.

[Hand-write Node v = null and replace the null in the answer by v.]

After this sequence is generated, Randoop will classify it as a component for future use. Recall that a BinaryTree with a null root will pass the repOk check, so the assertion in the BinaryTree won't fail; thus the sequence isn't illegal. Additionally, any default or user-specified contracts on the final object created by this sequence will also succeed; thus the sequence is not output as a bug. Hence, Randoop will add the sequence to the set of components.



CIS 547 - Software Analysis

QUIZ: Randoop Test Generation (Part 2)

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

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

Once generated, how does Randoop classify it?

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

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

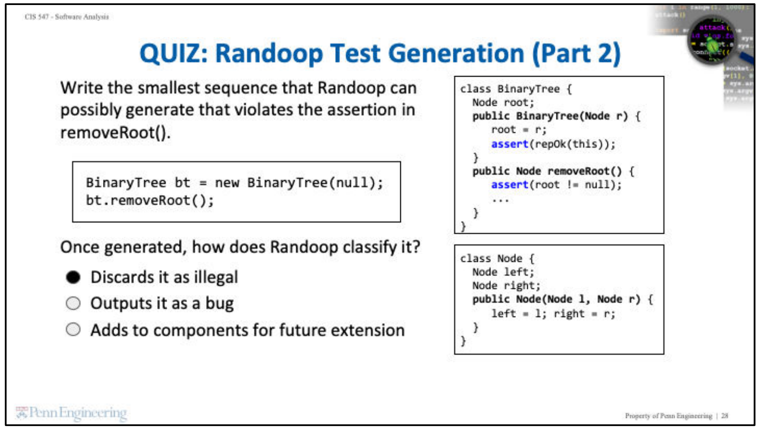
Penn Engineering

Property of Penn Engineering | 27

{QUIZ SLIDE}

Next, write the smallest sequence that Randoop can generate which will violate the assertion in the removeRoot() method.

Then decide how Randoop will classify the sequence generated.



CIS 547 - Software Analysis

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();
```

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

Once generated, how does Randoop classify it?

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

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

Penn Engineering

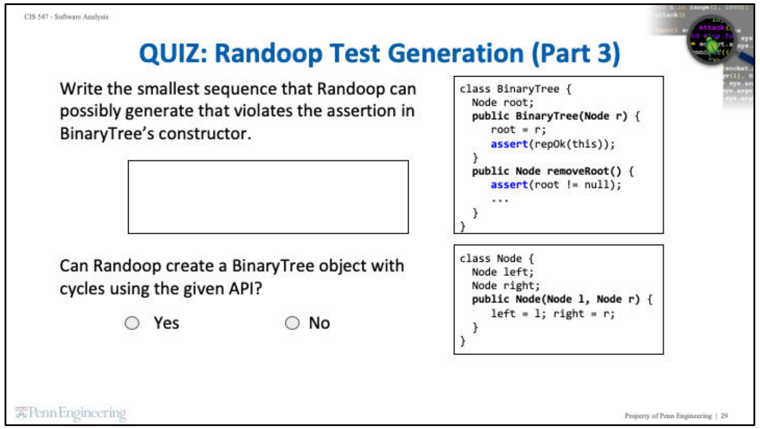
Property of Penn Engineering | 28

{SOLUTION SLIDE}

Because the assert statement in removeRoot() fails only if the tree's root is null, we must first create a BinaryTree with a null root. And then we call the removeRoot() method on that tree.

Indeed, this shows how Randoop can extend the sequence from the previous quiz which was added to the set of components.

Because the assert statement in removeRoot() is violated, though, Randoop will consider this sequence to be illegally violating a pre-condition and therefore discard the sequence.



CIS 547 - Software Analysis

QUIZ: Randoop Test Generation (Part 3)

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

```
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;
  }
}
```

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

○ Yes ○ No

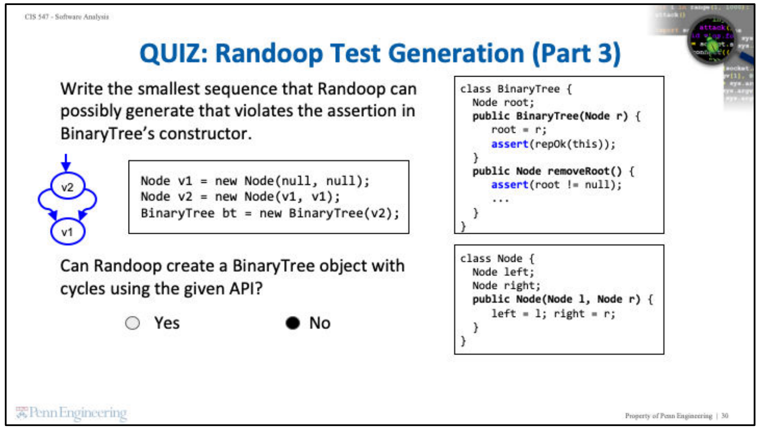
Penn Engineering

Property of Penn Engineering | 29

{QUIZ SLIDE}

Now, what's the smallest sequence Randoop can generate which will violate the assertion in the BinaryTree constructor method? (The code for the repOk() method is given in the instructor notes if you need it as a reference.)

After writing this sequence, answer the following question: is it possible for Randoop to create a BinaryTree object with a directed cycle using the given API?



CIS 547 - Software Analysis

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);
```

```
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;
  }
}
```

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

○ Yes ● No

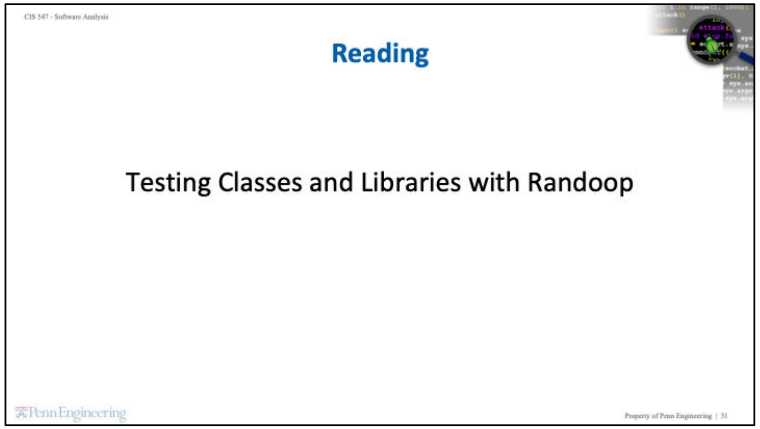
Penn Engineering

Property of Penn Engineering | 30

{SOLUTION SLIDE}

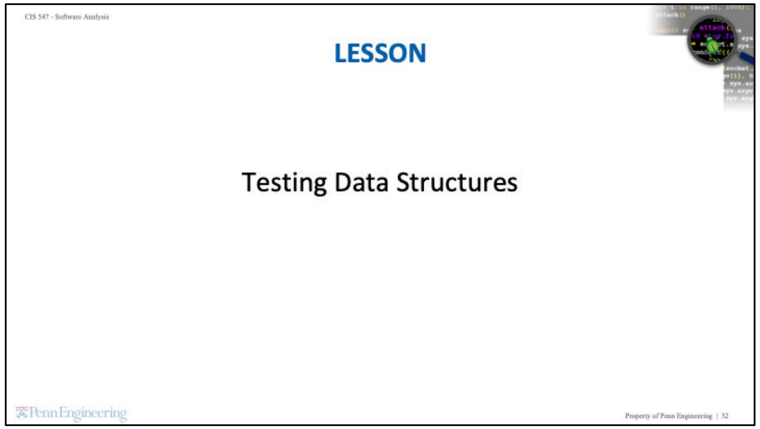
The smallest linked structure that does not pass the repOk method is a single node which points to itself, but it turns out that Randoop actually cannot create any cycles using this API. This is because a cycle would require some Node in the cycle to be passed as an argument to a previously created Node, but Randoop can only pass arguments which are components that have already been created in a previous sequence.

Therefore, in order to create a linked structure failing the repOk check, we need to create a structure with multiple directed paths to the same Node. The smallest such structure has a root node, both of whose children fields point the same node. To generate this structure in Randoop, three method calls are necessary: one to create the child node, one to create the root node, and one to call the BinaryTree constructor that will fail the repOk assertion.



