CS 569
Selected Topics in Software Engineering: Program Analysis & Evaluation

Automated Test Generation

Oregon State University, Winter 2024

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

Outline

- Previously: Random Testing (Fuzzing)
  - Security, mobile apps, concurrency
- Feedback-directed random testing: Randoop
  - Classes and libraries
- Systematic testing: Korat
  - Linked data structures
- Test generation using natural language specification: Swami
  - JavaScript compilers

Key Idea

- Using specifications to guide test generation
  - Types
  - Invariants
  - Pre- and Post-Conditions
  - Natural language specifications

Example: Using 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 inputs

Example: Using Invariants

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

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

Example: Using 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;
}
```

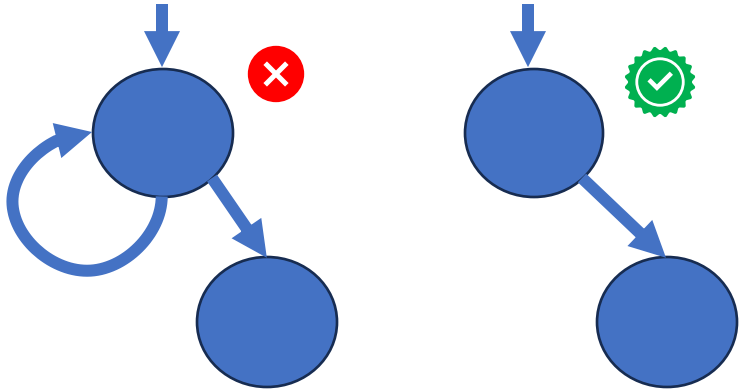
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

Checks if a binary tree object is valid (satisfies invariant)

Example: Using Invariants

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

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



Example: Using 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 tree structures that do not satisfy invariant
- Also serves as a contract to check that resultant tree at the end of remove method is valid binary tree

Example: Using 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 program execution states on entry and exit of remove method

Example: Using Natural Language Specifications

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to **`"Array"`**. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the **`length`** property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Describes how Array constructor should be implemented in JavaScript compiler

```
function test_array_len( len ){
      if ( ToUint32(len)!=len) {
         try{
              var output = new Array ( len );
              return;
         }catch(e){
              assert.strictEqual(true, (e instanceof RangeError));
              return;
         }
       }
}

