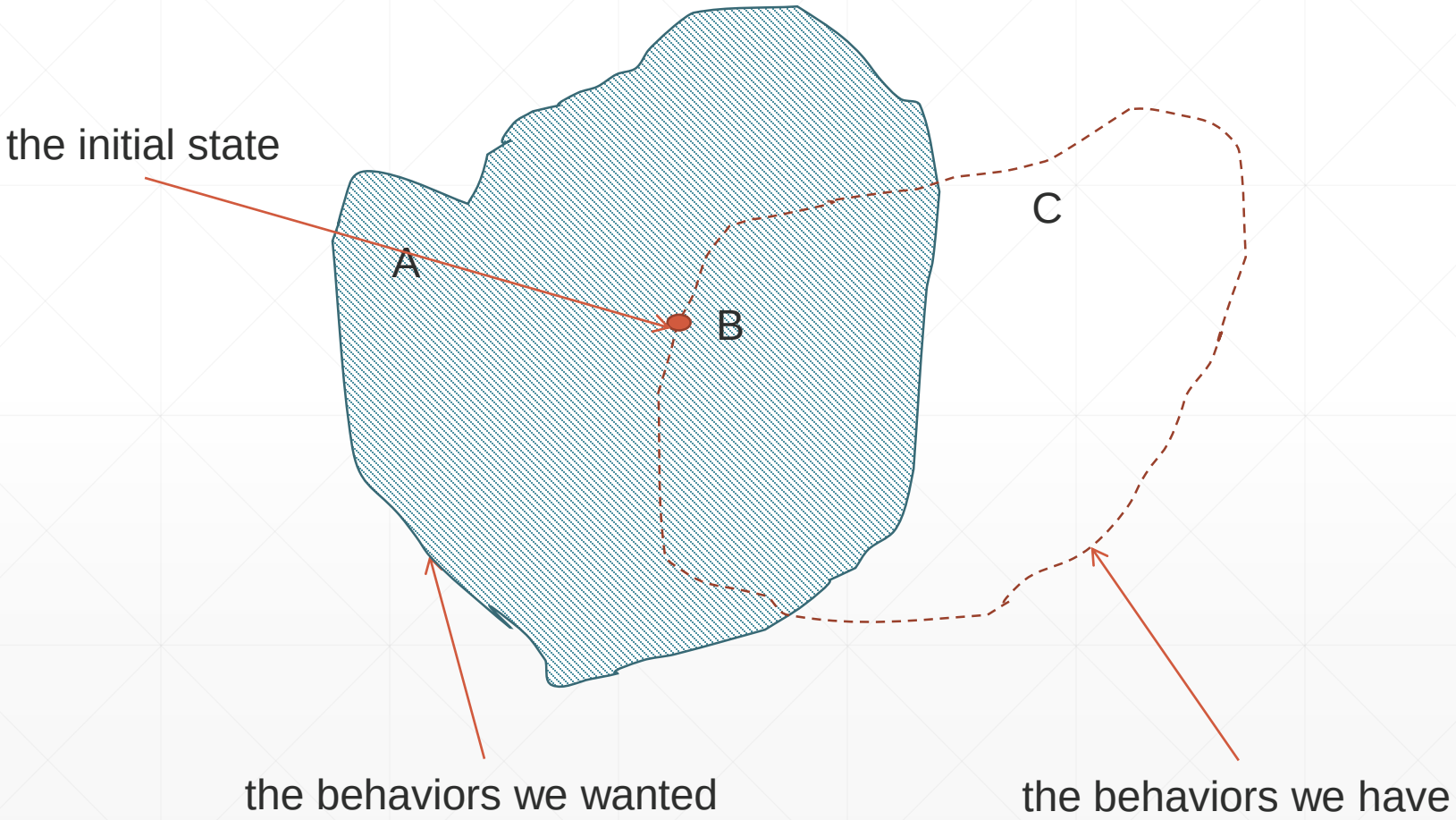
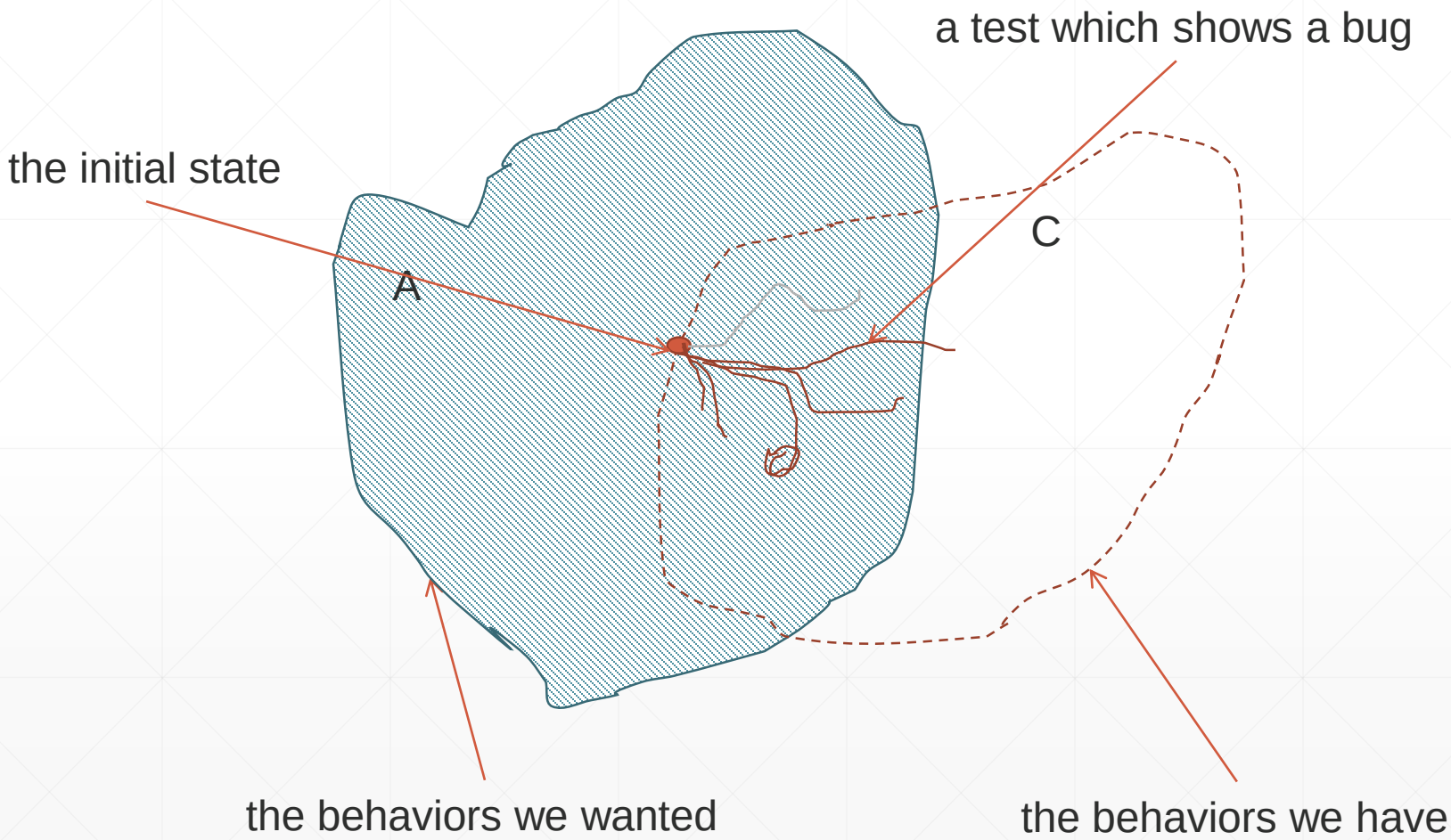
Software Testing