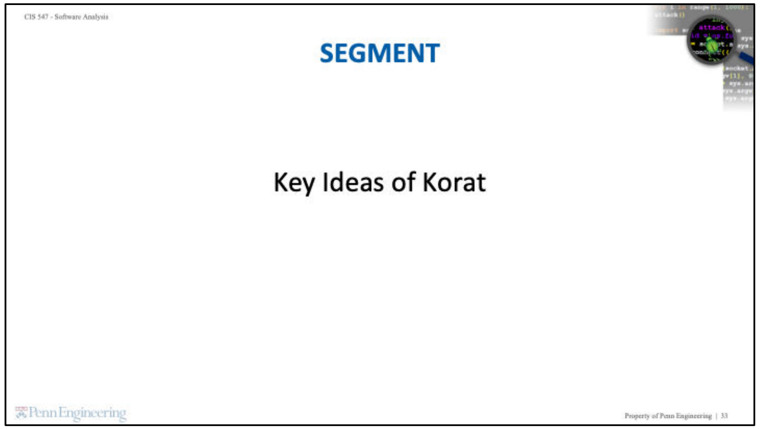
Reading

Testing Classes and Libraries with Randoop



LESSON

Testing Data Structures



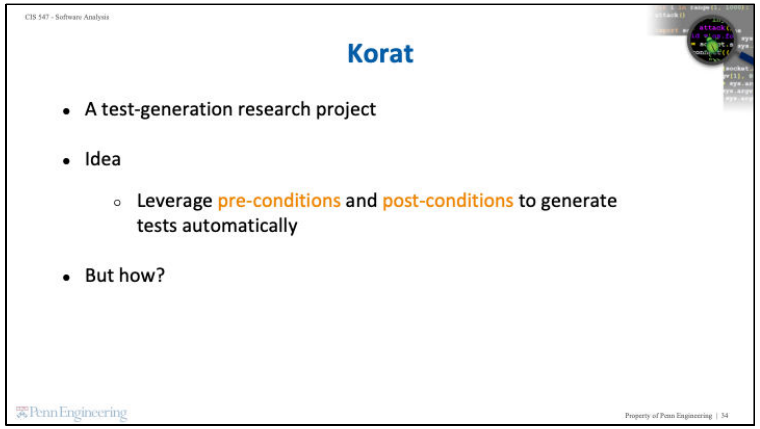
CIS 547 - Software Analysis

SEGMENT

Key Ideas of Korat

Penn Engineering

Property of Penn Engineering | 33



CIS 547 - Software Analysis

Korat

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

Penn Engineering

Property of Penn Engineering | 34

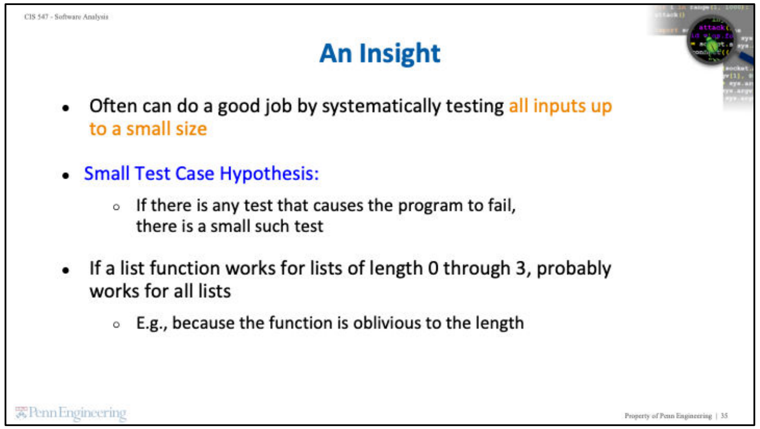
While Randoop was better suited for creating different sequences of methods in an object-oriented library that provides an application programmer interface, or API, Korat uses a deterministic test generation approach that is better suited for generating different shapes of a data structure.

This is not to say one technique is better than another in testing; in fact, they are complementary to each other. For instance, automatically testing a linked list implementation in a robust manner would require not only generating different sequences of list operations but also generating lists of different sizes.

Korat is a deterministic test generator which originated as a research project by a team of three graduate students at MIT.

The idea behind Korat is to leverage the pre- and post-conditions of a function to automatically generate relevant tests for the function.

How does it do this?



CIS 547 - Software Analysis

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

Penn Engineering

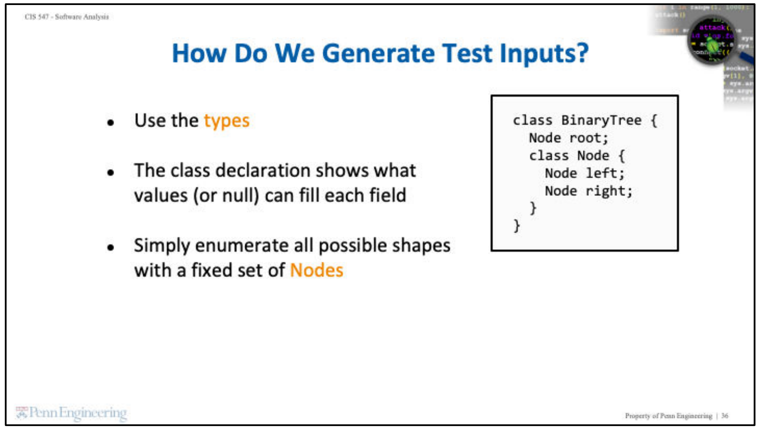
Property of Penn Engineering | 35

One insight into automating test creation can come from thinking about small test cases. In practice, we often do a good job of catching bugs by testing all inputs up to some small size.

This can be expressed in what is called the “small test case” hypothesis: if there is any test that causes the program to fail, then there is a small such test.

For example, if a list function works on lists of length 0, 1, 2, and 3, then it’s likely to work on lists of all sizes. The intuition behind this is that such a function is typically written in a manner that is oblivious to the length of the list. For instance, it is unlikely for a programmer to write such a function with a giant switch statement for lists of length 1, 2, 3, 4, etc.

Recall that we made a similar assumption with respect to bug depth in testing multithreaded programs using fuzzing: if there is a concurrency bug in a program, there is usually one with small depth (size 1 or 2). Again, this led us to focus our testing on a smaller search space while maintaining good concurrency bug coverage.



CIS 547 - Software Analysis

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;
  }
}
```

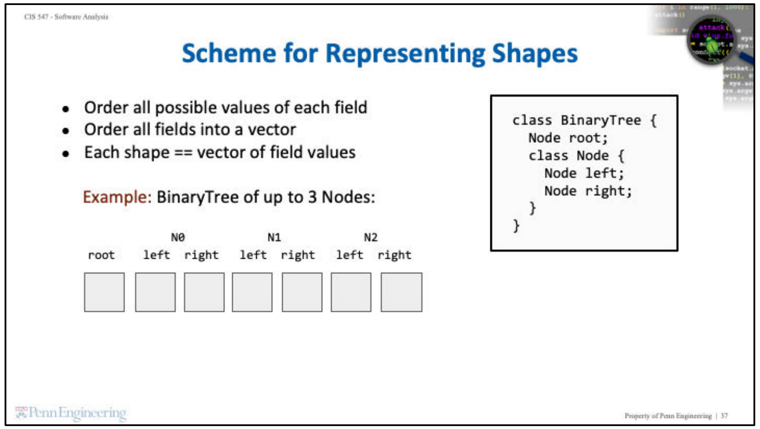
Penn Engineering

Property of Penn Engineering | 36

In order to systematically generate all test inputs upto a small size, Korat leverages knowledge of the type information of the input. (For this reason, we would consider Korat a white-box testing method.)

For instance, if we have a function that operates on a BinaryTree object, then Korat knows from the type of this object that it has a root field that points to a Node object or null, and that each Node object has two other fields, each of which points to a Node object or null.

Thus, to generate a set of small test inputs, Korat need only enumerate all possible shapes of BinaryTree objects that can be created from a fixed set of Node objects.



CIS 547 - Software Analysis

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;
  }
}
```

Penn Engineering

Property of Penn Engineering | 37

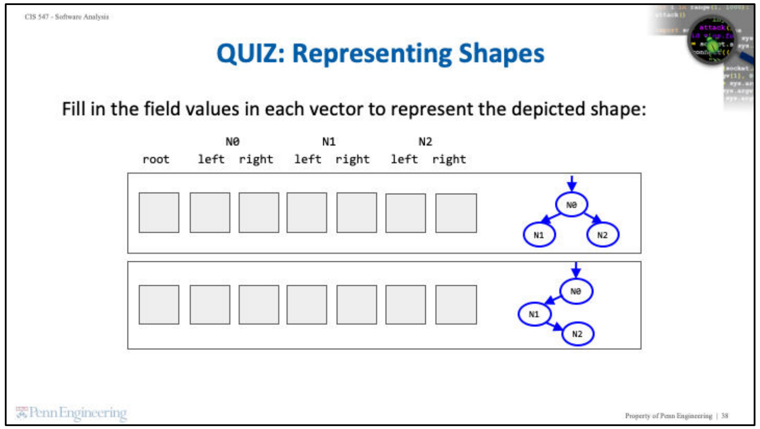
Before we see how Korat enumerates such shapes, let's look at the scheme that Korat uses to represent different shapes uniformly. This scheme lies at the heart of how Korat systematically and efficiently enumerates the shapes.

To represent shapes, Korat arbitrarily orders all possible values of each field, assigning each a unique ID, and it arbitrarily orders all fields into a vector. Each shape then is simply an assignment of values to fields in the vector.

Let's look at our BinaryTree example to elucidate this concept. Suppose we wish to generate all possible binary trees of up to 3 nodes.

Let's give unique identifiers N0, N1, and N2 to the three nodes. Then, the vector of fields contains 7 elements: a field for the root of the tree, and a field for the left and right child of each of the three nodes.

Each of the boxes can contain either a null, N0, N1, or N2. Each possible assignment of these values results in a different input shape to test. Let's do a quiz next to see how different assignments result in different shapes.



CIS 547 - Software Analysis

QUIZ: Representing Shapes

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

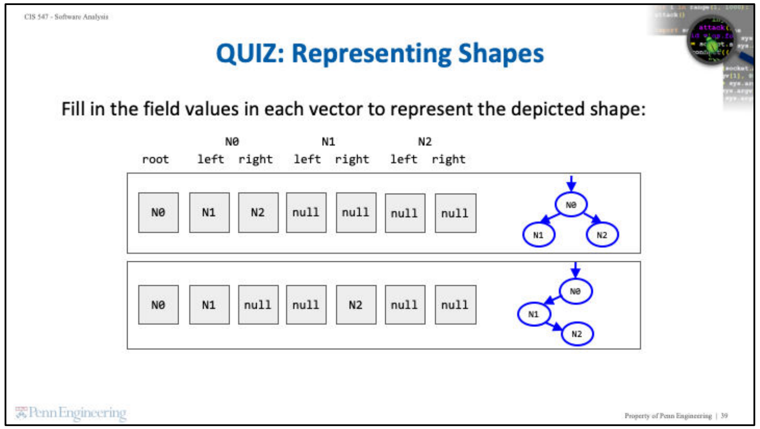
Penn Engineering

Property of Penn Engineering | 38

{QUIZ SLIDE}

This quiz is to ensure you understand how these vectors, called candidate vectors, correspond to concrete representations of objects.

Here are two possible shapes of BinaryTree objects. Fill in the boxes in the candidate vectors with the correct value for each field. (Remember that you can also enter the word "null" into a field as appropriate.)