/*TEST INPUTS*/

test_array_len(1.1825863363010669e+308);
test_array_len(null);
test_array_len(-747);
test_array_len(368);
```

Test oracle

Test inputs

Randoop: Feedback-Directed Random Testing

- Random test generation for object-oriented programs (Java, C#)
- Generates unit tests for methods of Java classes
- Motivating examples of generated tests

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

Tests generated for Java HashSet class provided in Java Util library

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

Only difference => **redundant test**

Motivating Examples

- Three randomly generated tests for Date class:

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Violates pre-condition

Motivating Examples

- Three randomly generated tests for Date class:

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Illegal tests

Randoop: Feedback-Directed Random Testing

IDEA

- Guide randomized creation of new test inputs by feedback about execution of previous inputs
  - Avoid redundant inputs
  - Avoid illegal inputs

- Test input = sequence of method calls
- Software under test: Classes in Java-like programming languages

Randoop: Feedback-Directed Random Testing

EXAMPLE (from Randoop paper that exposed bug in Java Utils library)

No contracts are violated in all method calls

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

Fails because of bug in JDK implementation

Randoop: Feedback-Directed Random Testing

APPROACH

- Build test inputs incrementally
  - New test inputs extend previous ones
- As soon as test input is created, execute it
- Use execution results to guide input generation
  - Away from redundant or illegal method sequences
  - Toward sequences that create new object states

Randoop: Feedback-Directed Random Testing

ALGORITHM

- Input:
  - Classes under test
  - Time limit
  - Set of contracts
    - Method contracts, e.g., o.hashCode() throws no exception
    - Class invariants, e.g., o.equals(o) == true
- Output:
  - Test cases with assertions

Randoop: Feedback-Directed Random Testing

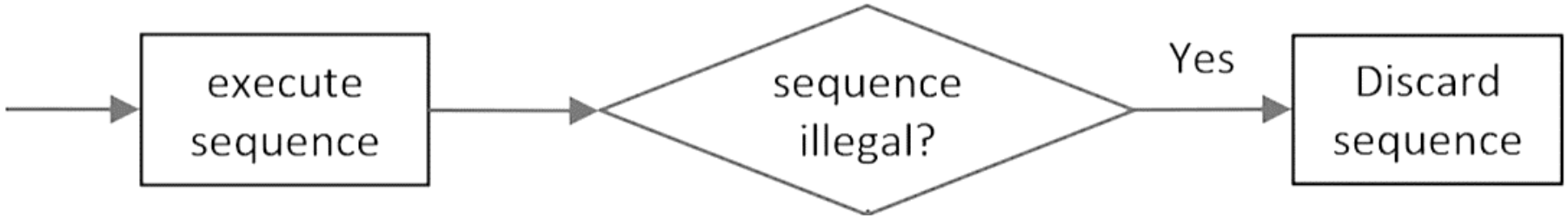
ALGORITHM

- Initialize seed components: int *i=0;* Boolean *b=false;* …
- Do until time expires:
  1. Randomly pick a method to test
     $C.m(T_1, \ldots, T_k)/T_{ret}$, where C = class under test, m = method under test, $(T_1, \ldots, T_k)$ = method arguments' types, and T_{ret} = method return type
  2. For each argument of type T_i, randomly pick a sequence S_i from the components that constructs a value v_i of type T_i
  3. Create new sequence
     $S_{new} = S_1;\ldots; S_k; T_{ret}\ v_{new} = C.m(v_1, \ldots, v_k)$
  4. If S_{new} was previously created (lexically), go to step 1
  5. Else classify the sequence S_{new}
     - May discard, output as test case, or add to components

New method call added to previously created sequence

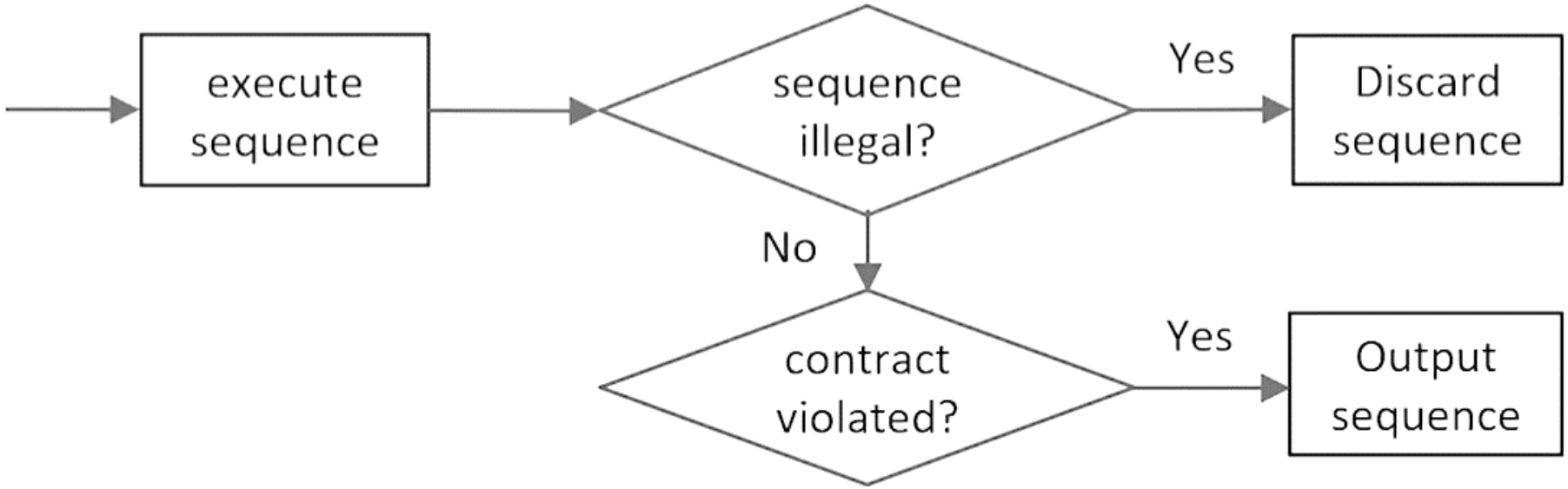
Randoop: Feedback-Directed Random Testing

Classify the sequence



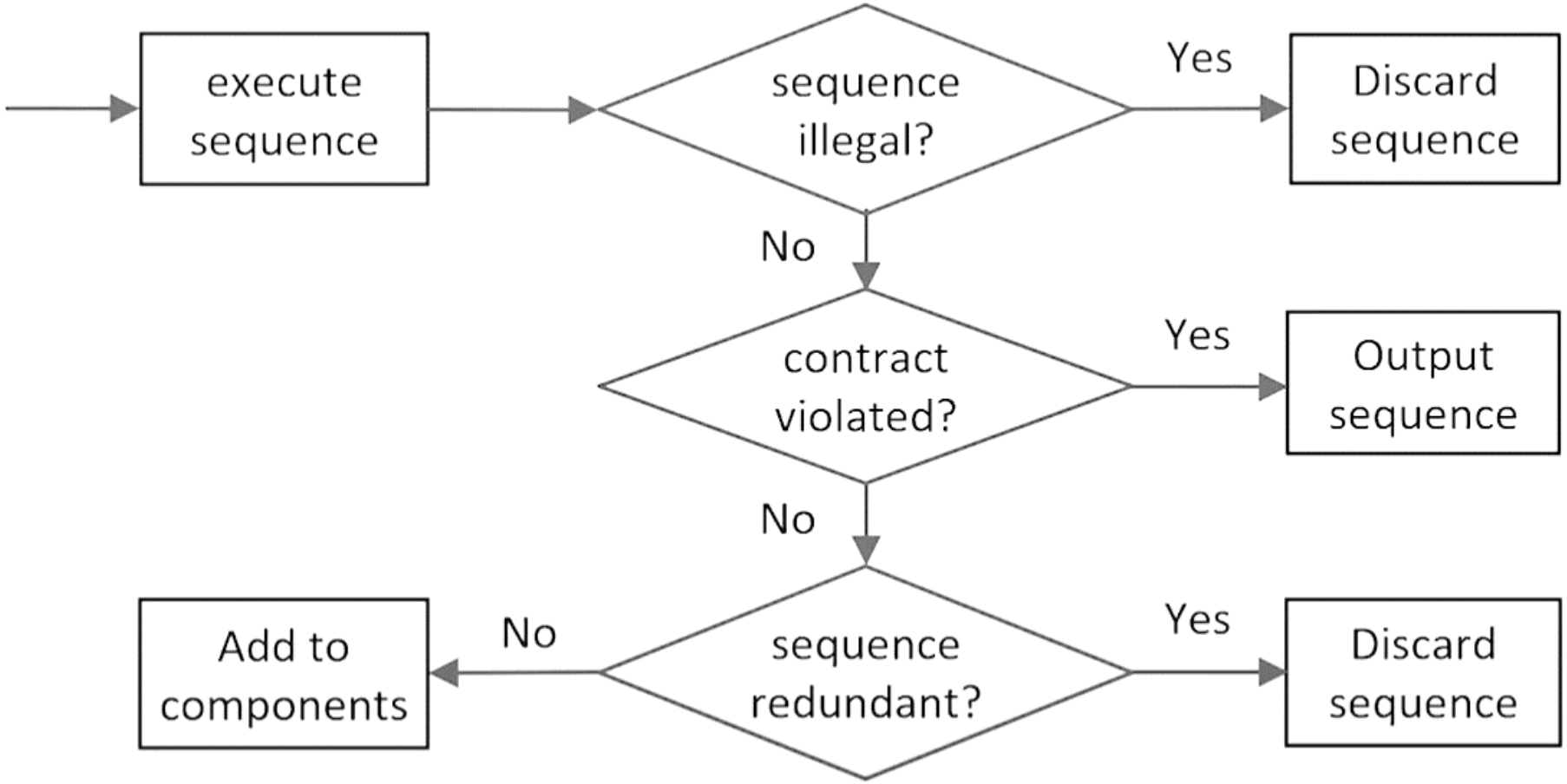
Randoop: Feedback-Directed Random Testing

Classify the sequence



Randoop: Feedback-Directed Random Testing

Classify the sequence



Randoop: Feedback-Directed Random Testing

How to detect Redundant Sequences?

- During sequence-generation and execution, maintain a set of all objects created
- Sequence is redundant if all objects created during its execution are in the maintained set
- Use `equals()` to compare objects for efficiency reasons.
- Could also use more sophisticated state equivalence methods that are more computationally expensive
  - E.g., heap canonicalization

Some Errors Found by Randoop

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

QUIZ(1/2): Randoop Test Generation

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(1/2): Randoop Test Generation

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