Classical Testing Techniques

A Big View: Testing



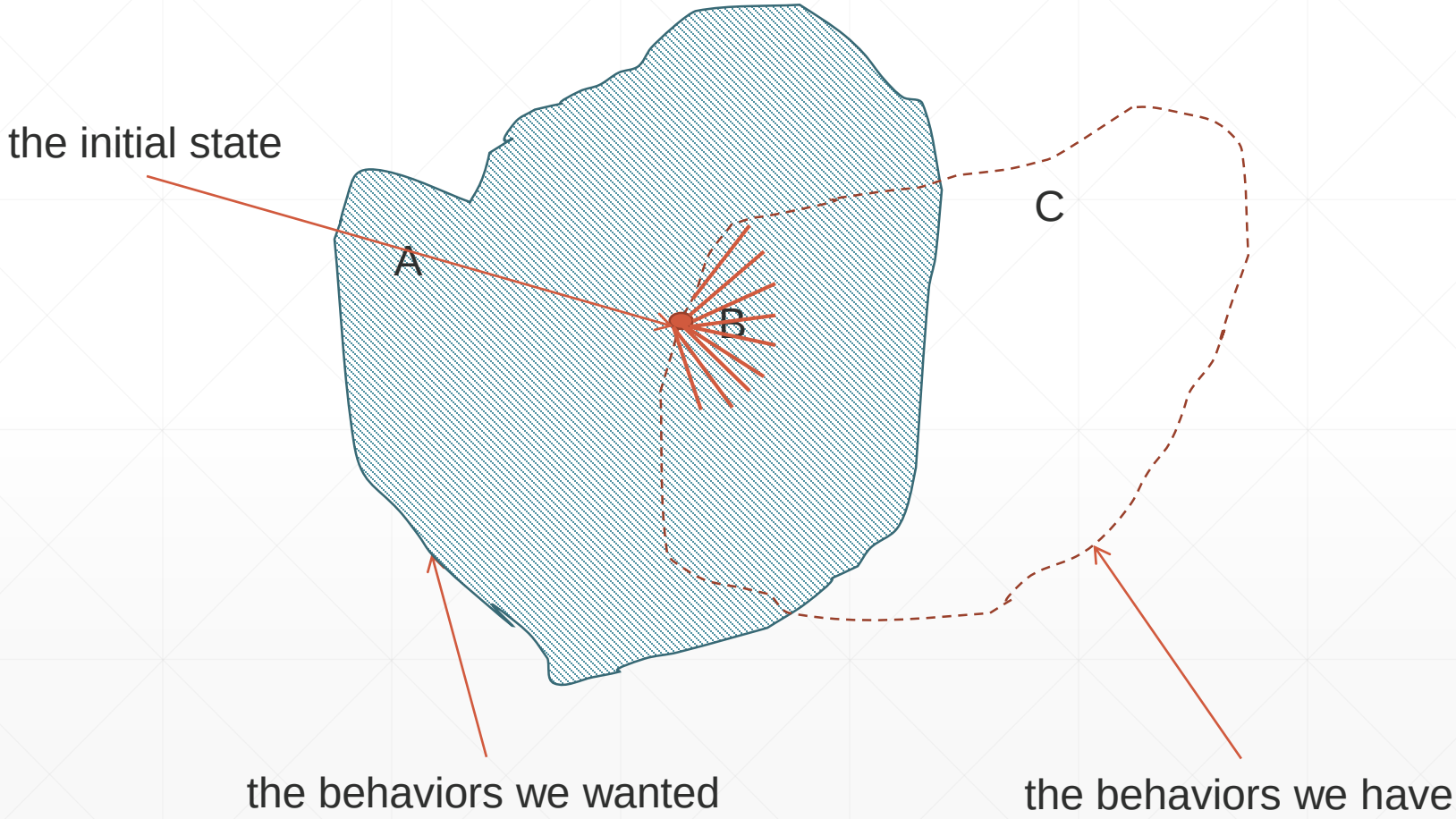
A Big View: Testing



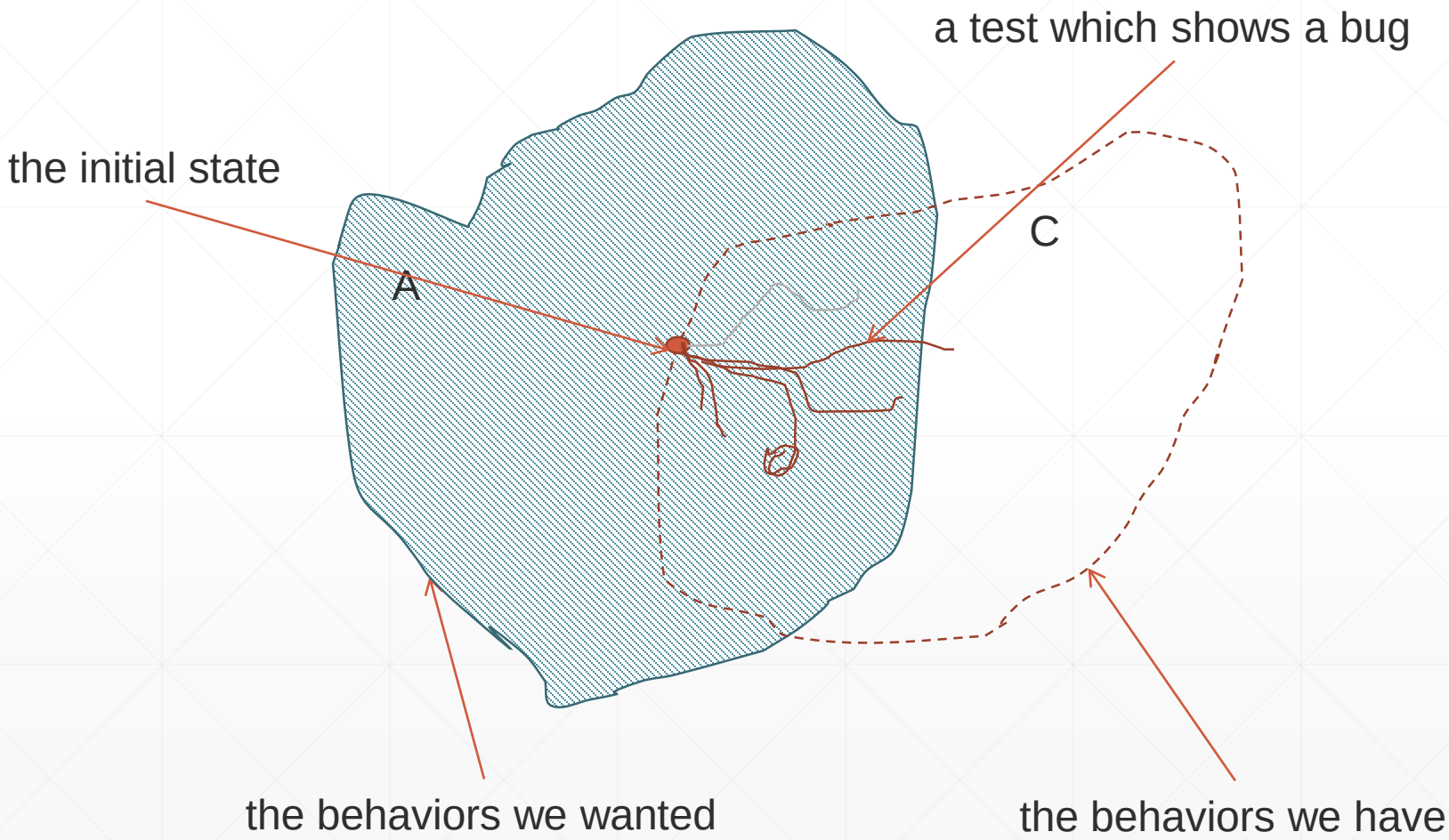
Testing

- Methods: white-box testing, black-box testing, grey-box testing
- Levels: unit testing, integration testing, system testing, etc.
- Types: compatibility testing, regression testing, acceptance testing, alpha testing, beta testing, function/non-functional testing, combinatorial testing, performance testing, security testing, etc.

A Big View: Systematic Testing



A Big View: Random Testing



Testing techniques

- Regression Testing
- Fuzz Testing

Regression vs. Fuzzing

- Regression: Run program on many normal inputs, look for badness.
  - Goal: Prevent normal users from encountering errors (e.g. assertions bad).

- Fuzzing: Run program on many abnormal inputs, look for badness.
  - Goal: Prevent attackers from encountering exploitable errors (e.g. assertions often ok)

Regression Testing

Regression Testing

- When we try to enhance the software
  - We may also bring in bugs
  - The software works yesterday, but not today, it is called “regression”

- Numbers -- Empirical study on eclipse 2005
  - 11% of commits are bug-inducing
  - 24% of fixing commits are bug-inducing