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

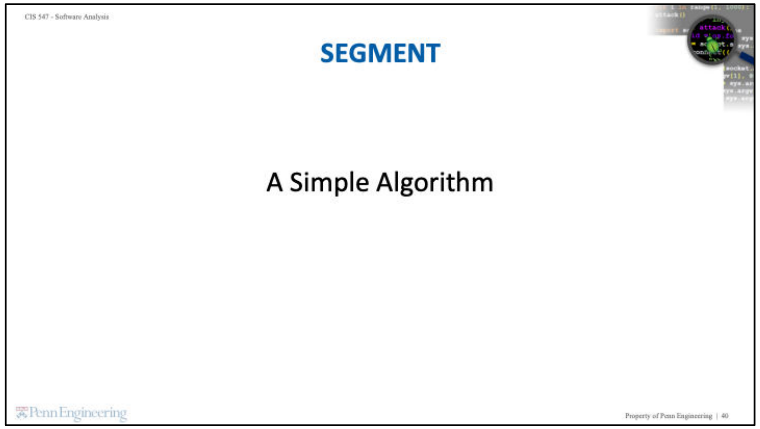
{SOLUTION SLIDE}

Let's look at the solution. For both trees, the root field points to node N0, and node N0's left field points to node N1.

In the first tree, however, N0's right field points to N2, while in the second tree, N0's right field is null (since it has no right child).

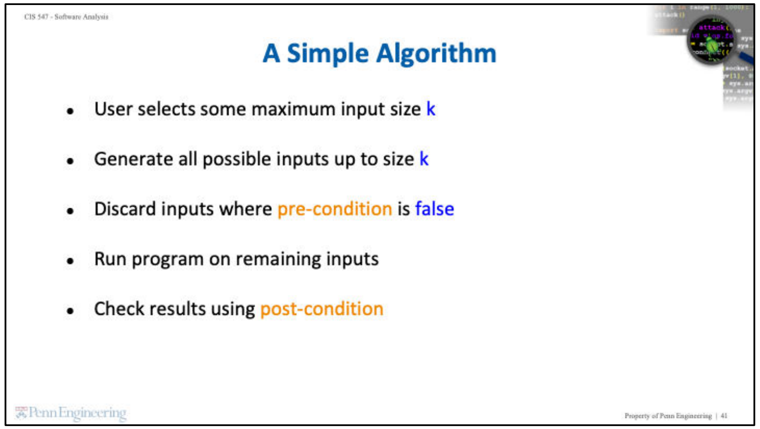
In the first tree, both N1 and N2 have no children, so their left and right fields are all null.

Finally, in the second tree, N1 has a right child, N2; since there are no other edges in the tree, the remaining fields are all null.



SEGMENT

A Simple Algorithm



CIS 547 - Software Analysis

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

Penn Engineering

Property of Penn Engineering | 41

Now that we have seen how Korat represents each possible shape, let’s look at how Korat enumerates them systematically.

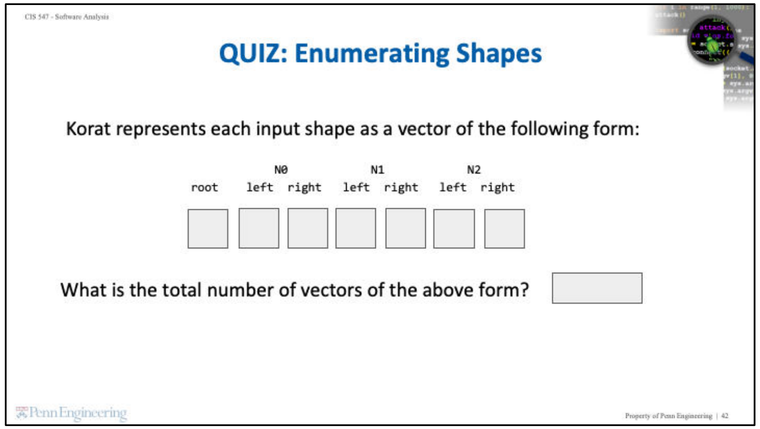
Here is the basic idea that Korat uses: the user selects a maximum input size, which we’ll call k. Then Korat generates all possible shapes of size at most k.

The pre-condition of the method under test is used to filter out invalid inputs: for example, we might not want to call a remove_root() method on an empty binary tree.

Those test inputs meeting the pre-condition are run through the method being tested, and the results are checked against the method’s post-conditions.

This algorithm should be more-or-less familiar: the same basic flow happened with randomly generated test cases. The only difference is the way in which test cases are generated.

Let’s take a closer look at this: after all, how hard could it be to generate all binary trees with, say, three nodes?



CIS 547 - Software Analysis

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? □

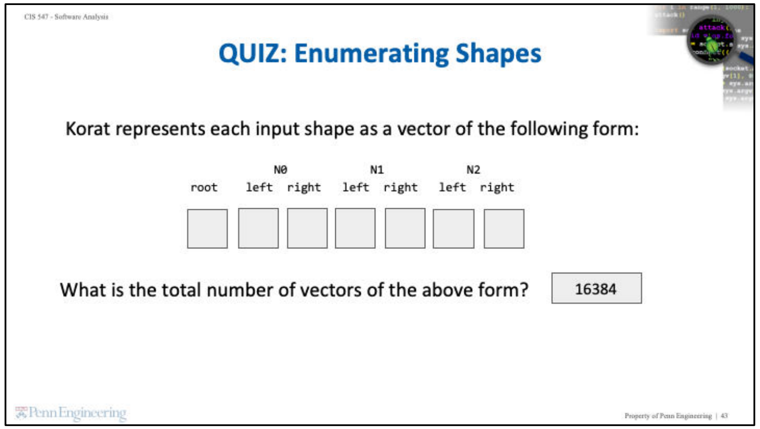
Penn Engineering

Property of Penn Engineering | 42

{QUIZ SLIDE}

Why don’t you take a shot at this question? Suppose we want to count how many possible shapes we could create by filling in the cells of a candidate vector: how many different candidate vectors are there?

Fill in your answer in the box at the bottom.



QUIZ: Enumerating Shapes

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

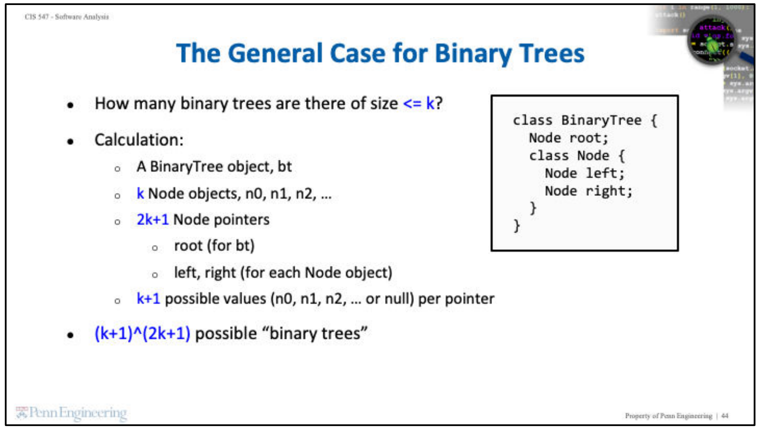
	N0		N1		N2	
root	left	right	left	right	left	right

What is the total number of vectors of the above form? 16384

{SOLUTION SLIDE}

Let's count the number of possible vectors we could consider. The root field could be null, N0, N1, or N2; so could N0.left, N0.right, N1.left, and so on.

Since there are seven fields with four choices each, that gives a total number of 4^7, or 16 thousand 384 candidate vectors. This is the size of the *state space* of the problem of generating a BinaryTree with three Node objects.