```
Node n = null;
BinaryTree t = new BinaryTree(n);
```

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(2/2): Randoop Test Generation

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

Once generated, how does Randoop classify it?

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

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

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

QUIZ(2/2): Randoop Test Generation

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

```
Node n = null;
BinaryTree t = new BinaryTree(n);
t.removeRoot();
```

Once generated, how does Randoop classify it?

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

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

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

Korat

- Deterministic test-generation approach suitable for linked data-structures
- Idea
  - Use pre-conditions and post-conditions to generate tests automatically
- 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 small such test
- If a list function works for the lists of length 0 through 3, probably it will work for all list sizes
  - E.g., because the function is oblivious to length (recall bug depth for concurrency bugs – Cuzz)

How Do We Generate All Test Inputs?

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

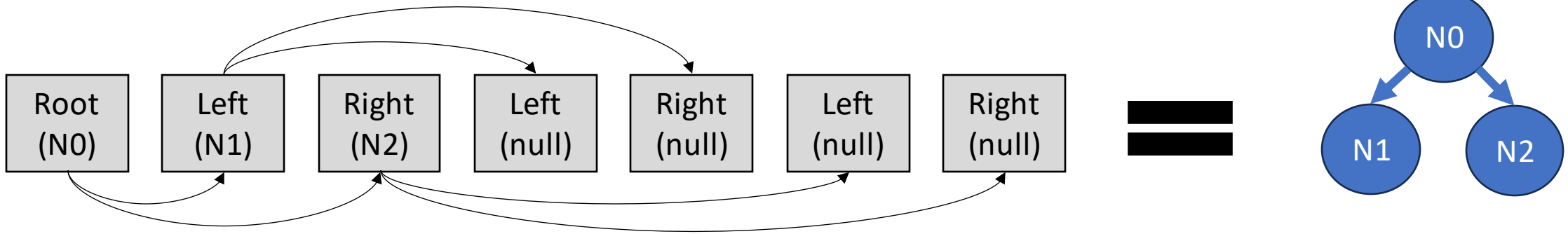
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

Scheme for Representing Shapes

- Order all possible values of each field assigning each order a unique ID
- 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;
  }
}
```