Regression Example

Original Program

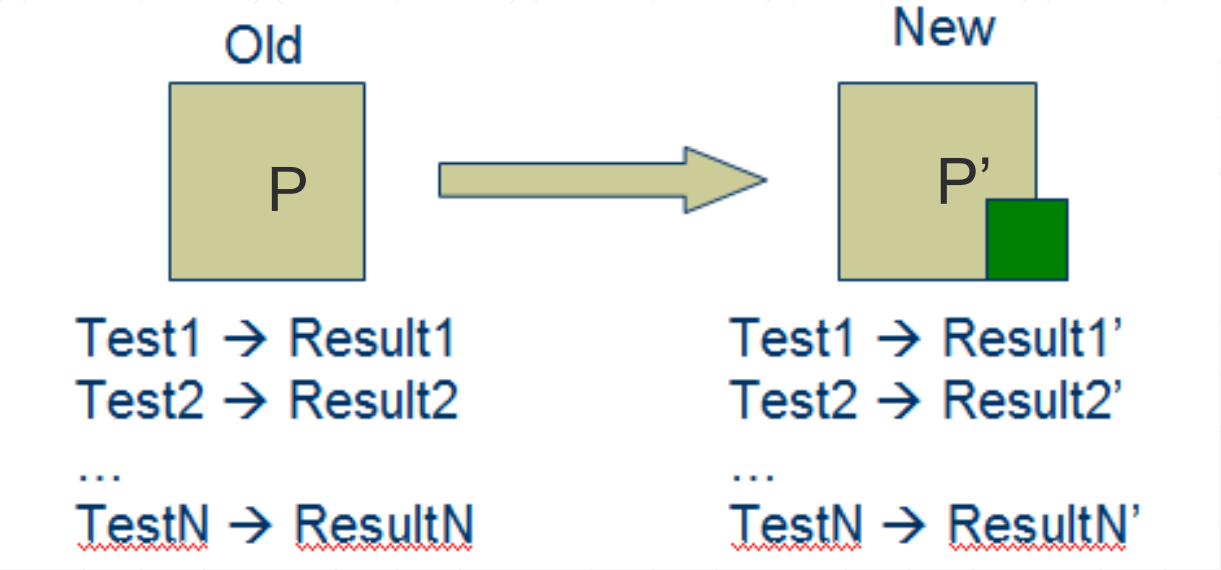
```
public int[] reverse(int[] origin){
    int[] target = new int[origin.length];
    int index = 0;
    while(index < origin.length - 1){
        index++;    //bug, missing origin[0]
        target[origin.length-index] = origin[index];
    }
    return target;
}
```

Modified Program

```
public int[] reverse(int[] origin){
    int[] target = new int[origin.length];
    int index = 0;
    while(index < origin.length - 1){
        index++;
        target[origin.length-index] = origin[index];
    }
    target[origin.length-1] = origin[0]
    return target;
}
```

Regression Testing

More Formally Given program P, its modified version P', and a ***test set T*** that was used to previously test P, find a way to utilize T to gain sufficient ***confidence*** in the correctness of P'



$Result_i = Result_{i'}$, forall i

Regression Faults -- Example

- When a stable baseline system B and a delta component D pass individually adequate tests, but fail when used together
- D can cause some component in B to fail only if there is some *dependency* between them.
- Dependencies occur for many reasons and so regression faults can occur in many ways. **Question**: how many can you think of?
- There is no practical means to develop a test suite guaranteed to reveal all regression faults

Regression Faults -- Example

- Case 1: **D has a side effect on B:**
  - B fails because a new action of D is inconsistent with B's requirements, assumptions, or contract with respect to D
- Case 2: **D is a client of B:**
  - D sends a message that violates B's invariant/precondition. B is not defensive and accepts the incoming message, leading to failure
- Case 3: **D is a server of B:**
  - B sends a message to D. D's post-consitions have changed (or contain bugs). D returns a value that causes a violation of B's invariant . B fails or returns an invalid value to another baseline component C. C fails
- …

Maintenance Testing

Regression Testing

- Run old test cases on the new version of software
- It will cost a lot if we run the whole suite each time
- Try to save time and cost for new rounds of testing

Partial regression testing?

If the time (and money …) interval allocated to regression testing is limited, **only a fraction of the test suite can be executed**, with the risk of missing test cases that are able to reveal defect not yet discovered

Partial Regression Testing

- **Regression test selection:** the cost of regression testing is reduced by selecting a subset of the existing test suite based on information about the program, modified version and test suite.
- **Test suite minimization:** the test suite is reduced to a minimal subset that maintains the same *coverage* as the original test suite with respect to a given coverage criterion.
- **Test case prioritization:** test cases are ordered so that those with the highest priority are executed earlier, for example with the objective of achieving code coverage at the fastest possible rate, or of exercising the modules according to their propensity to fail.

Test Prioritization

- Rank all the test cases
- Run test cases according to the ranked sequence
- Stop when resources are used up
- How to rank test cases?
  - to discover bugs sooner
  - to achieve higher coverage sooner

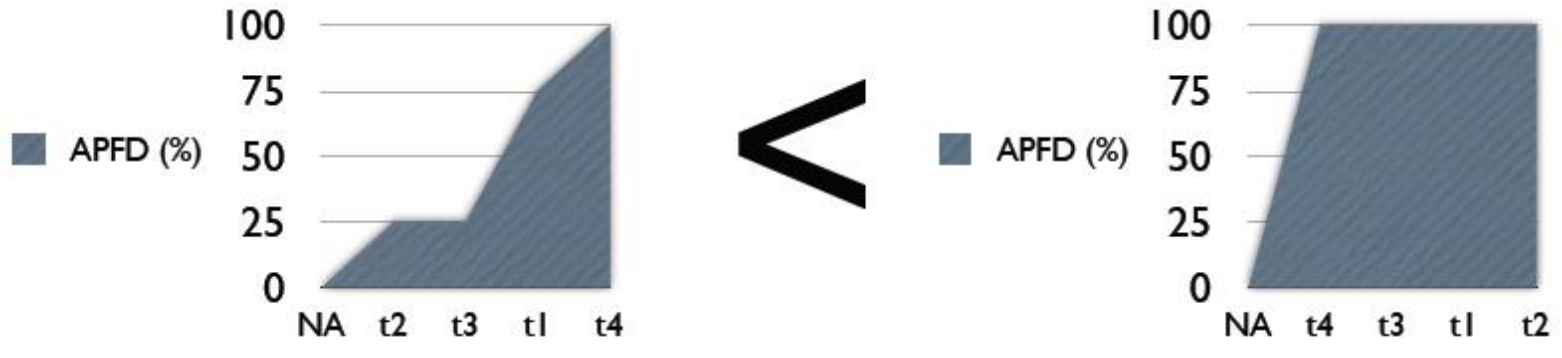
Measurement of Test Prioritization -- Average Percentage of Fault Detected (APFD)

选择大的值

- Compare two test case sequences
- A number of faults (bugs) are detected after each test case
- The following two sequences, which is better?
  - Seq1: t1(2), t2(3), t3(5)
  - Seq2: t2(1), t1(3), t3(5)
- APFD is the average of these numbers (normalized with the total number of faults), and 0 for initial state
  - APFD (Seq1) = (0/5 + 2/5 + 3/5 + 5/5) / 4 = 0.5
  - APFD (Seq2) = (0/5 + 1/5 + 3/5 + 5/5) / 4 = 0.45

APFD: Ilustration

- APFD can be deemed as the area under the TestCase-Fault curve
- For example: Consider t1(f1, f2), t2(f3), t3(f3), t4(f1, f2, f3, f4)



Coverage-based test case prioritization

- Code coverage based
  - Require recorded code-coverage information in previous testing

- Total Strategy
- Additional Strategy

Total Strategy

- The simplest strategy -- always select the unselected test case that has the best coverage
- Example: consider code coverage on five test cases:
  - T1: s1, s3, s5
  - T2: s2, s3, s4, s5
  - T3: s3, s4, s5
  - T4: s6, s7
  - T5: s3, s5, s8, s9, s10
  - Ranking: T5, T2, T1 / T3, T4

Additional Strategy

- An adaption of total strategy: instead of always choosing the test case with highest coverage
  - choose the test case that result in most extra coverage
  - starts from the test case with highest coverage
- Example: Consider code coverage on five test cases:
  - T1: s1, s3, s5
  - T2: s2, s3, s4, s5
  - T3: s3, s4, s5
  - T4: s6, s7
  - T5: s3, s5, s8, s9, s10
  - Ranking: T5(5), T2(2, s2, s4) / T4(2, s6, s7), T1(1, s1), T3

Setting the threshold

- Prioritization help us to find bugs earlier
- Due to resource limit, we do not want to execute all test cases
- The testing should stop at some place in the prioritized rank list
  - Resource limit
    - Money, time
  - Coverage based
    - Cover all/certain percent of statements
    - Cover all/certain percent of mutations

Regression Testing – Test Relevant Code

- Basic Idea:
  - Only use test cases that cover the changed code
  - Can be combined with test prioritization
  - Give more priority to the test cases that cover more code affected by the change
  - Determine the affected code with program slicing

Which test case is better?

```
1    void main() {
2            int sum, i;
3            sum = 0;  -> sum = 1;
4            i = read();
5            if(i >= 12){
6              String rep = report(invalid, i);
7              sendReport(rep)
8            }else{
9                while ( i<11 ) {
10                   sum = add(sum, i);
11                   i = add(i, 1);
12               }
13           }
14 }
```

Test case: 0
Test case: 13

Fuzz Testing

Professor Messer's CompTIA Security+ Certification Training Course

Professor Messer

SY0-301: CompTIA Security+
Section 4.1 - Application Security

Fuzzing

Content provided by:
http://www.gtslearning.com

Picture Credit: awmyl (robin red)
http://www.flickr.com/photos/awellmanneredyounglady/750630563/
http://creativecommons.org/licenses/by/2.0/

© 2011 Messer Studios, LLC

Fuzzing Basics

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target interface
- Application is monitored for errors
- Inputs are generally either
  - file based, e.g., .pdf, .png, .wav, .mpg
  - Or network based, e.g., http, SNMP, SOAP
  - Or other…, e.g., crashme()

Trivial Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- Anomalous requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAA.html HTTP/1.1
  - GET /index.html HTTPTTPTTPTTPTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1

Different Ways To Generate Inputs

- Mutation Based - “Dumb Fuzzing”: A fuzzer that generates completely random input.
- Generation Based - “Smart Fuzzing”: programmed with knowledge of the input format
- Example: Input format of a password

- More readings: https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics
  - e.g. remove NUL, shift character forward
- Examples:
  - Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.