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;
  }
}
```

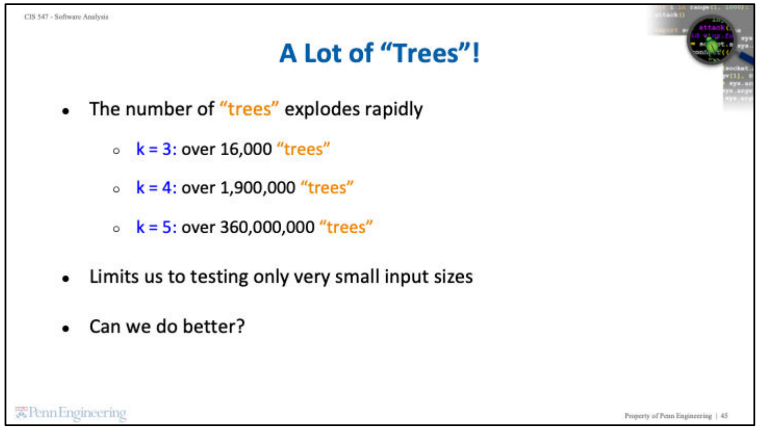
Let's generalize the setting in the quiz, from binary trees using up to 3 nodes, to binary trees using up to k nodes.

Let's calculate how many binary trees we can make using up to k nodes.

We have a single BinaryTree object, and k Node objects, which we'll call n0, n1, n2, etc. There are 2k+1 Node pointers to assign values to: the root field of the BinaryTree object, and the left and right field of each of the k Node objects. Also allowing a null pointer value, we have k+1 choices for each Node pointer: that is, each of the k Node objects or null.

Thus, we have (k+1)^(2k+1) "binary trees."

You can verify the solution to the quiz by plugging in the value 3 for k in this expression: we get 4^7, that is 16,384 binary trees using up to 3 nodes.



CIS 547 - Software Analysis

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?

Penn Engineering

Property of Penn Engineering | 45

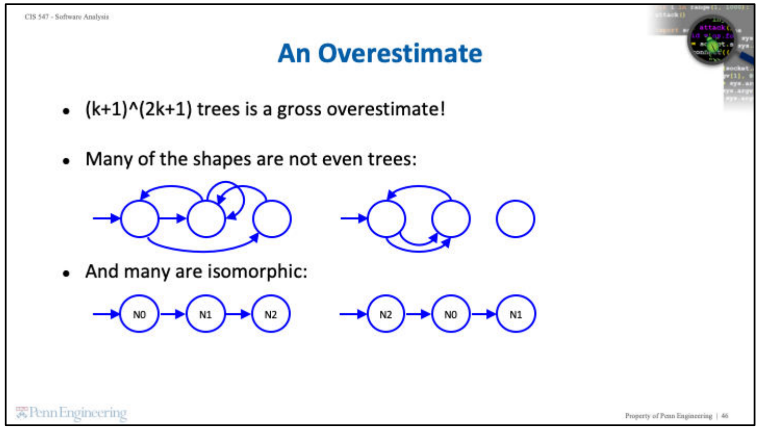
That’s a lot of trees! This quantity grows super-exponentially in k:

For three nodes, we have over 16 thousand possible Binary Trees that can be defined.

For four nodes, we have nearly 2 million possible Binary Trees.

And for five nodes, we have over 360 million possible Binary Trees.

Clearly this limits us to only very small test input sizes. Is there a way to do better?



CIS 547 - Software Analysis

An Overestimate

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

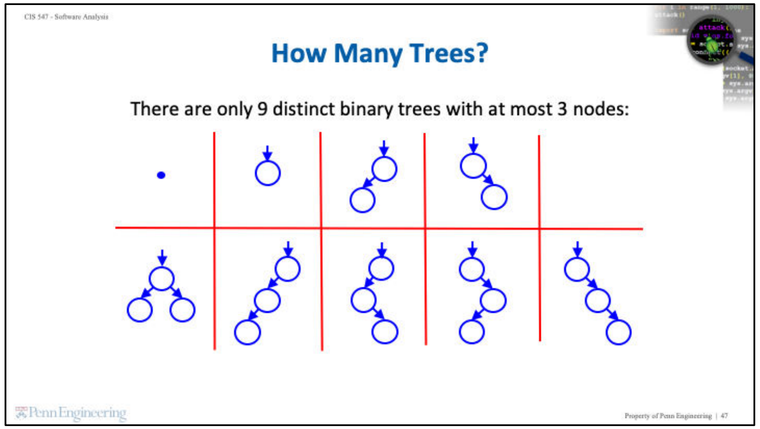
Penn Engineering

Property of Penn Engineering | 46

As you may have guessed, our count of (k+1)^(2k+1) trees is a gross overestimate of the number of BinaryTree objects we’d really be interested in testing.

Many of the structures that would be created by systematically filling in every possible value for left and right in the Node objects would be disconnected or contain cycles and therefore would not be trees.

Additionally, many of the trees that are created would be essentially the same: they would be isomorphic, indistinguishable for the purposes of testing.



CIS 547 - Software Analysis

How Many Trees?

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

PennEngineering

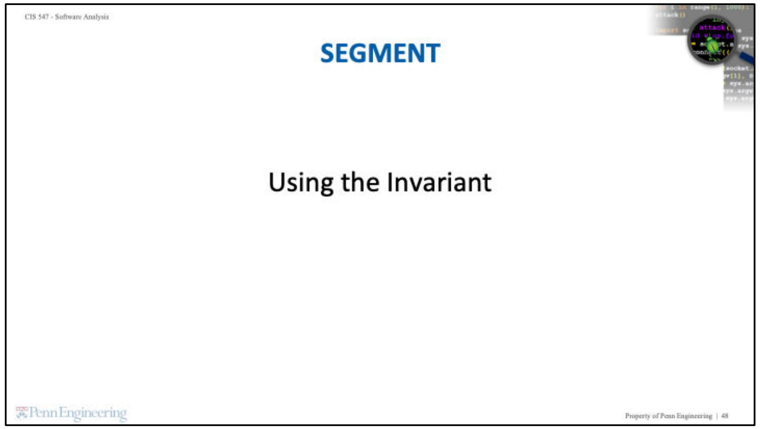
Property of Penn Engineering | 47

In fact, there are only nine non-isomorphic binary trees using at most three nodes:

One binary tree with a null root.
One binary tree with a single node as the root and no children.
Two binary trees with a root and one child (one on the left and one on the right).
One binary tree with a root and two children.
And four binary trees with a root, one child of the root, and one grandchild of the root.

To summarize, there are two central challenges: how to avoid generating illegal test inputs, which in this case are invalid trees, and how to avoid generating redundant test inputs, which in this case are isomorphic trees.

These are challenges for any automated test generation technique, not only Korat. We will see shortly how Korat addresses these two challenges.



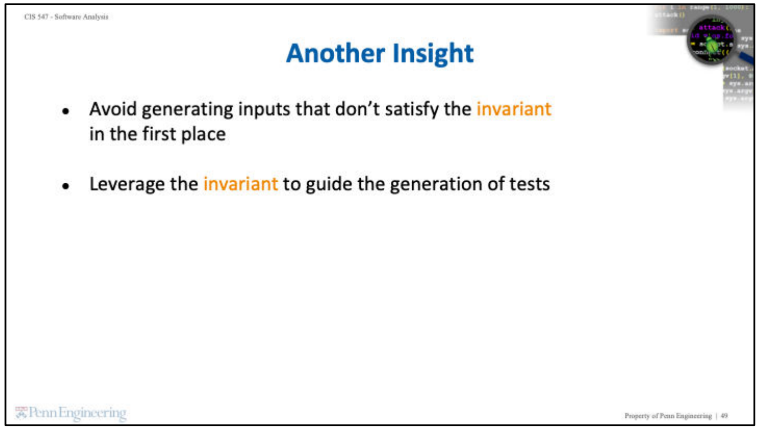
CIS 547 - Software Analysis

SEGMENT

Using the Invariant

PennEngineering

Property of Penn Engineering | 48



CIS 547 - Software Analysis

Another Insight

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

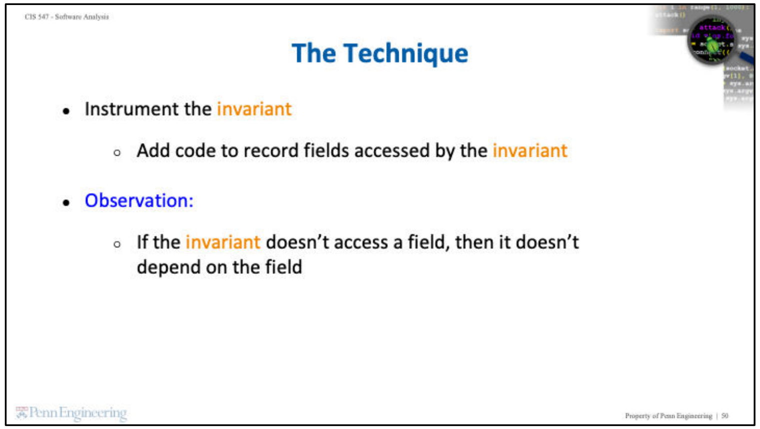
Penn Engineering

Property of Penn Engineering | 49

Let's focus on first effectively filtering out invalid trees.

It's inefficient to generate each candidate input and then check whether it actually satisfies the conditions of being a tree. Instead, we want to avoid generating candidate inputs that don't satisfy the invariant in the first place.

Let's look at how Korat uses the invariant as a guide in the generation of test inputs.



CIS 547 - Software Analysis

The Technique

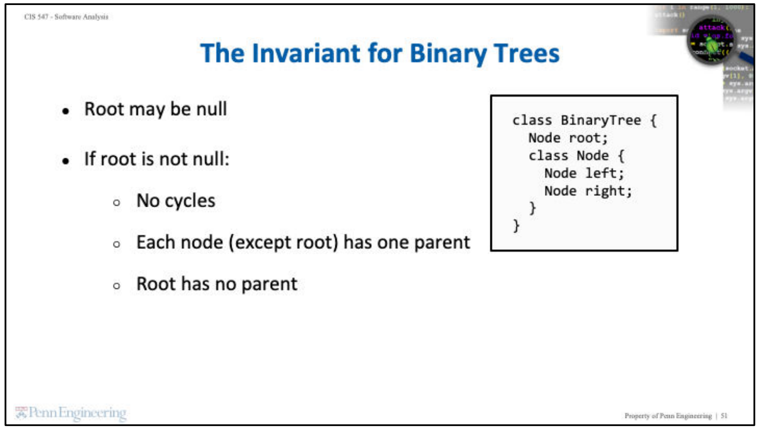
- Instrument the invariant
  - Add code to record fields accessed by the invariant
- Observation:
  - If the invariant doesn't access a field, then it doesn't depend on the field

Penn Engineering

Property of Penn Engineering | 50

The technique Korat uses is to instrument the invariant. That is to say, it adds code to the invariant checking to observe how it acts on the test input. In particular, it records which fields of the input that the invariant accesses.

A key observation is that if an invariant never accesses a field of the test input, then the result of the invariant doesn't depend on that field.



CIS 547 - Software Analysis

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;
  }
}
```

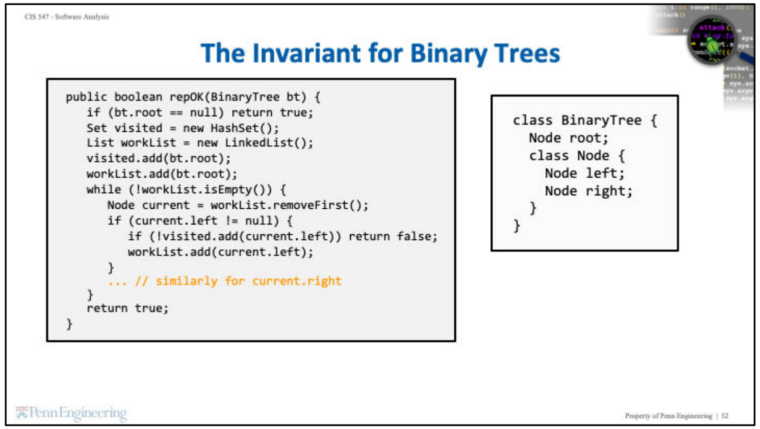
Penn Engineering

Property of Penn Engineering | 51

Recall that we looked at an example invariant that a data structure of this type should satisfy in order to be a valid binary tree.