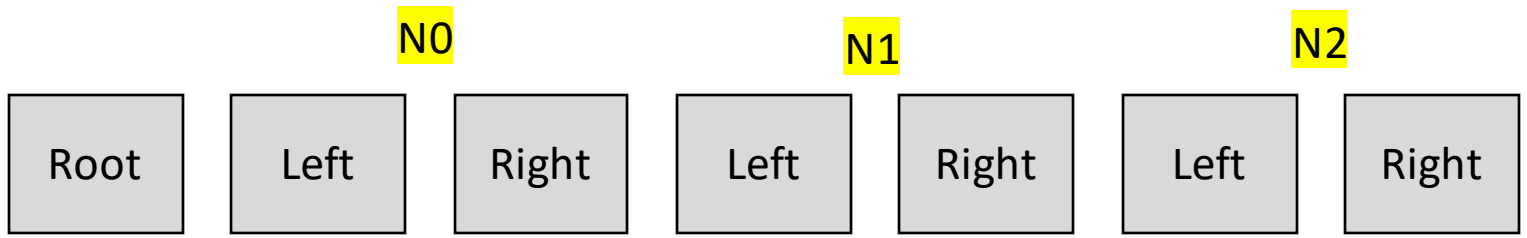


Scheme for Representing Shapes

- Order all possible values of each field assigning each order a unique ID
- Order all fields into a vector
- Each shape == vector of field values

Example: BinaryTree of up to 3 Nodes:

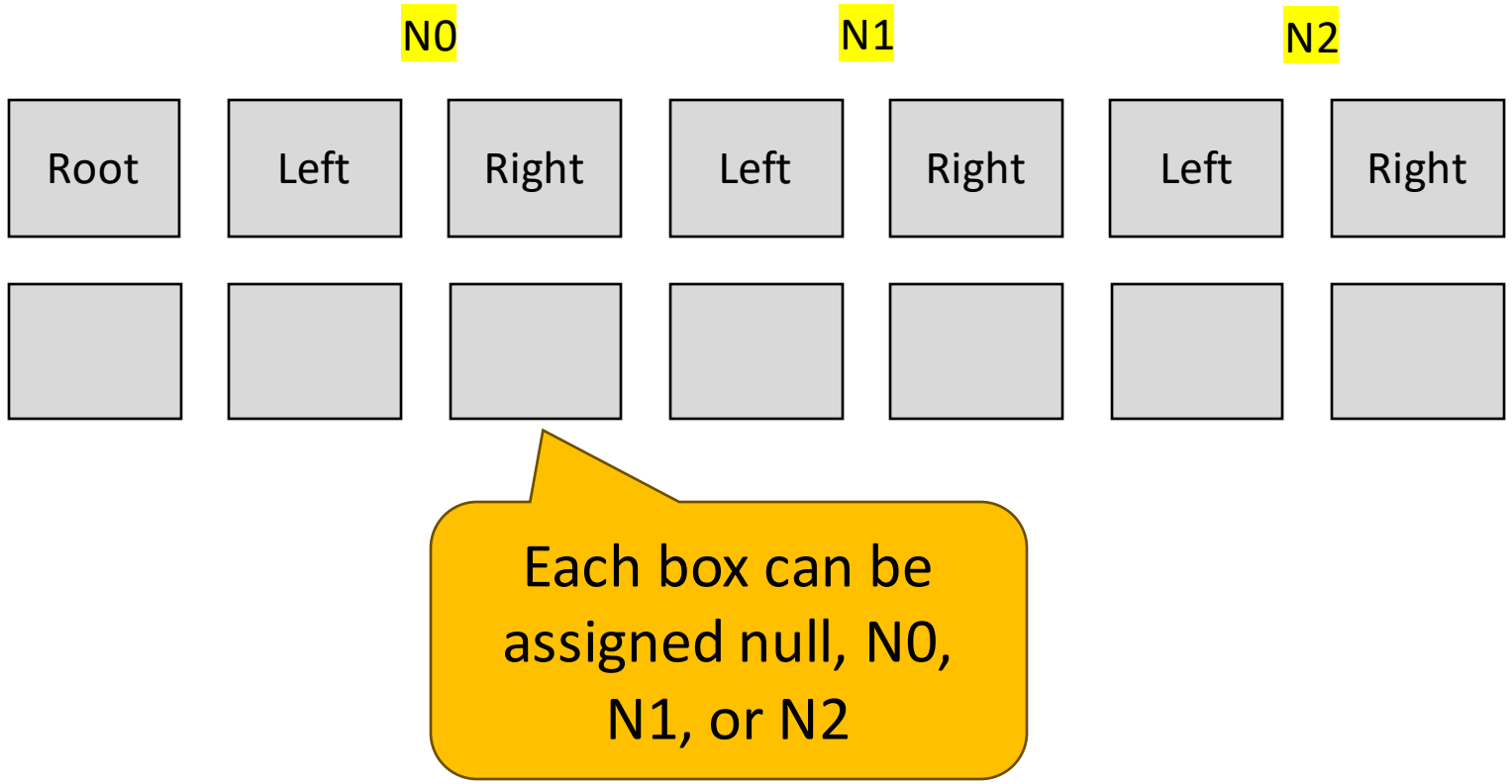
```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```

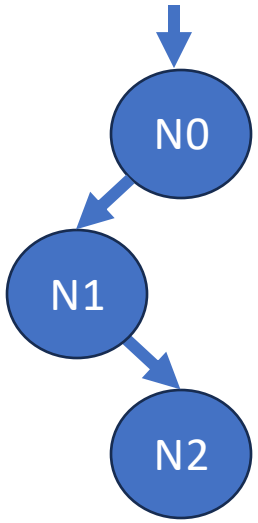


Each box can be assigned null, N0, N1, or N2

QUIZ: Representing Shapes

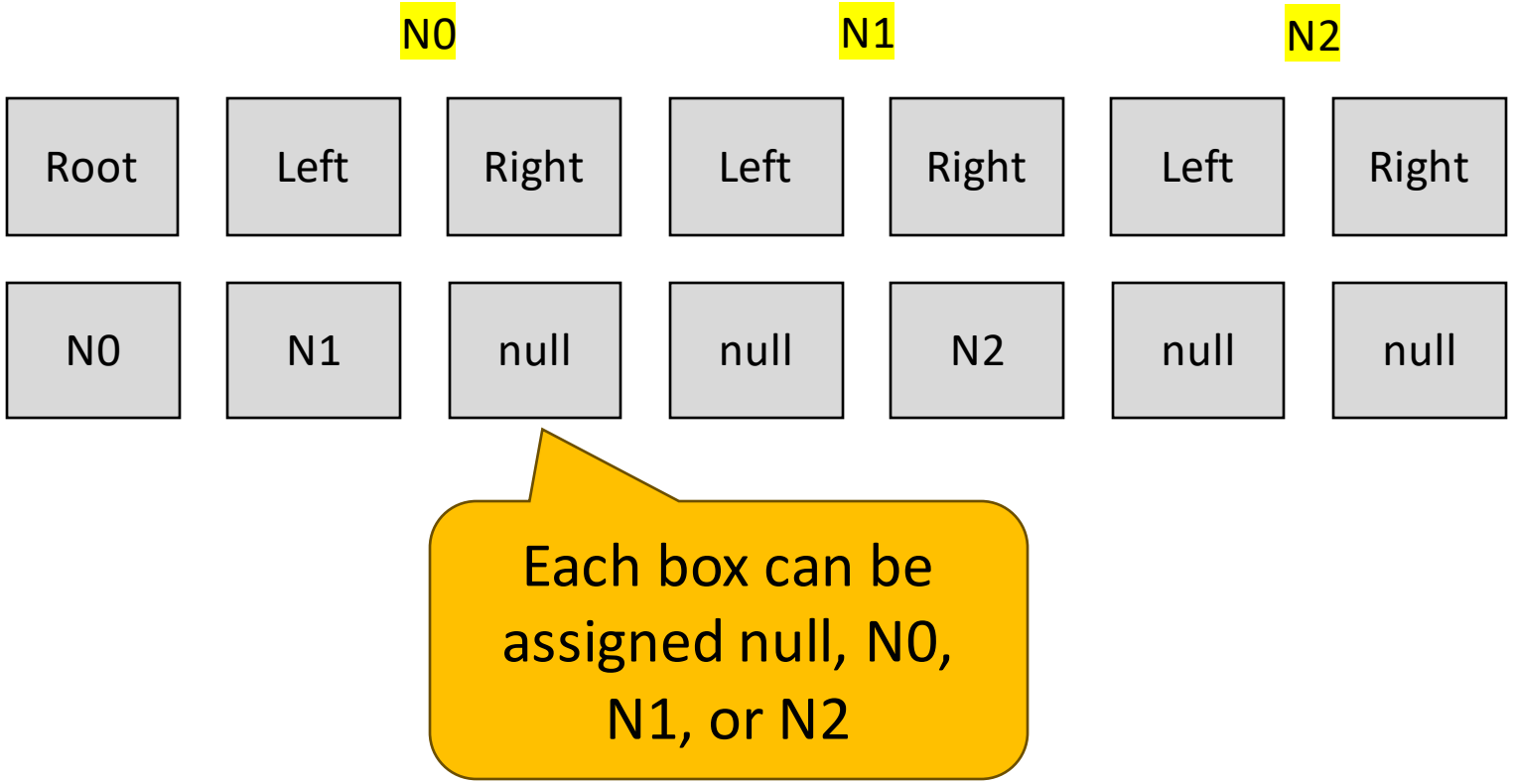
- Fill in the values in the box to represent the depicted shape:

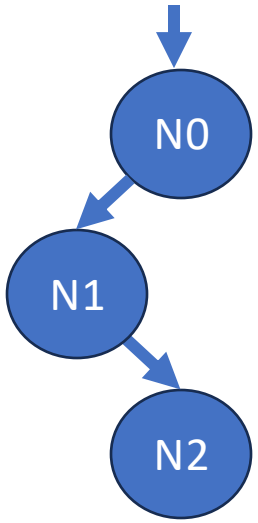




QUIZ: Representing Shapes

- Fill in the values in the box to represent the depicted shape:



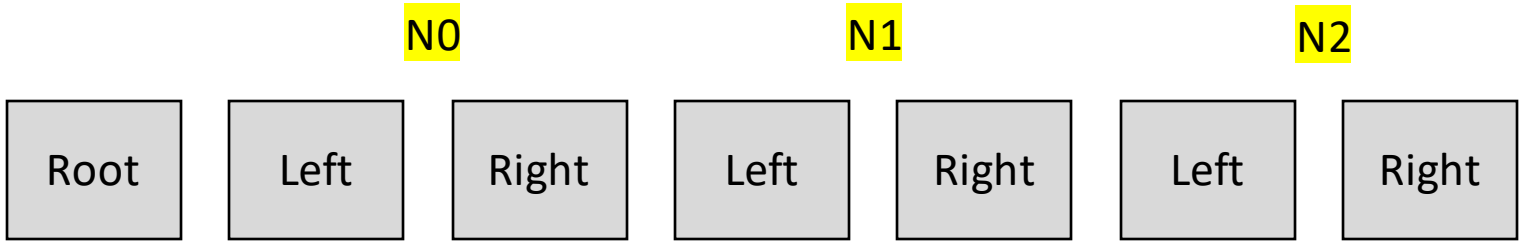


A Simple Algorithm

- User selects some maximum input size k
- Generate all possible inputs up to size k
- Discard inputs where pre-condition is false
  - E.g., calling removeRoot() method on empty BinaryTree
- Run program on remaining inputs
- Check results using post-condition

Enumerating Shapes

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

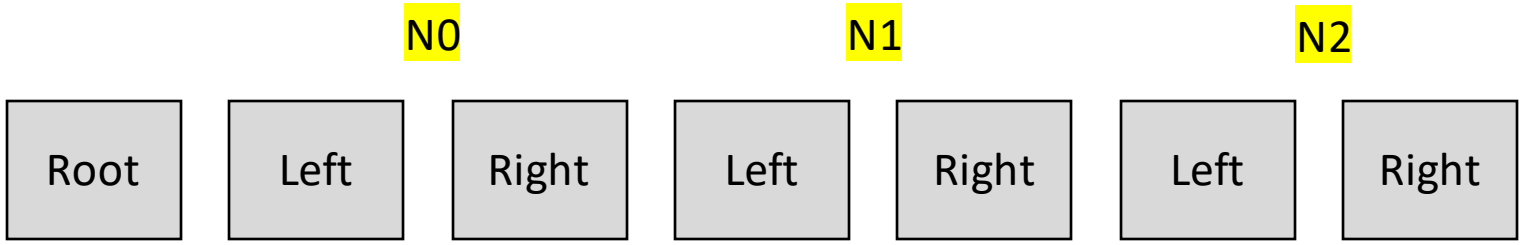