Example: fuzzing a pdf viewer

- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script to)
  1. Grab a file
  2. Mutate that file
  3. Feed it to the program
  4. Record if it crashed (and input that crashed it)

Dumb Fuzzing In Short

- Strengths
  - Super easy to setup and automate
  - Little to no protocol knowledge required
- Weaknesses
  - Limited by initial corpus
  - May fail for protocols with checksums, those which depend on challenge response, etc.

Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing

Example: Protocol Description

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation Based Fuzzing In Short

- Strengths
  - completeness
  - Can deal with complex dependencies, e.g. checksums

- Weaknesses
  - Have to have spec of protocol
    - Often can find good tools for existing protocols, e.g. http, SNMP
  - Writing generator can be labor intensive for complex protocols
  - The spec is not the code

Fuzzing Tools

Input Generation

- Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  - Mu-4000, Codenomicon, PROTOS, FTPFuzz
- Fuzzing Frameworks: You provide a spec, they provide a fuzz set
  - SPIKE, Peach, Sulley
- Dumb Fuzzing automated: you provide the files or packet traces, they provide the fuzz sets
  - Filep, Taof, GPF, ProxyFuzz, PeachShark
- Many special purpose fuzzers already exist as well
  - ActiveX (AxMan), regular expressions, etc.

How Much Fuzz Is Enough?

- Mutation based fuzzers can generate an infinite number of test cases... When has the fuzzer run long enough?
- Generation based fuzzers generate a finite number of test cases. What happens when they're all run and no bugs are found?

Example: PDF

- I have a PDF file with 248,000 bytes
- There is one byte that, if changed to particular values, causes a crash
  - This byte is 94% of the way through the file
- Any single random mutation to the file has a probability of .00000392 of finding the crash
- On average, need 127,512 test cases to find it
- At 2 seconds a test case, that's just under 3 days...
- It could take a week or more...

Code Coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g., gcov

Types of Code Coverage

- Line coverage
  - Measures how many lines of source code have been executed.
- Branch coverage
  - Measures how many branches in code have been taken (conditional jmps)
- Path coverage
  - Measures how many paths have been taken

Example

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- Requires?
  - 1 test case for line coverage
  - 2 test cases for branch coverage
  - 4 test cases for path coverage

Problems with Code Coverage

- Code can be covered without revealing bugs

- Error checking code mostly missed (and we don't particularly care about it)

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

Fuzzing Rules of Thumb

- Protocol specific knowledge very helpful
  - Generational tends to beat random, better spec's make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you find
- Best results come from guiding the process
- Code coverage can be very useful for guiding the process

The Future of Fuzzing

Outstanding Problems

- What if we don’t have a spec for our protocol/How can we avoid writing a spec.

- How do we select which possible test cases to generate

Whitebox Fuzzing

- Infer protocol spec from observing program execution, then do generational fuzzing

- How to generate constraints?
  - Observe running program
    - Instrument source code (EXE)
    - Binary Translation (SAGE, Catchconv)
  - Treat inputs as symbolic
  - Infer contraints

How do we generate constraints?

- Observe running program
- Treat inputs as symbolic
- Infer constraints

```
int test(x)
{
    if (x < 10) {            //X < 10 and X <= 0 gets us this path
            if (x > 0) {     //0 < X < 10 gets us this path
        return 1;
    }
  }
                             //X >= 10 gets us this path
  return 0;
}
```

Constraints:
X >= 10
0 < X < 10
X <= 0
Solve Constraints -- we get test cases: {12,0,4}

- Provides maximal code coverage

Korat: Automated Testing Based on Java Predicates

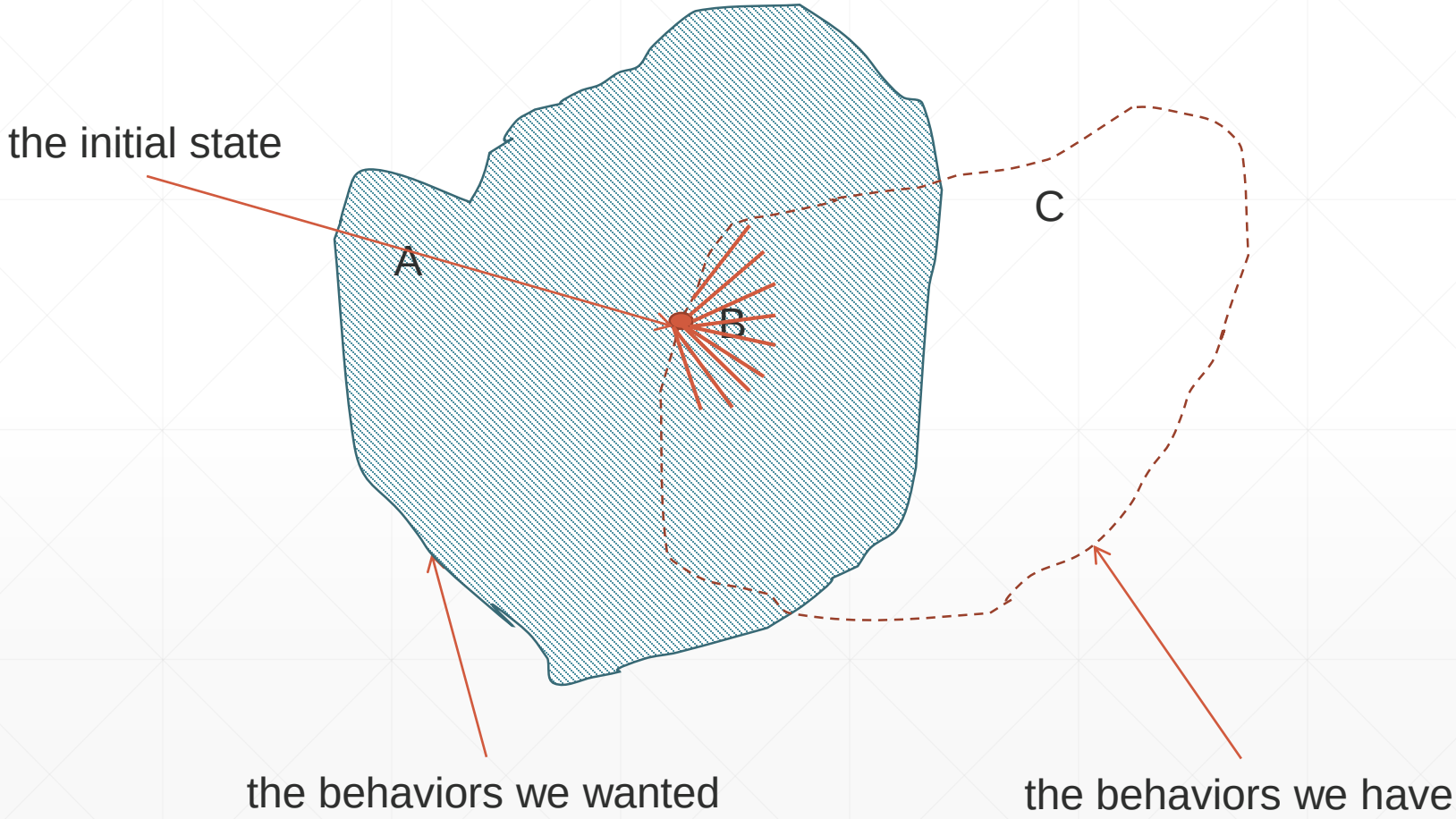
Boyapati et al., ISSTA 2002, ***ACM SIGSOFT Distinguished Paper Award***

Research Question

Isn’t jUnit good enough?

How do we automatically generate test cases so as to reveal bugs?