For a data structure of this type [gesture to code] to be a valid binary tree, it should satisfy one of two different cases:

the root may be null
if the root is not null, then the structure should be a directed tree. That is:
It should have no cycles
Every node except the root should have one parent
And the root should have no parent



CIS 547 - Software Analysis

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;
  }
}
```

Penn Engineering

Property of Penn Engineering | 52

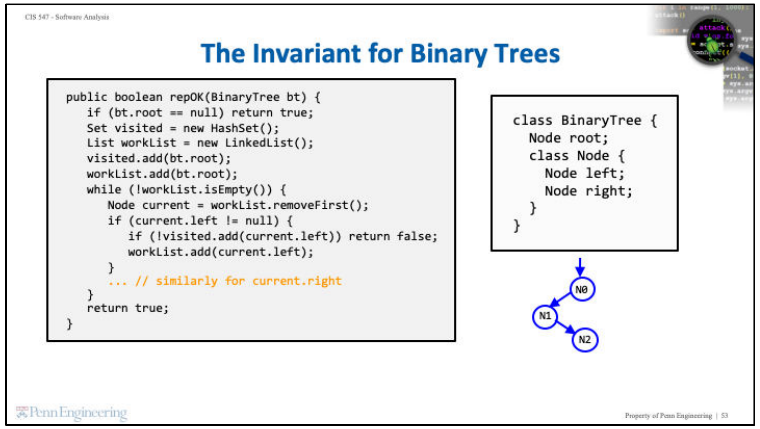
Let's look at a concrete implementation of the condition we just outlined for binary trees. It is a function called repOK in Korat terminology. It takes as input a BinaryTree object bt, and returns true if it is a valid binary tree, and false otherwise.

In addition to this requirement, the repOK function must be careful not to access unnecessary fields of the BinaryTree object bt. Recall that Korat will monitor which fields this function accesses. So, the fewer fields are accessed, the more Korat will be able to prune the space of candidate vectors that it considers.

The repOK function starts out by checking whether the root is null. If so, it exits by returning true, since an empty binary tree is a valid one. If the root is non-null, the function uses a classic approach to traverse all the nodes reachable from the root and determine whether the resulting structure is indeed a tree. It uses two data structures for this purpose: a hashset that keeps track of all nodes visited so far, and a worklist that keeps track of nodes that have themselves been visited but whose left and right children might not have been visited.

While the worklist hasn't been fully processed, the function removes the earliest node in the worklist, and checks whether its left child is non-null. If so, it then checks whether that left child has already been visited. If it has, then we have detected an invalid tree; in particular, we have detected more than one path from the root node to the current node, which violates the condition for a data structure to be a tree. If the left child wasn't previously visited, we add it to the visited set and also to the worklist.

We then do a similar check for the right child of the current node. If both these checks succeed for all nodes that are reachable from the root, then the repOK function returns true, indicating that the input BinaryTree object represents a valid binary tree.



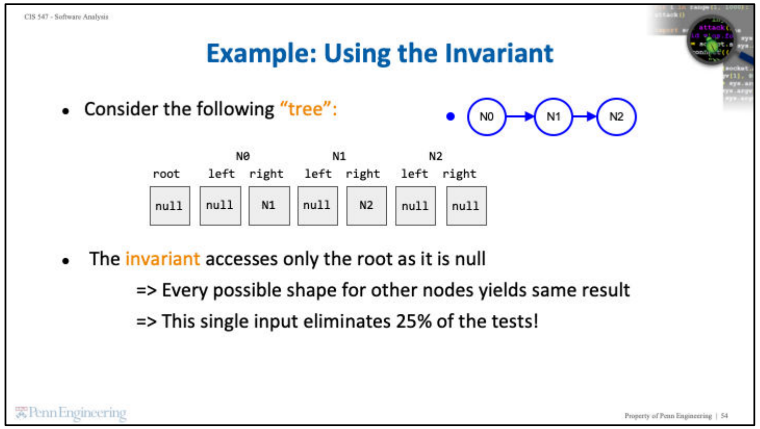
The Invariant for Binary Trees