Each box can be assigned null, N0, N1, or N2

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

Enumerating Shapes

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



Each box can be assigned null, N0, N1, or N2

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

$4^7 = 16{,}384$

Size of the state space of generating binary trees of up to 3 nodes

The General Case for Binary Trees

- How many binary trees shapes can be enumerated using <=k nodes?
- Calculation:
  - A BinaryTree object bt
  - k Node objects, N0, N1, …
  - 2k+1 Node pointers
    - Root (for bt)
    - left, right (for each Node object)
  - k+1 possible values (N0, N1, …, Nk, or null) per pointer
- **$(k+1)^{(2k+1)}$** possible "binary trees"

A Lot of "Binary Trees"!

- The number of trees grows exponentially in input size (**$(k+1)^{(2k+1)}$**)
  - 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

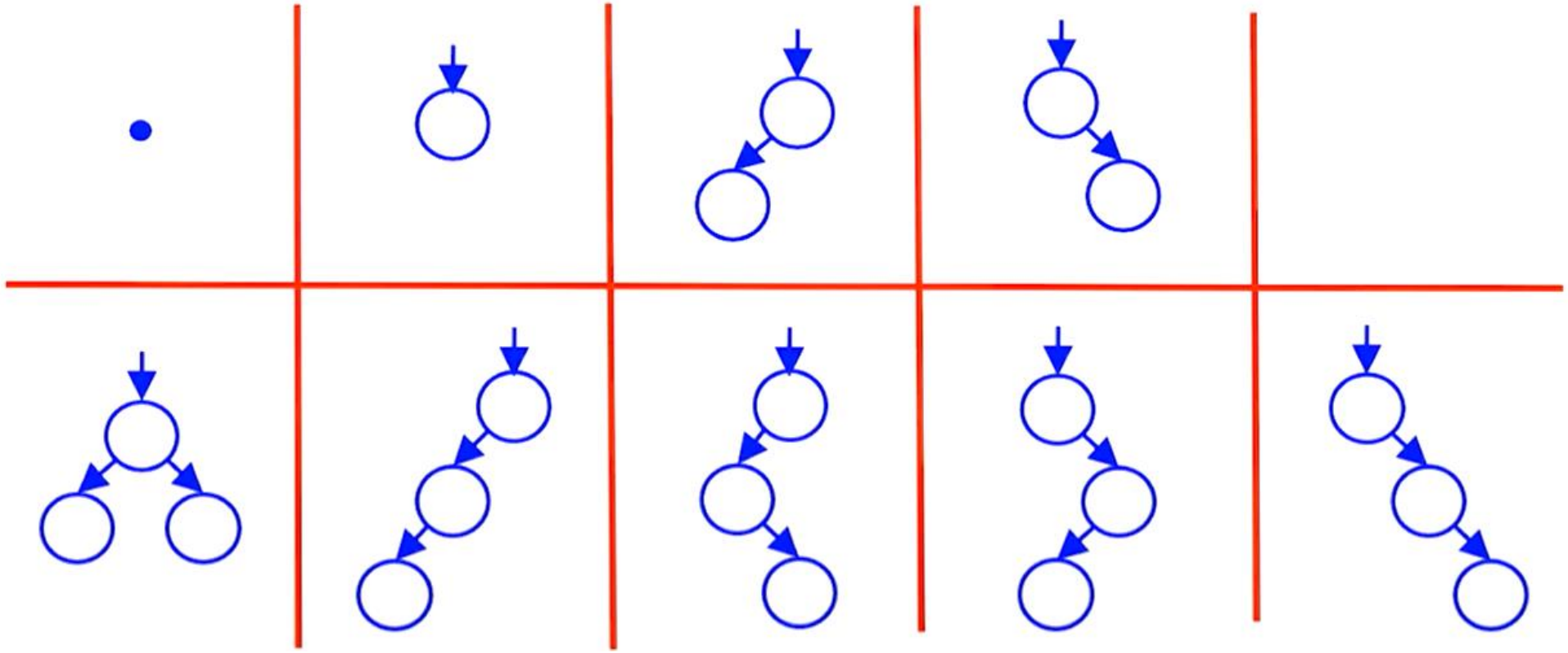