A Big View: Systematic Testing



Motivation

- It is important to be able to generate test cases automatically.
- It is important to generate test cases which are representative.
- Korat is merely a sample approach for systematic test case generation, however, it is similar in spirit to many systematic testing techniques (e.g., combinatorial testing, parameterized testing).

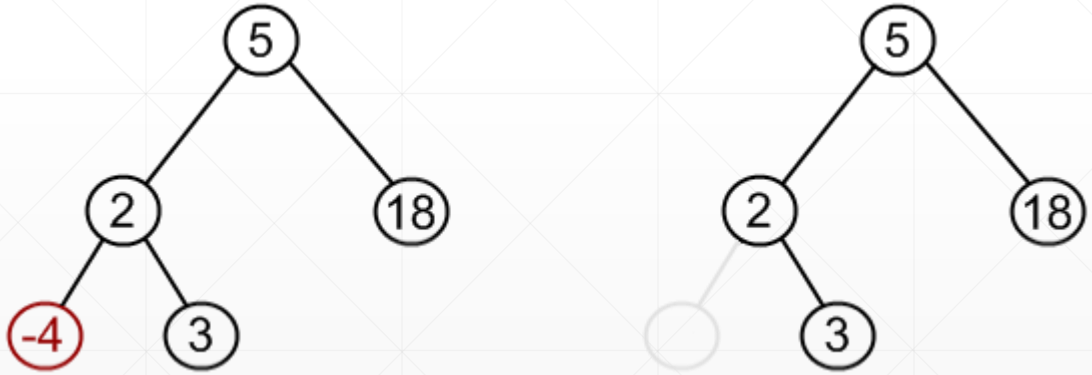
Example

```
public class BinaryTree {
    public static class Node {
        Node left; Node right;
    }
    private Node root;
    private int size;

    public void remove (Node n) {
        //some code
    }

    …
}
```

How do we test remove(node n)?



Example

- How do we test remove(Node n)?
  - We need a *valid* BinaryTree object **bt**.
  - We need a *valid* Node object **nd**.
  - We need to know what is expected after executing **bt.remove(nd)**

```
public class BinaryTree {
   public static class Node {
      Node left; Node right;
   }
   private Node root;
   private int size;

   public void remove (Node n) {
      //some code
   }

   …
}
```

Vocabulary

Class invariant:

- an invariant used to define what are valid objects of the class
- e.g., size == 0 if root == null and size equals to the number of nodes in the tree

```
public class BinaryTree {
    public static class Node {
        Node left; Node right;
    }
    private Node root;
    private int size;

    public void remove (Node n) {
        //some code
    }

    …
}
```

Vocabulary

Pre-condition (of a method)

- a condition which must be true prior to the execution of the method
- e.g., n must not be null.

The class invariant is always part of the pre-condition.

```
public class BinaryTree {
    public static class Node {
        Node left; Node right;
    }
    private Node root;
    private int size;

    public void remove (Node n) {
        //some code
    }

    …
}
```

Vocabulary

Post-condition (of a method)

- a condition which must be true after the execution of the method
- e.g., after remove, size is decremented by 1.

The class invariant is always part of the post-condition.

```
public class BinaryTree {
    public static class Node {
        Node left; Node right;
    }
    private Node root;
    private int size;

    public void remove (Node n) {
        //some code
    }

    …
}
```

Korat: Assumption

A class invariant is encoded as a method repOk(), which return true if and only if the object is in a state which satisfies the class invariant.

```
public boolean repOK() {
    if (root == null)
        return size == 0;

    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Node> workList = new
LinkedList<Node>();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }

    return (visited.size() == size);
  }
```

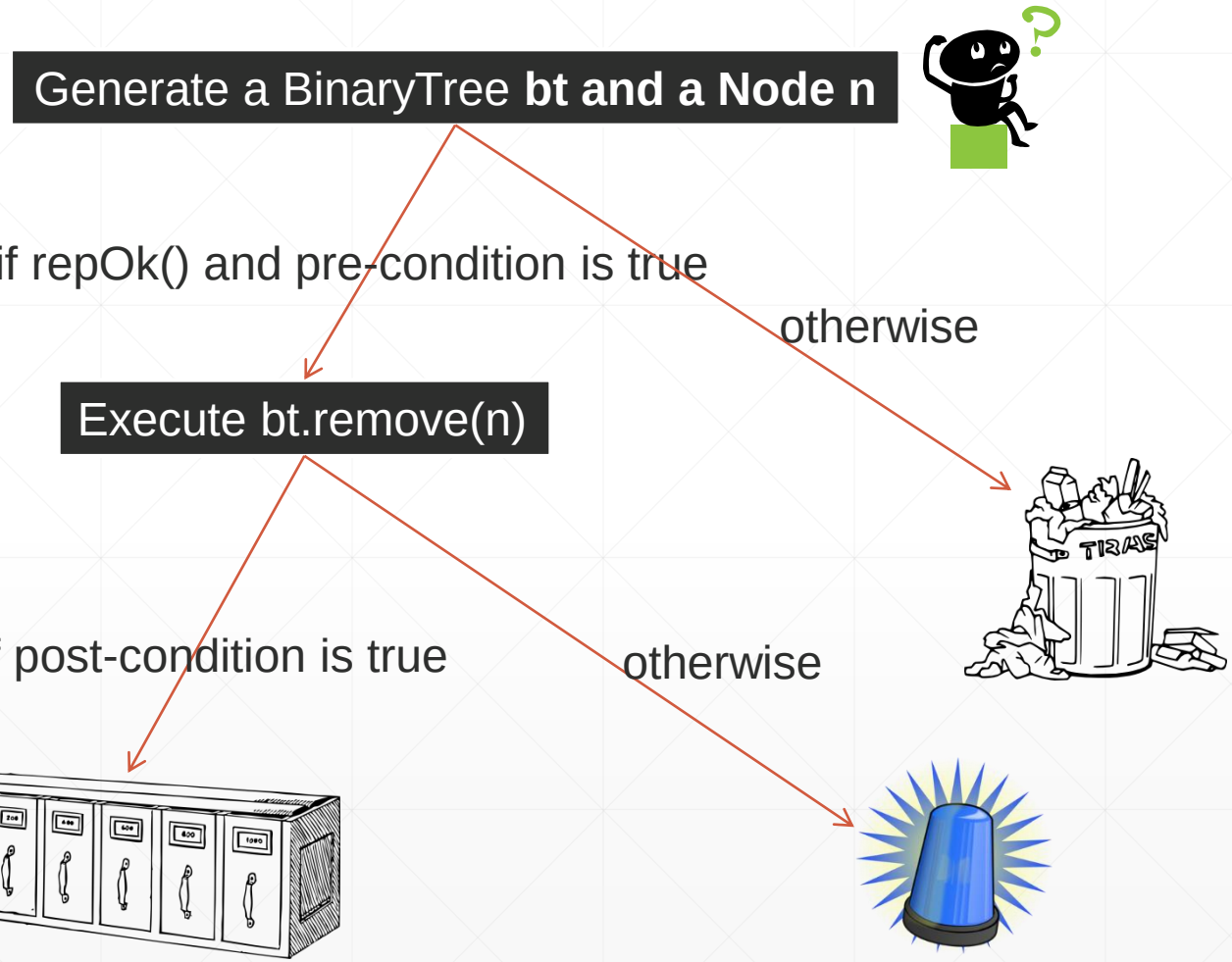
Korat: Assumption

- Pre-condition and post-condition are encoded in Java Modeling Language

```
//@ public invariant repOk(); // class invariant
// for BinaryTree
/*@ public normal_behavior // specification for remove
@ requires has(n); // precondition
@ ensures !has(n); // postcondition
@*/
public void remove(Node n) {
// ... method body
}
```

This is probably too harsh a pre-condition?