```
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;
  }
}
```

Observe the order in which this function traverses nodes. This will be important in understanding how Korat systematically enumerates different shapes of binary trees. The order is breadth-first left-to-right order. [Animation appears]

Let’s look at an example to better understand this order. Suppose this BinaryTree object is input to the repOK function. We first visit the root field which leads us to node N0. We next visit the left child of N0, leading us to node N1. We next visit the right child of N0, which is null. We then visit the left child of node N1, which is null. We next visit the right child of N1, which leads us to node N2. We then visit the left and right children of N2, both of which are null. At this point, we are done visiting all left and right fields of all nodes reachable from the root field.



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

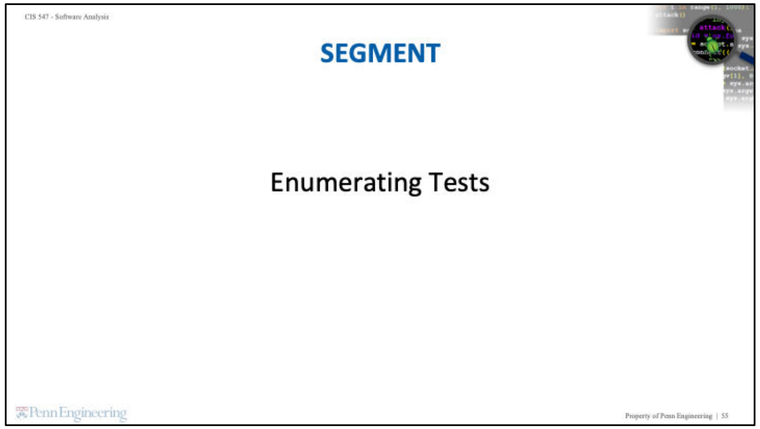
  => Every possible shape for other nodes yields same result

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

Let's see how we can use the invariant we just saw for the purpose of our test generation.

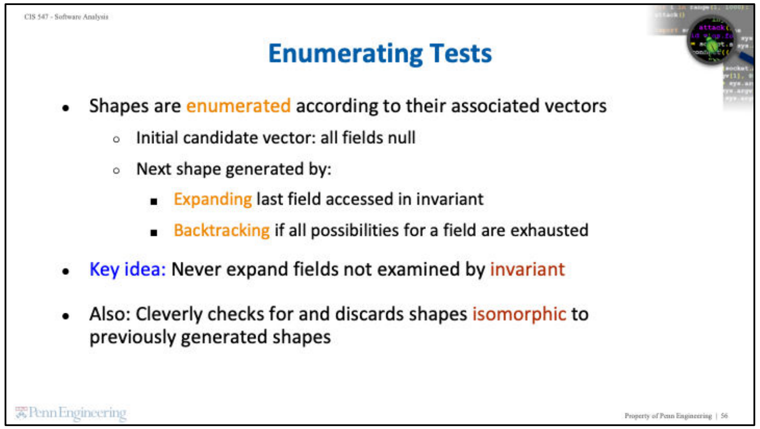
Consider the following candidate input which corresponds to the following BinaryTree object. Its root pointer is null, so there is no path to the Node objects N0, N1, or N2 that were also generated in this input.

For this input, the repOK function only accesses the root field because it is null. The repOK function therefore doesn't check any of the other fields. Indeed, any other shape for the remaining nodes would yield the same result, so there is no need to generate any other candidate input with a null root. This single input therefore eliminates 25% of the tests that would have been generated otherwise.



SEGMENT

Enumerating Tests



CIS 547 - Software Analysis

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

Penn Engineering

Property of Penn Engineering | 56

The general procedure Korat uses for generating tests is to enumerate the possible shapes by their associated vectors. The first vector generated is the vector with all null entries, corresponding to all fields being null. Then Korat checks that this shape satisfies the invariant; if so, then the shape is kept as a test case.

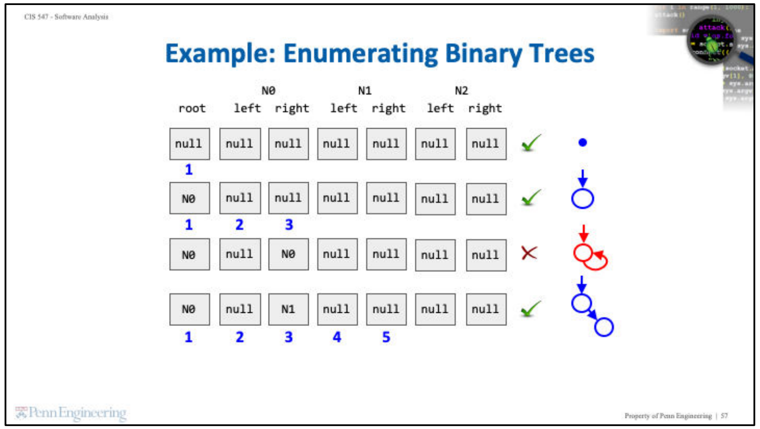
Next, Korat *expands* the last field that was accessed by the repOK function; In other words, the field is assigned to the next object that could potentially be assigned to that field.

Once all possible assignments for a field have been checked, then Korat backtracks in the following way: it resets that field to null, and then it expands the second-to-last field checked by the repOK function.

The key rule that lets Korat dramatically narrow its search space is that it never expands fields that are not examined by the repOK function. This is what allows Korat to skip all but one BinaryTree with a null root: the repOK function will stop at the root if the root is null, so it won't assign objects to any of the other fields while the root is null.

Additionally, by keeping track of which objects point to which types, Korat cleverly checks for whether it generates isomorphic (and therefore redundant) test cases, discarding any isomorphic test cases it finds.

For more details on the operation of the algorithm behind Korat, please see the paper: http://mir.cs.illinois.edu/marinov/publications/BoyapatiETAL02Korat.pdf



CIS 547 - Software Analysis

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			

Penn Engineering

Property of Penn Engineering | 57

This algorithm is a rather abstract manipulation of data structures, so let's take a look at how Korat would work in a concrete example to get a better idea of what it is doing. Let's see how Korat would generate test cases for a BinaryTree object with three Nodes.

Recall that the BinaryTree object itself has a root field that points to a Node, and each Node has a left and a right field that points to a Node. Candidate vectors for test cases will consist of seven cells, one for each of these seven fields.

We begin with all the cells in the candidate vector equal to null. This vector corresponds to the BinaryTree shape with a null root. We would then check that this shape passes the invariant for a BinaryTree object. While the check is running, let's write down the order in which repOK checks each field of the shape.

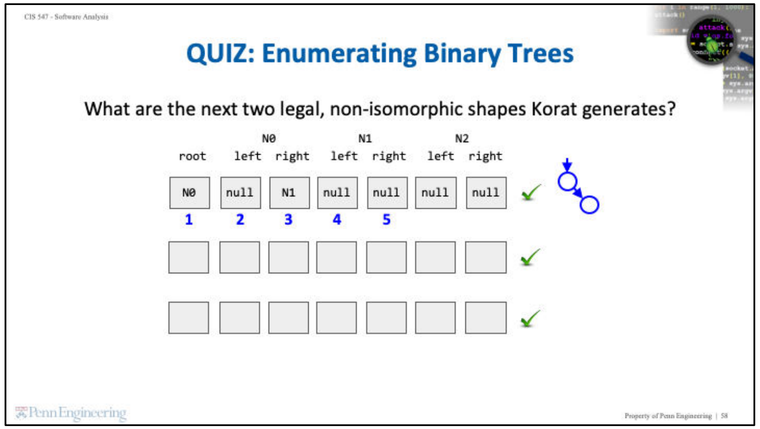
Recall that the first line of repOK returns true if the root is null. Since the check returns true, we accept this object as a test case. Additionally, the check will inspect just the root field before it returns, so the only field we will number is the root, which we number 1.

Now we expand the last field that was checked, which will be the root field. By expanding, we increment the root field to the next object in the shape that is compatible with that field, which is a node object. (Let's call the node objects N0, N1, and N2, and let's suppose that we'll increment in the order null -> N0 -> N1 -> N2.)

Now that we've expanded the root field, we have a shape consisting of the root pointing to a node which in turn has no children. Now we check that this shape passes the invariant. The order in which repOK accesses the fields is first the root, then the left child of the root, then the right child. The check will return true, so we accept this shape as a new test case.

Now, since N0.right is the last field checked by repOK, we expand the N0.right field. This will result in N0.right changing from null to N0. But this shape contains a cycle, so it will fail the invariant check for a BinaryTree object. Therefore we discard this shape and expand N0.right again, assigning to it the next node in our order, which is N1.

This new shape is indeed a binary tree, so it will pass the invariant check. Recall that repOK traverses the shape in a breadth-first manner, choosing the left child before the right child, so the ordering of the field accesses will be 1, 2, 3, 4, 5. We would then use this ordering to determine the field to expand for the next iteration.



CIS 547 - Software Analysis

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			
							✓
							✓

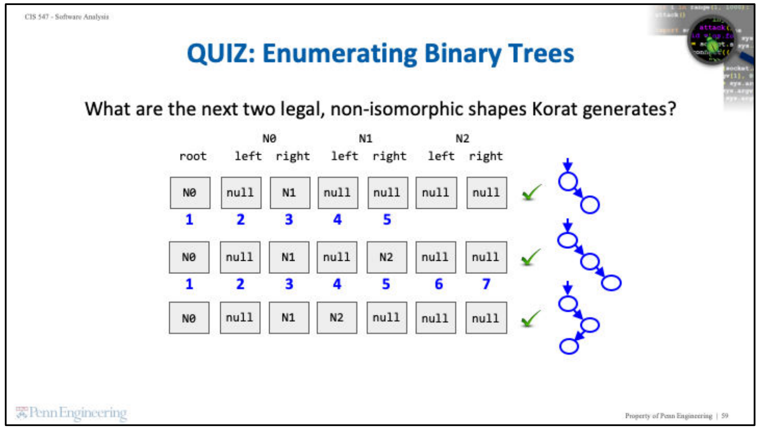
Penn Engineering

Property of Penn Engineering | 58

{QUIZ SLIDE}

To check your understanding of the expand operation in Korat, fill in the candidate vectors for the next two shapes that Korat will generate. Omit any illegal shapes (that is, those that fail the repOK check) and any shapes that are isomorphic to any previously generated shapes.