- **$(k+1)^{(2k+1)}$**) is gross overestimate!

- Many of the shapes are not even trees:

- Many of the trees are isomorphic (indistinguishable for the purpose of testing):

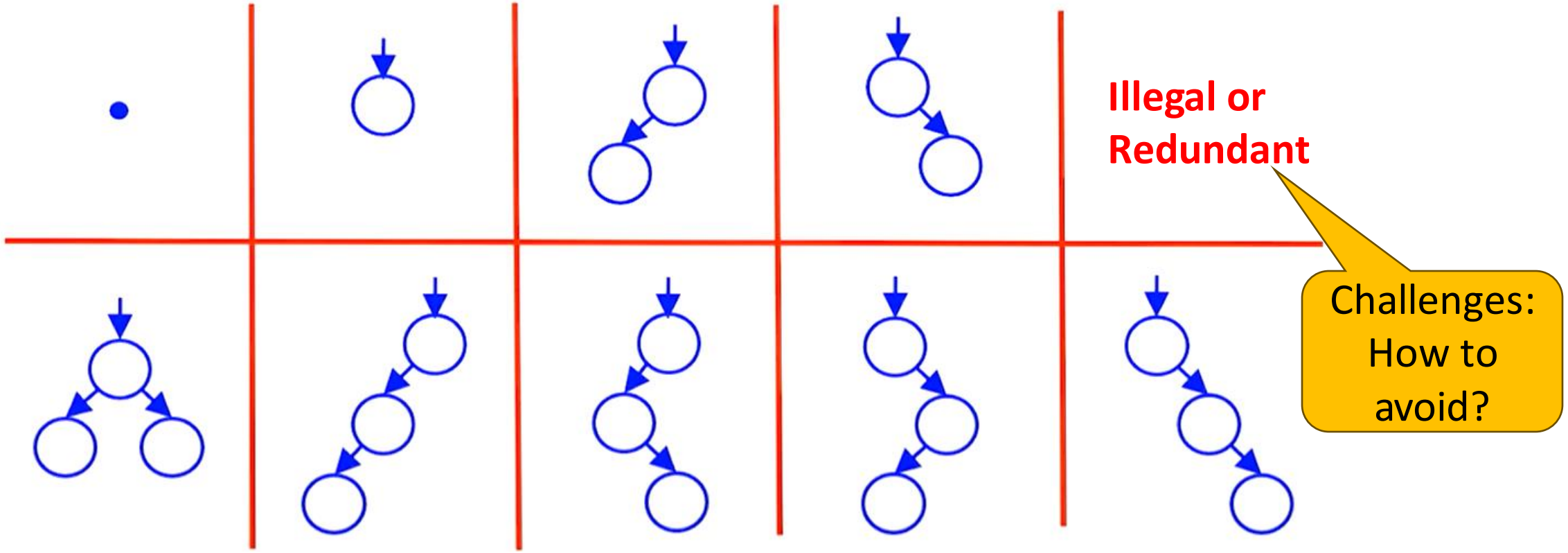
How Many Trees?

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



How Many Trees?

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



Another Insight

- Avoid generating inputs that do not satisfy the invariant
- Use the invariant to guide the generation of tests

Korat – The Technique

- Instrument the invariant
  - Add code to record which fields of the input are accessed by the invariant

- Observation:
  - If the invariant doesn't access a field, then it doesn't depend on that field

The Invariant for Binary Trees

- Root may be null
- If Root is not null:
  - No cycles
  - Each node (except Root) must have 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;
  }
}
```