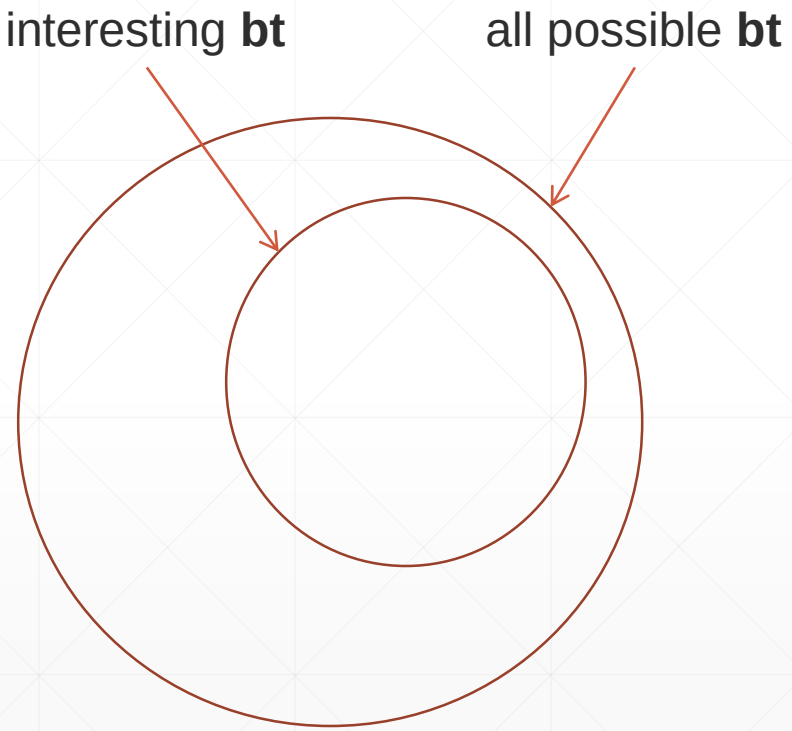
Karat: Approach



Finitization

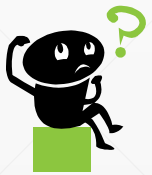
- There are infinitely many candidates for **bt** and **n**.
  - For each variable

    in the class, define its domain



Finitization

```
public static Finitization finBinaryTree(int NUM_Node) {
    Finitization f = new Finitization (BinaryTree.class);
    ObjSet nodes = f.createObjSet(“Node”, NUM_Node);
    nodes.add(null);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    return f;
}
```



```
public class BinaryTree {
    public static class Node {
        Node left; Node right;
    }
    private Node root;
    private int size;

    …
}
```

Finitization

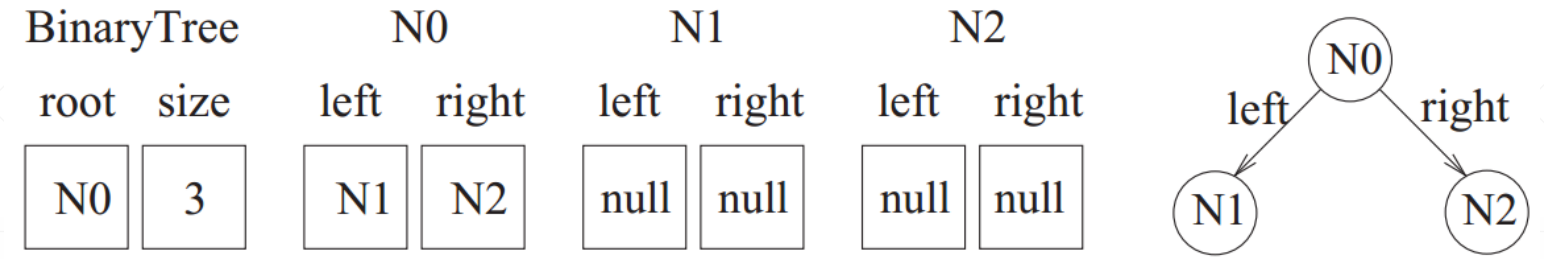
```
public static Finitization finBinaryTree(int NUM_Node) {
    Finitization f = new Finitization (BinaryTree.class);
    ObjSet nodes = f.createObjSet("Node", NUM_Node);
    nodes.add(null);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    return f;
}
```

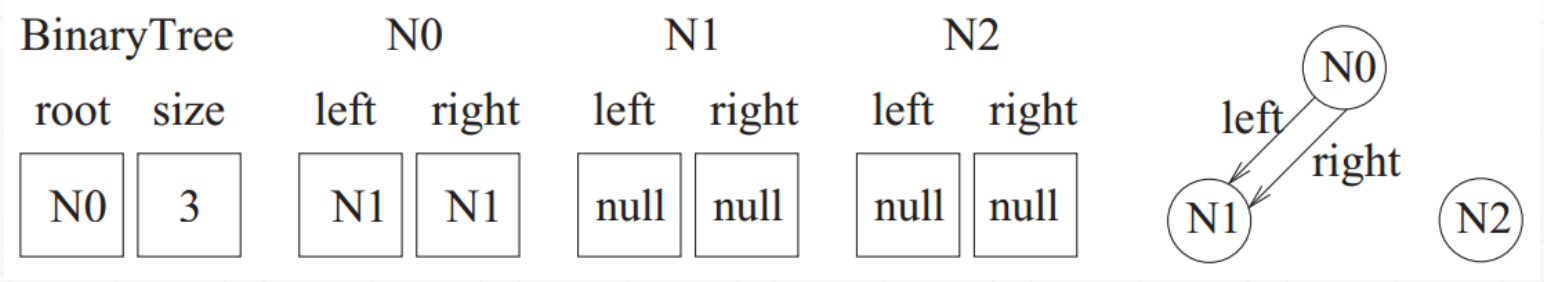
translation

nodes = {null, N0, N1, N2}
BinaryTree.root is a member of nodes
Node.left is a member of nodes
Node.right is a member of nodes

Example Trees

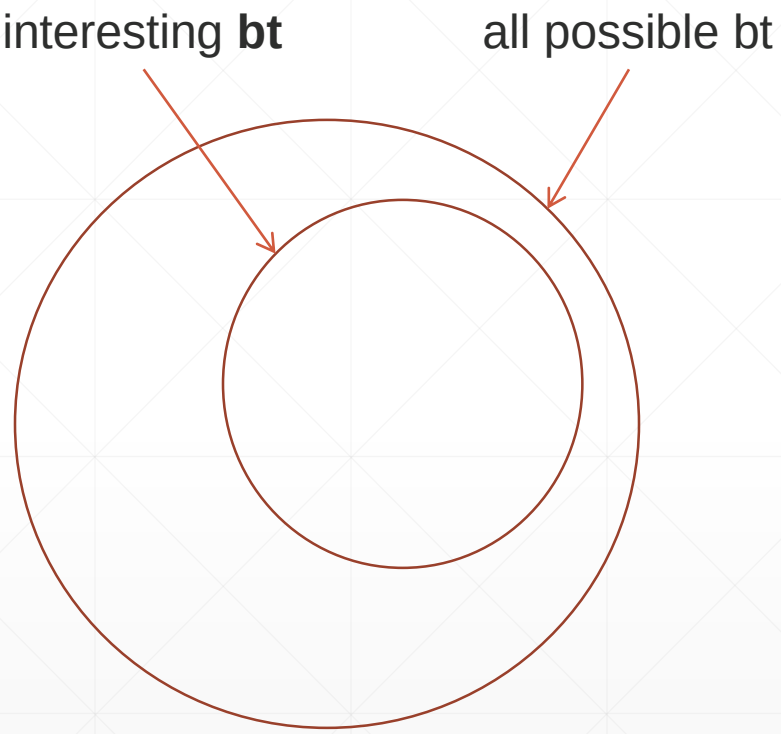
With finBinaryTree(3), there are 4 objects: one BinaryTree object, three Node objects, which could be set up as follows.





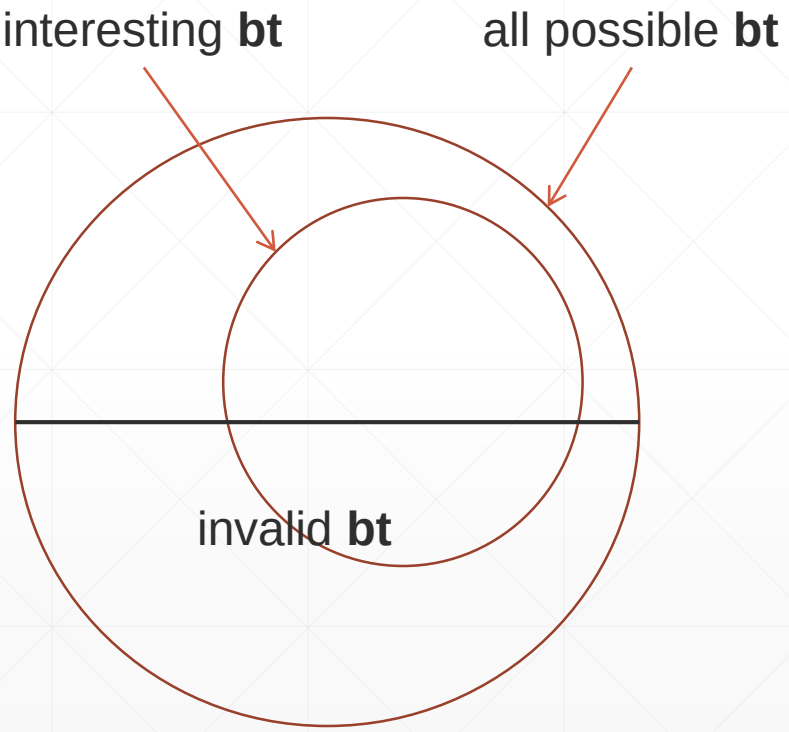
Finitization: the Space

- How many **bt** are there with finBinaryTree(3), assume that bt.size is always set to the right value?
  - 4^7
- How many **bt** are there with finBinaryTree(n)?
  - (n+1)^(2n+1)



Filtering 1

- For each candidate **bt** and **n**, check the pre-condition of remove. If the pre-condition is not satisfied, ignore that tree.