As a hint, here is the ordering in which repOK inspected each field: first the root, then N0.left, then N0.right, then N1.left, and finally N1.right.



CIS 547 - Software Analysis

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	✓

Penn Engineering

Property of Penn Engineering | 59

{SOLUTION SLIDE}

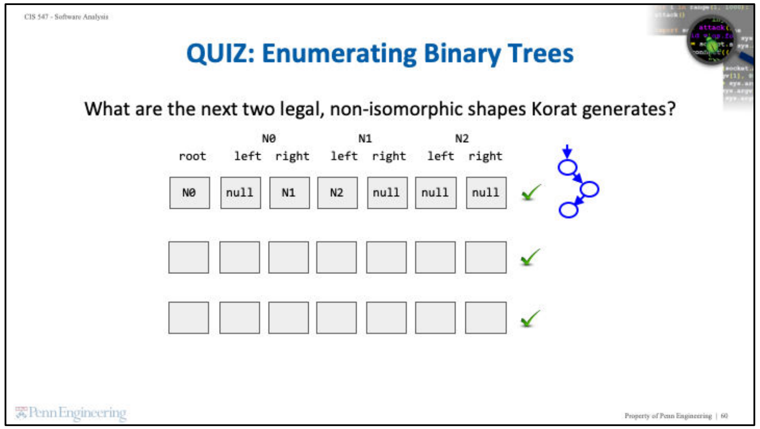
Remember that the next field to be expanded will be the last one checked by repOK. In this case, we will expand N1.right. Korat would first assign N0 and N1 to N1.right, but these would result in illegal shapes. So the next legal shape to be generated would be when N1.right equals N2.

Now, because repOK traverses the tree breadth-first, visiting left children first, it would visit each field in order from left to right. So the next field to be expanded is N2.right. But notice that setting N2.right to anything other than null would result in an illegal shape. Thus, we backtrack for the first time and set N2.right to null.

Since N2.left is the second-to-last field checked by repOK, we start attempting to expand N2.left. But, again, assigning anything other than null to N2.left results in an illegal shape. So Korat discards all these test cases, sets N2.left to null, and backtracks to N1.right.

Since N1.right is N2 (the last Node object), there are no more Nodes that can be assigned to the field. Therefore, we set N1.right to null and backtrack again to N1.left.

Setting N1.left to N0 results in an illegal shape, as does setting it to N1. Finally, assigning N2 to it gives us a legal shape, pictured here. This is the next legal shape that Korat generates.



CIS 547 - Software Analysis

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	✓
							✓
							✓

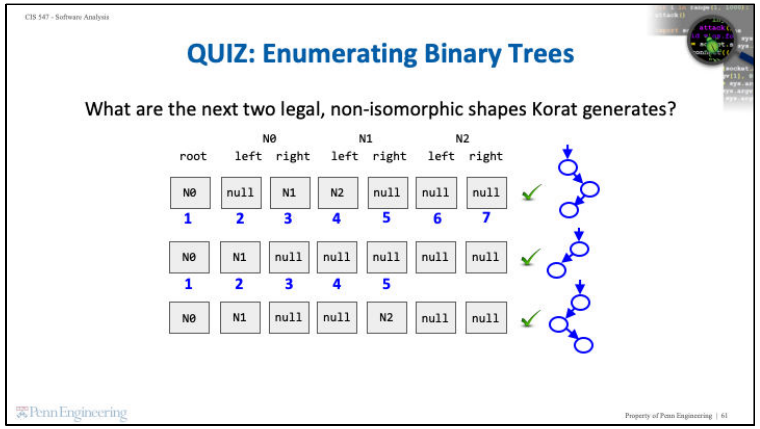
Penn Engineering

Property of Penn Engineering | 60

{QUIZ SLIDE}

Let's do one more quiz to solidify your understanding of the Korat algorithm.

Starting from where we left off in the previous quiz, enter the candidate vectors of the next two legal shapes that Korat generates. Remember to disregard any shapes that are isomorphic to a previously generated shape.



CIS 547 - Software Analysis

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

Penn Engineering

Property of Penn Engineering | 61

{SOLUTION SLIDE}

We start by determining the order in which the repOK accesses each field. Since repOK traverses the tree in a breadth-first manner (left before right), it first accesses the root N0, then it accesses N0.left, then N0.right (which is N1), then it accesses N1.left (which is N2), then N1.right, then N2.left, and finally N2.right.

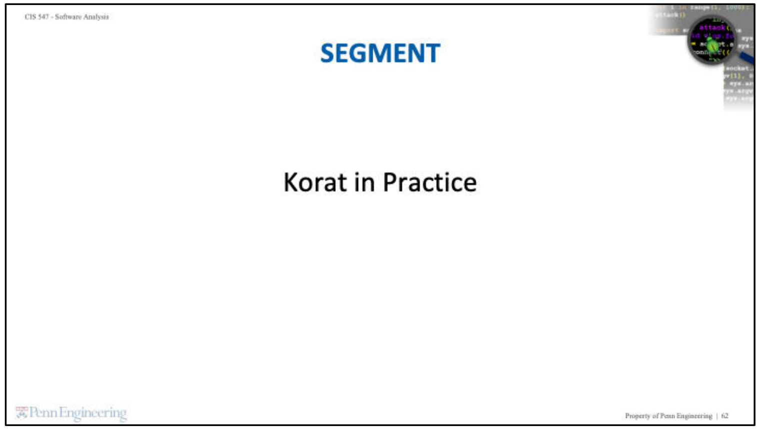
Therefore, expansion begins at N2.right. Notice that assigning any non-null value to N2.right would result in an illegal shape; similarly for N2.left and N1.right. Thus, Korat would set all those fields to null and backtrack to N1.left, which it would then expand. However, there are no Nodes after N2, so we have to backtrack again to N0.right.

Expanding N0.right would increment the node in the field to N2 instead of N1. However, this would result in a tree consisting of a root with a single right node, isomorphic to a shape we've already generated.

Thus, we try to expand N0.right again, with no luck (since N2 is the last node), so we backtrack to N0.left. Expanding N0.left once would give us N0, but that's illegal because N0 can't point to itself. Expanding it again would give us N1, and this (finally), is a new, legal shape.

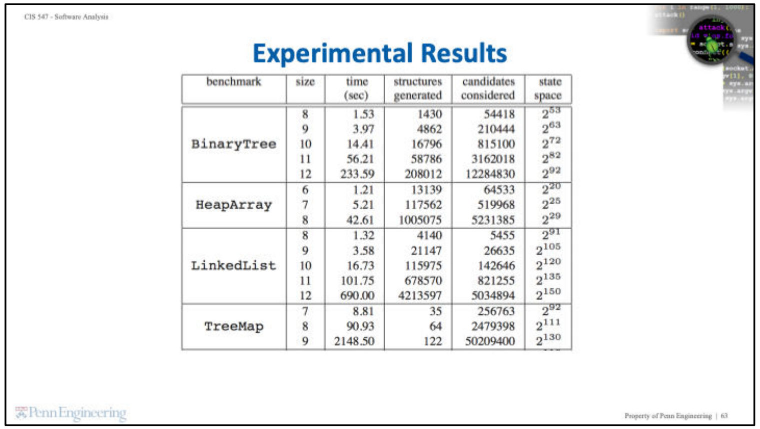
repOK accesses the fields of this shape in the order: root, N0.left, N0.right, N1.left, N1.right. So we expand N1.right first. The only non-null value we can assign to it is N2 (other values would result in an illegal tree), and this gives us the following binary tree with three nodes, a new, legal shape.

So far we've generated 7 non-isomorphic test cases; we could continue this process until the last two trees with at most 3 nodes are generated, but we will stop here for now.



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}

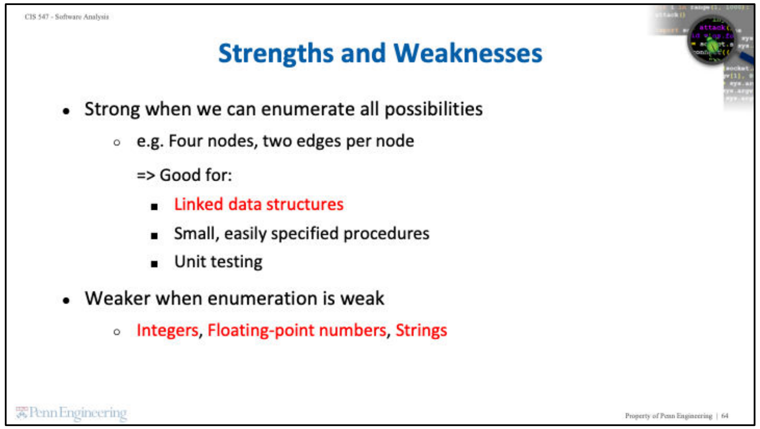
As we've seen, with relatively few steps, Korat generates all the legal and isomorphically distinct test cases for binary trees with at most 3 nodes.

In practice, Korat's scheme for eliminating illegal and redundant test cases have proven to be highly effective. Here, you can see that for BinaryTree objects, for each node in the tree, the full state space grows by a factor of about 1000. But Korat is able to reduce its search space to about 4 times as many candidates considered for each additional node, a massive speedup.

Similarly for HeapArray objects: the state space grows by a factor of 32 for each new node while the number of candidates considered grows by only a factor of about 10.

And for linked lists: the state space grows by a factor of 32000 per new node, but the number of candidates considered by Korat only grows about sixfold per new node.

The time taken to run these tests is on the order of minutes or even seconds, so Korat could be easily included into regression tests to ensure that code changes don't break previously established functionality.



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

Let's wrap up our discussion about Korat by summarizing its strengths and weaknesses.

Korat's test generation is at its best when we can easily enumerate all possibilities for the objects to be used in the test cases: for example, if our test cases all involve four nodes, and each node has two pointers to other nodes.

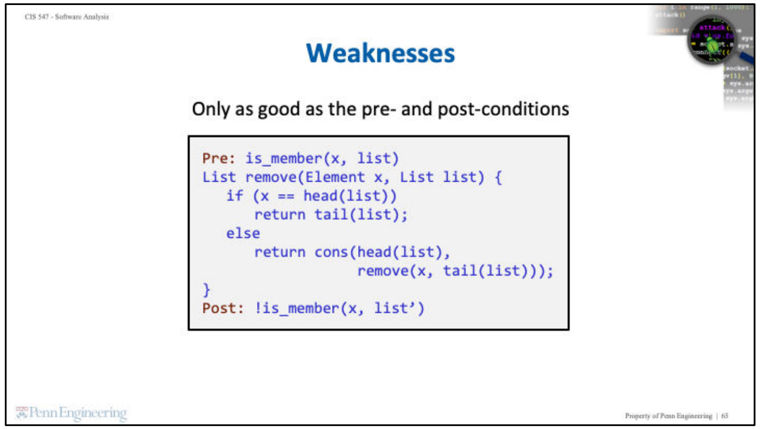
More specifically, Korat is useful for testing linked data structures with small, easily specified procedures (such as emptying a stack or inserting an element into a heap). For this reason, Korat is a strong candidate for unit testing components of larger systems.

Korat is weaker when it is harder to enumerate the set of all possible test cases. For example:

the set of all integers (at least those representable in a 4-byte format on a computer),
the set of all single- or double-precision floating-point numbers, or
the set of all n-character strings with ASCII or Unicode characters.

In these cases, while enumeration can be done, even small test case sizes (one int, one double, or strings with 10 characters) lead to large numbers of test cases that Korat will not flag as isomorphic (because there are no links in the data to exploit in the culling process that occurs through pre-condition checking).

We will see a different technique later in the course, called dynamic symbolic execution, that is capable of handling such data types.



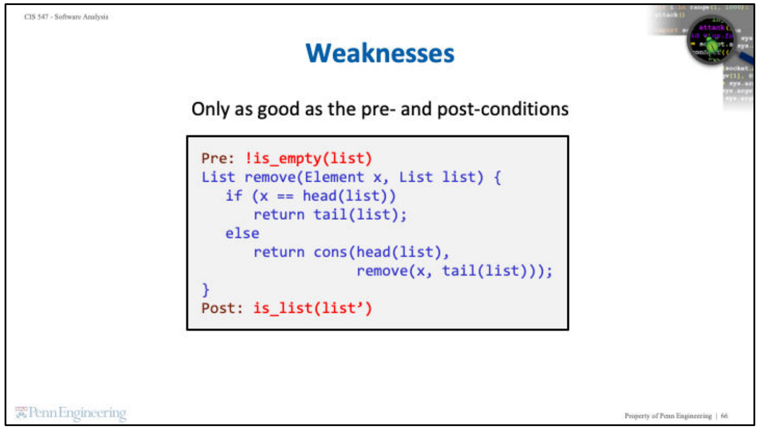
Weaknesses

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’)
```