Returns true when binary tree object satisfies the invariant

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;
  }
}
```

How is this method traversing the tree? BFS or DFS?

The Invariant for Binary Trees

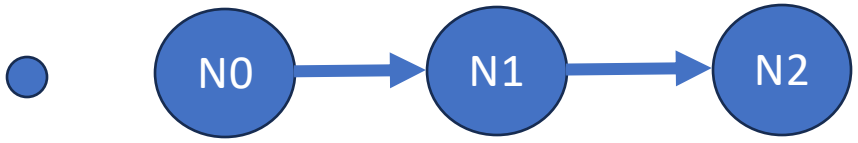
```
public boolean repOK(BinaryTree bt) {
   if (bt.root == null) return true;
   Set visited = new HashSet();
   List workList = new LinkedList();
   visited.add(bt.root);
   workList.add(bt.root);
   while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left !=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

How is this method traversing the tree?
BFS or DFS?

Example: Using the Invariant

- Consider the following tree:



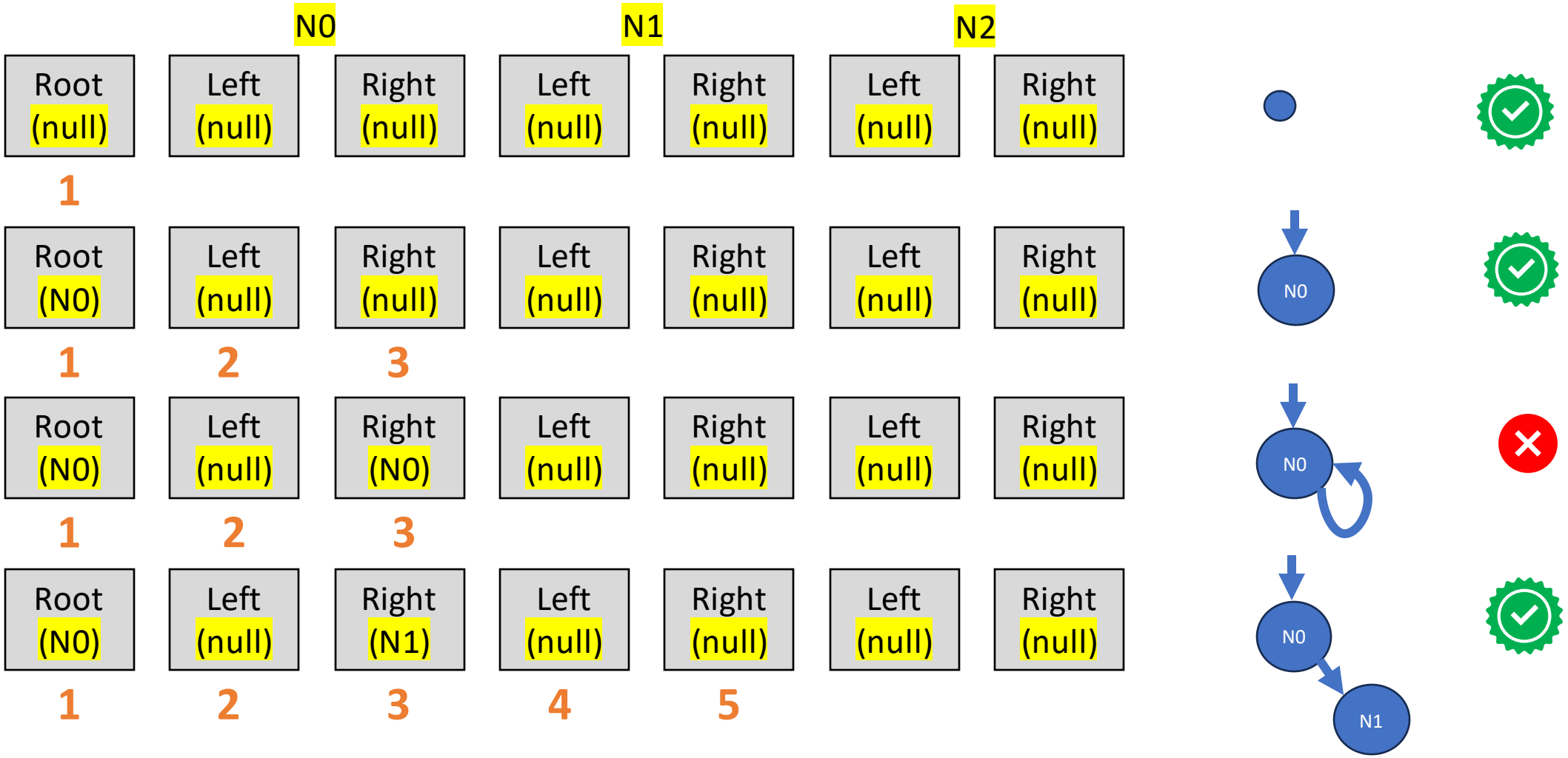
Root (null)	Left (null)	Right (N1)	Left (null)	Right (N1)	Left (null)	Right (null)

- The invariant (repOK) accesses only the root as it is null
  - Every possible shape for nodes other than root will yield the same result
  - This single input eliminates 25% of the tests!

Korat Approach

- 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 check and discard shapes isomorphic to previously generated shapes

Example: Enumerating Binary Trees



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}

Korat: Strengths and Weaknesses

- Strong when we can enumerate all possibilities
  - E.g., Four nodes, two edge per node
  - Good for:
    - Linked data structures
    - Small easily specified procedures
    - Unit testing
- Weaker when
  - Enumeration is weak
    - Integers, Floating-point numbers, Strings.
  - Only good as pre- and post-conditions
    - Is_member(x, list) vs !is_empty(list) pre-condition for remove() method of List
    - !contains(x, list') vs is_list(list') post-condition for remove() method of List

Announcements

- Project Plan Presentation and Report assignment released
  Due **next Wednesday, 02/14/2024**
  - Describe **exactly** what you have decided to do in project

- Homework-2 released
  Due **in two weeks, 02/19/2024**

- Class Participation
  - Involve in project and paper presentations by asking/answering questions on Canvas (see Class Participation assignment)