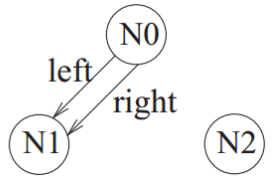
```
public boolean repOK() {
    if (root == null)
        return size == 0;

    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Node> workList = new LinkedList<Node>();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }

    return (visited.size() == size);
}
```

Is the following **bt** valid?

BinaryTree		N0		N1		N2	
root	size	left	right	left	right	left	right
N0	3	N1	N1	null	null	null	null

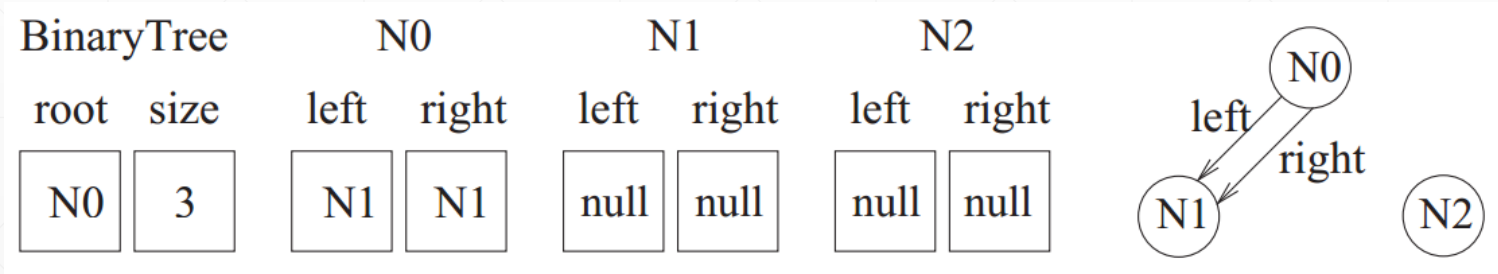


Korat: Search Algorithm

1. Order all the elements in every class domain and every field domain
   1. Node class ordering: <null, N0, N1, N2>
   2. Assume domain of size: <3>
2. Generate a candidate as a vector of field domain indices, e.g., [1,0,2,2,0,0,0,0]

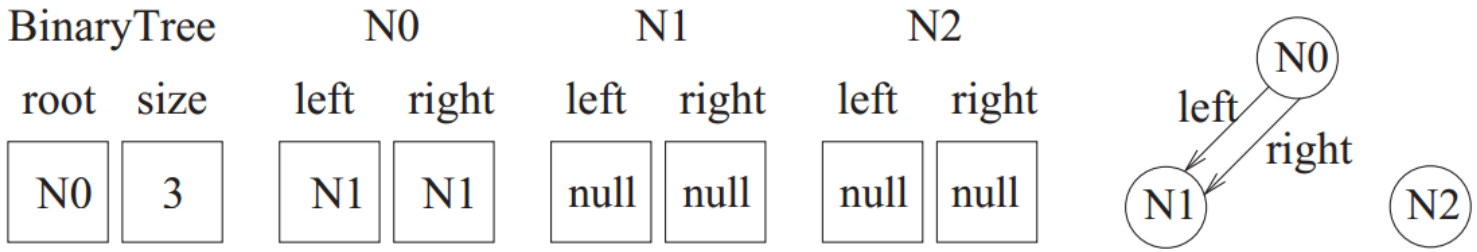
`boolean add(E e)`

Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)



Korat: Search Algorithm

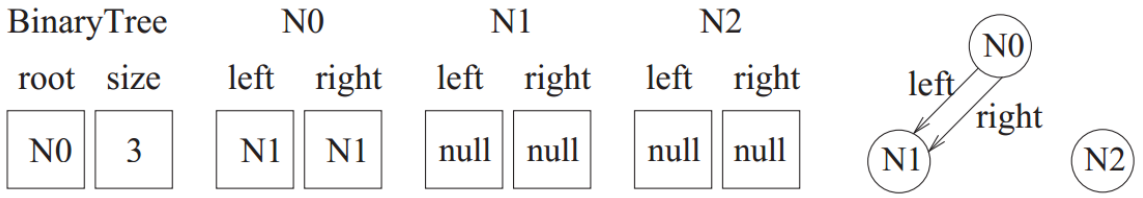
3. Invoke repOk() to check if the candidate is valid, e.g., [1,0,2,2,0,0,0,0] is invalid



4. Backtrack to generate the next candidate in line, e.g., [1,0,2,2,0,0,0,1]

Optimization 1

- During the execution of repOk, Korat monitors the fields that repOk accesses.
  - e.g., [0, 2, 3] for the following example

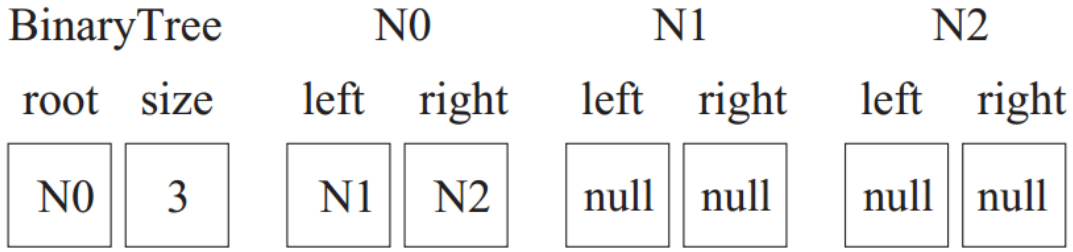


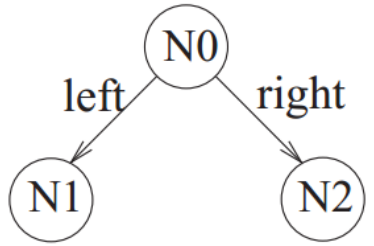
- If repOk() results in false, backtrack until the accessed fields are different
  - e.g., try [1,0,2,3,0,0,0,0] after [1,0,2,2,0,0,0,0]

Theory

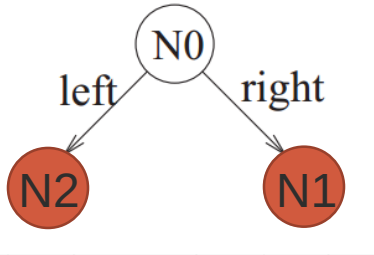
- For non-deterministic repOk methods,
  - All candidates for which repOk() always returns true are generated
  - Candidates for which repOk() always returns false are never generated;
- Candidates for which repOk() sometimes returns true and sometimes false may or may not be generated.

Optimization 2



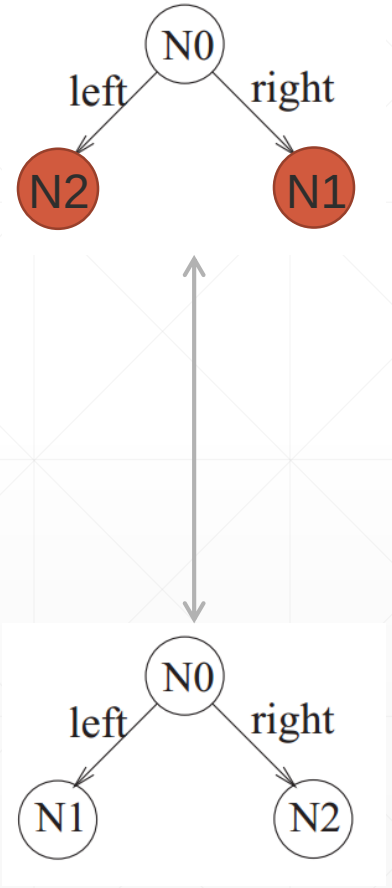


- If we generated the above, we may not want to generate [1, 0, 3, 2, 0, 0, 0, 0].