Another weakness of Korat is that it is only as good as the pre- and post-conditions we specify for the methods being tested.

In the example code presented here, we want to test a method that removes an Element object from a List object. The pre-condition is that the element is a member of the list, and the post-condition is that the element is not a member of the list. These are sufficient conditions to ensure that the remove() method is functioning properly.



CIS 547 - Software Analysis

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')
```

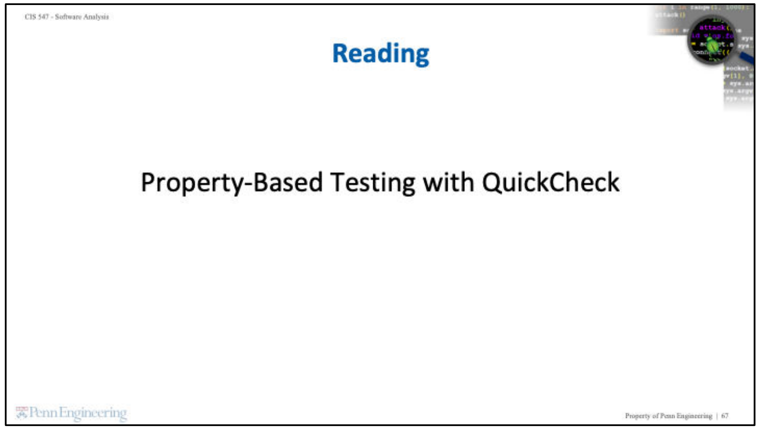
Penn Engineering

Property of Penn Engineering | 66

If we are not as careful in our selection of pre- and post-conditions, though, we may end up with many useless tests or miss useful tests. Note that the pre-condition shown here, that the list object is not empty, is weaker than the earlier pre-condition that x be a member of list: whenever x is a member of the list, the list will be nonempty. So we may end up with many useless tests that satisfy this weaker pre-condition but not the earlier pre-condition.

Conversely, Korat might not expand the element x because it is not accessed during this pre-condition, resulting in missing useful tests. In other words, there may be bugs in the method dependent on the value of x that wouldn't be found by Korat using this pre-condition.

Likewise, checking that the output of the remove() method is still a list is weaker than the earlier post-condition, which stated that the output is a list that no longer contains x. Since this post-condition does not check that the list no longer contains x, it will silently succeed if the remove() method has a bug that fails to remove x while at least preserving the list structure.



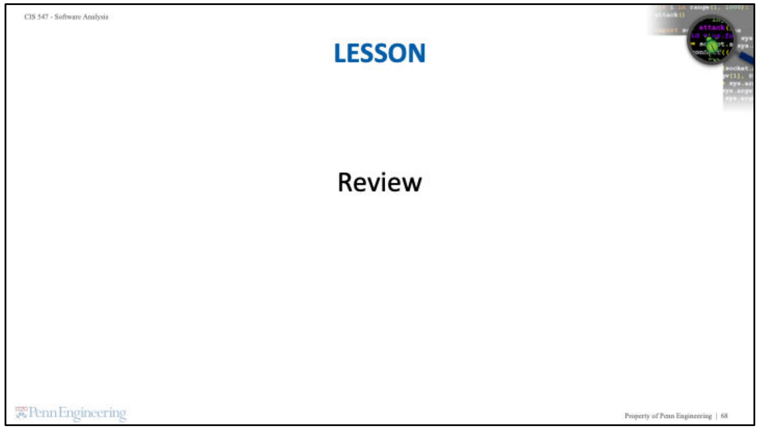
CIS 547 - Software Analysis

Reading

Property-Based Testing with QuickCheck

Penn Engineering

Property of Penn Engineering | 67



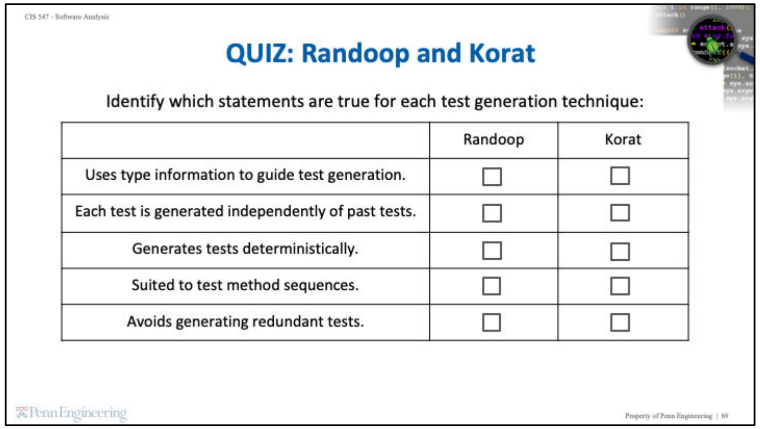
CIS 547 - Software Analysis

LESSON

Review

Penn Engineering

Property of Penn Engineering | 68



CIS 547 - Software Analysis

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.	□	□

Penn Engineering

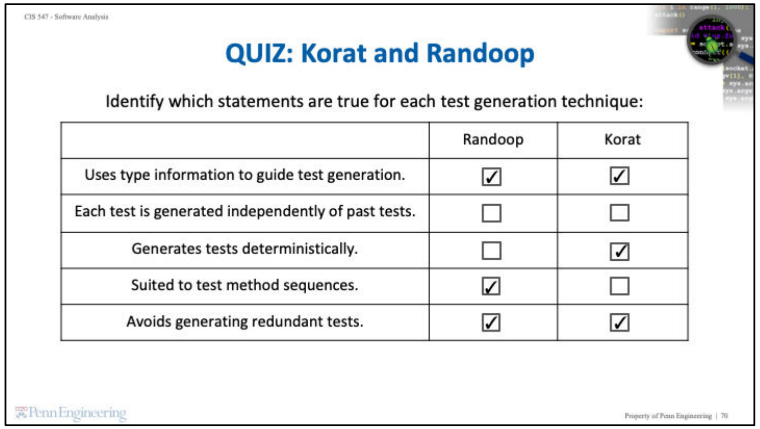
Property of Penn Engineering | 69

{QUIZ SLIDE}

As we begin to wrap up this module, let's take a moment to compare and contrast the two tools we have seen for automated test generation.

Identify which of the following statements are true for each of Randoop and Korat by checking the appropriate boxes.

Which of the techniques uses type information to guide test generation?
Which of the techniques generates new tests fully independently of past tests?
Which of the techniques generates tests deterministically?
Which of the techniques is suited to test sequences of methods?
And which of the techniques avoids generating redundant tests?



QUIZ: Korat and Randoop

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.	☑	☑

{SOLUTION SLIDE}

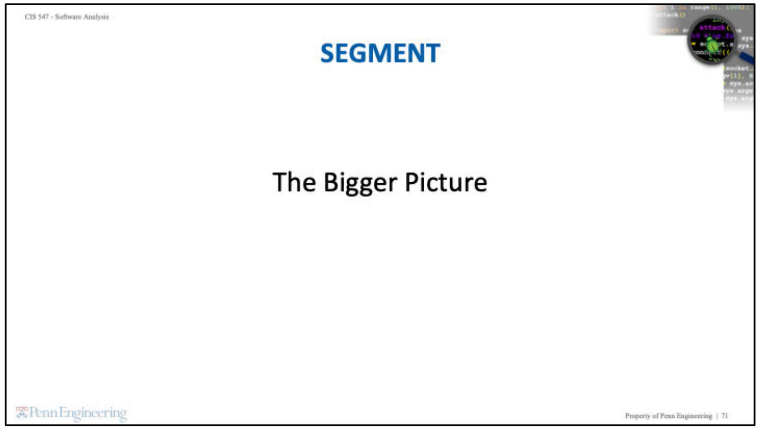
Both Randoop and Korat use type information to guide their test generation: Randoop uses types to determine which components to use in its sequence generation while Korat uses types to enumerate test cases and check for isomorphism.

Neither technique generates a test independently of past tests: this is a characteristic of pure random testing.

Korat is a purely deterministic test generation technique, while Randoop uses an element of randomness, at least in principle. In practice, however, the Randoop implementation is deterministic because it always starts with the same random seed, which is useful for replaying past runs of Randoop.

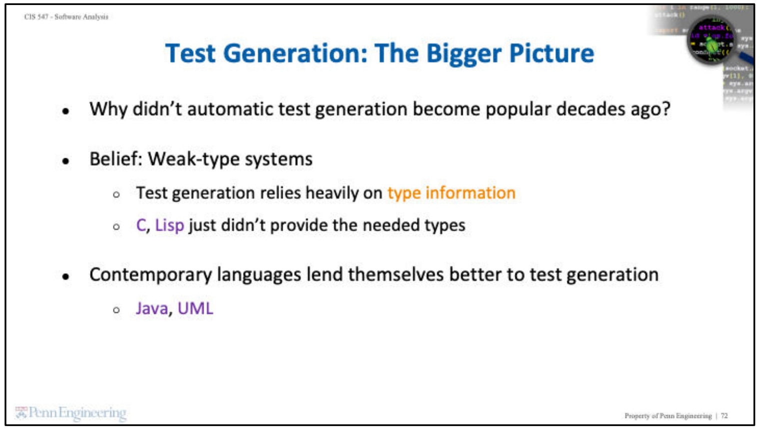
Randoop is suited for generating sequences of methods from an API, but recall that Korat is more suited to generating data structures (which may later be subjected to a sequence of calls from an API).

Finally, both tools avoid redundancy: Randoop does this through the heuristic of discarding test cases where the last object created had already been created in a previous test case, while Korat achieves this by checking newly generated structures for isomorphisms with previously generated structures.



SEGMENT

The Bigger Picture



Test Generation: The Bigger Picture

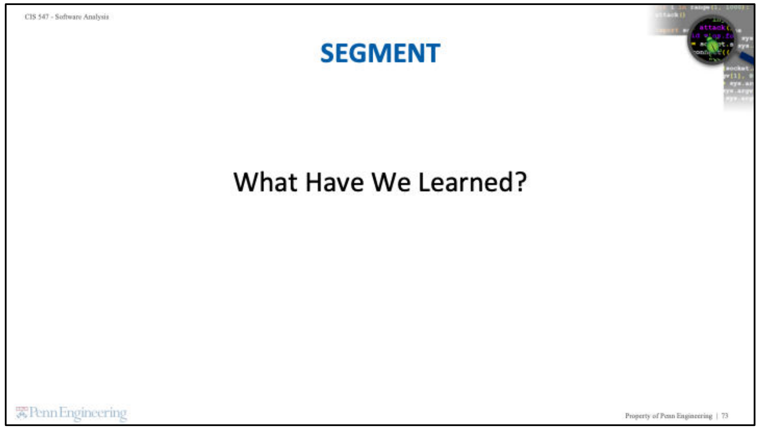
- Why didn't automatic test generation become popular decades ago?
- Belief: Weak-type systems
  - Test generation relies heavily on type information
  - C, Lisp just didn't provide the needed types
- Contemporary languages lend themselves better to test generation
  - Java, UML

In this module, you've learnt about two powerful test generation techniques in detail. Before we conclude, let's take a moment to step back and look at the bigger picture of test generation.

One question that you may have considered in this lesson: both Korat and Randoop were tools invented in the 2000's, but why didn't automatic test generation become popular decades ago?

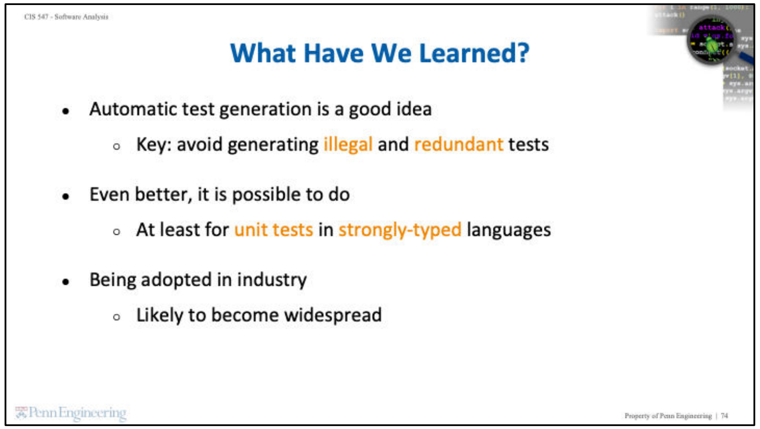
The most prominent reason for this delay is that earlier programming languages had weak-type systems, whereas automated techniques rely heavily on type information. Languages such as C and Lisp just did not provide the needed types.

Contemporary, strongly typed languages such as Java and UML lend themselves better to test generation, because automated tools can exploit type information to eliminate illegal and redundant test cases and effectively prune the space of candidate tests.



SEGMENT

What Have We Learned?



CIS 547 - Software Analysis

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

Penn Engineering

Property of Penn Engineering | 74

Let's wrap up with a summary of what we learned in this module.

Automatic test generation is a good idea: it helps find bugs quickly, and it does not require writing or maintaining tests. The key to making it work efficiently is to avoid generating illegal and redundant tests. If one is not careful, these kinds of tests dominate the test generation process and make it ineffective.

Even better, automatic test generation is possible to do today, at least for small units of a software system such as a data structure or a library API that is written in a strongly typed language like Java.

Finally, automatic test generation is being adopted in industry and is likely to become widespread in future, especially in targeting specific kinds of bugs such as security bugs or concurrency bugs, and specific classes of programs such as mobile apps or device drivers.