Vocabulary

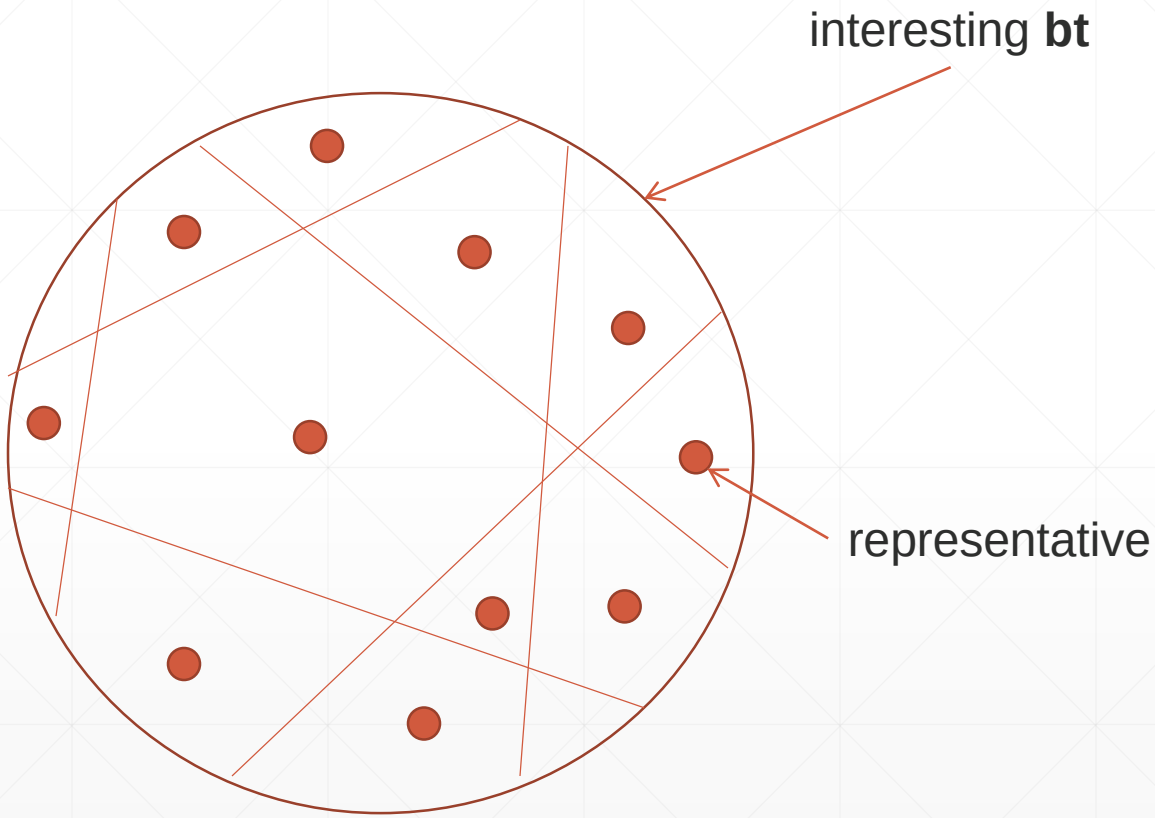
object graph:



Isomorphic: two object graphs C and C’ are isomorphic iff there is a permutation per such that per(C) = C’ and per(C’) = C

- e.g., per = {N1->N2, N2->N1}

Optimization 2



all candidates in the same region are isomorphic

Representative

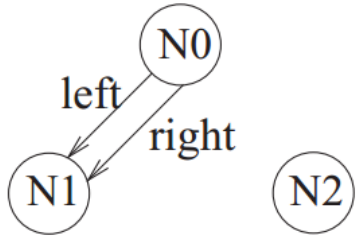
Given the two graphs below

[1, 0, 2, 3, 0, 0, 0, 0] and [1, 0, 3, 2, 0, 0, 0, 0], Korat takes the latter as a representative, as it is “bigger”.

Example

When backtrack from [1, 0, 2, 2, 0, 0, 0, 0],

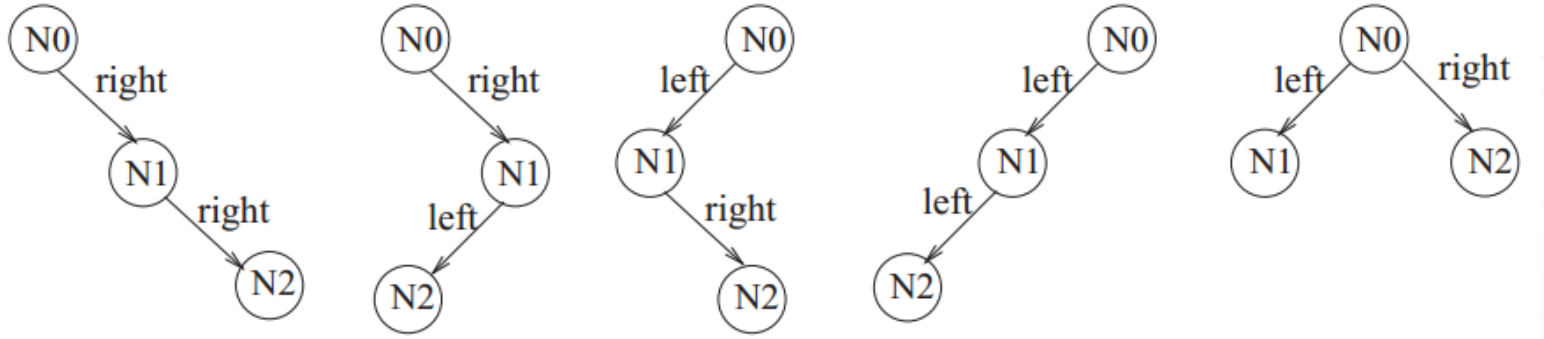
BinaryTree		N0		N1		N2	
root	size	left	right	left	right	left	right
N0	3	N1	N1	null	null	null	null

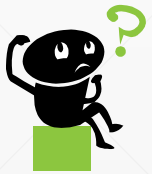


Korat skips [1, 0, 2, 3, 0, 0, 0, 0] (since there is a “bigger” representative [1,0,3,2,0,0,0,0]), and continues with [1,0,3,0,0,0,0,0]

Result

Only 5 **bt** are generated – assuming size is set to 3 always.





Evaluation

benchmark	package	finitization parameters
`BinaryTree`	`korat.examples`	NUM_Node
`HeapArray`	`korat.examples`	MAX_size, MAX_length, MAX_elem
`LinkedList`	`java.util`	MIN_size, MAX_size, NUM_Entry, NUM_Object
`TreeMap`	`java.util`	MIN_size, NUM_Entry, MAX_key, MAX_value
`HashSet`	`java.util`	MAX_capacity, MAX_count, MAX_hash, loadFactor
`AVTree`	`ins.namespace`	NUM_AVPair, MAX_child, NUM_String

Is this biased?

Experiment I

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}
HashSet	7	3.71	2386	193200	2^{119}
	8	16.68	9355	908568	2^{142}
	9	56.71	26687	3004597	2^{166}
	10	208.86	79451	10029045	2^{190}
	11	926.71	277387	39075006	2^{215}
AVTree	5	62.05	598358	1330628	2^{50}

Experiment II

		Korat			Alloy Analyzer		
benchmark	size	struc. gen.	total time	first struc.	inst. gen.	total time	first inst.
BinaryTree	3	5	0.56	0.62	6	2.63	2.63
	4	14	0.58	0.62	28	3.91	2.78
	5	42	0.69	0.67	127	24.42	4.21
	6	132	0.79	0.66	643	269.99	6.78
	7	429	0.97	0.62	3469	3322.13	12.86
HeapArray	3	66	0.53	0.58	78	11.99	6.20
	4	320	0.57	0.59	889	171.03	16.13
	5	1919	0.73	0.63	1919	473.51	39.58
LinkedList	3	5	0.58	0.60	10	2.61	2.39
	4	15	0.55	0.65	46	3.47	2.77
	5	52	0.57	0.65	324	14.09	3.51
	6	203	0.73	0.61	2777	148.73	5.74
	7	877	0.87	0.61	27719	2176.44	10.51
TreeMap	4	8	0.75	0.69	16	12.10	6.35
	5	14	0.87	0.88	42	98.09	18.08
	6	20	1.49	0.98	152	1351.50	50.87
AVTree	2	2	0.55	0.65	2	2.35	2.43
	3	84	0.65	0.61	132	4.25	2.76
	4	5923	1.41	0.61	20701	504.12	3.06

Experiment III

benchmark	method	max. size	test cases generated	gen. time	test time
BinaryTree	remove	3	15	0.64	0.73
HeapArray	extractMax	6	13139	0.87	1.39
LinkedList	reverse	2	8	0.67	0.76
TreeMap	put	8	19912	136.19	2.70
HashSet	add	7	13106	3.90	1.72
AVTree	lookup	4	27734	4.33	14.63

Conclusion

- Korat generates test cases from a specified domain and correctness specification.
- Korat reduces test cases based on
  - pre-condition
  - a simple learning
  - symmetry reduction

Excercise

- Think about how to use Korat to generate test cases for a HeapArray?
- Refer to http://korat.sourceforge.net/tutorial.html for hints.