Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution

Group 15

- Ana Torres 99050
- Kun Fang 105623
- Tobias Schwarz 111575

Reviewing Unit Testing

Definitions

- Object-oriented **unit tests** are programs that test classes.
- A **test case** consists of a fixed sequence of method invocations with fixed arguments that explores a particular aspect of the behavior of the class under test.

Importance of Unit Testing

- Detecting problems early in the development cycle
- Reducing costs by catching bugs early.
- Promoting test-driven development

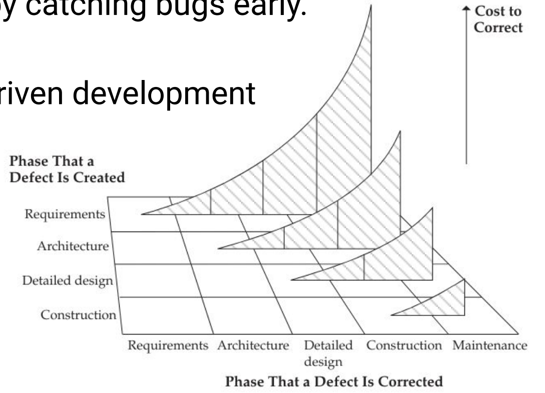


FIGURE 3-5 *Increase in defect cost as time between defect creation and defect correction increases. Effective projects practice "phase containment"—the detection and correction of defects in the same phase in which they are created.*

[Steve McConnell. "Software Project Survival Guide"]

Problem – Popular Testing Approaches

	Tools	Downsides
Random Generation	Jtest, JCrasher and Eclat	Execute same sequences and has poor coverage
Model-Based Testing	AsmLT	Requires: - careful selection of sufficiently large concrete domains - right abstraction functions
Non-Isomorphic Object Graphs	Korat	Doesn’t ensure good coverage

Problem - Current Symbolic Execution Solutions

BZ-TT Tool

Uses constraint solving to derive method sequences from B specifications,...

not object-oriented

Khurshid et al.'s Approach

Focuses on generating receiver-object states as object graphs without relying on method sequences

Generates tests faster than model checking of method sequences with concrete arguments due to symbolic representations describing sets of states.

Requires user-provided class invariants to describe an over-approximation of reachable object graphs.

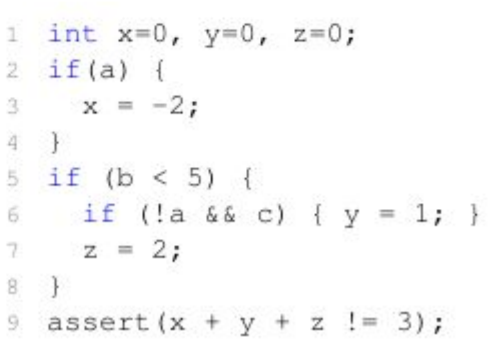
Problem – Test Generation Constraints

- For test generation, states must remain separate to ensure different tests are used for different paths.
- Tools like SLAM and Blast have been adapted for test generation but struggle with complex data structures, which Symstra aims to address.

Proposed Solution - Symstra

Symstra is a framework that creates method sequences using symbolic execution. It explores all possible method sequences with symbolic variables for primitive-type arguments, representing all possible values for each argument. Symstra uses symbolic execution to work with symbolic states, which include these symbolic variables.

Symbolic Execution - An abstract execution of a program that encompasses multiple possible inputs sharing the same execution path



```
1  int x=0, y=0, z=0;
2  if(a) {
3    x = -2;
4  }
5  if (b < 5) {
6    if (!a && c) { y = 1; }
7    z = 2;
8  }
9  assert(x + y + z != 3);
```

a = 1, b = 2, c = 1 **FAIL**

a = α, b = β, c = γ **?**

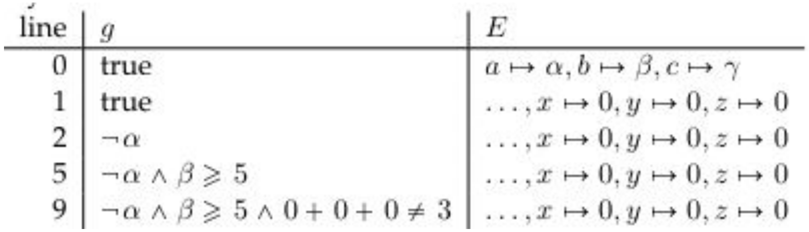
Instead of using specific values, symbolic execution uses symbolic inputs like α, β, and γ. It tracks the execution using these symbolic values.

If a branch condition depends on these unknown values, the symbolic execution engine picks a branch to follow and records the condition for that choice.

After completing one path, the engine can go back and explore other paths through the program.

As we encounter each branch in the code, we build a path condition from these boolean predicates.

The assignment to x, y, and z updates the symbolic state E with expressions for each variable



line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \wedge \beta \geqslant 5 \wedge 0 + 0 + 0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

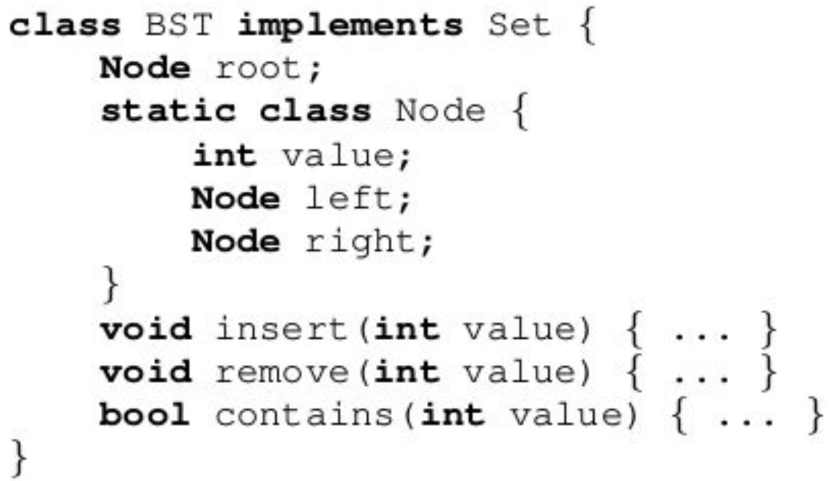
Symbolic Execution

Jonathan Aldrich, and Claire Le Goues : “Lecture Notes: Symbolic Execution”. Program Analysis (Spring 2018), Carnegie Mellon University, 2018.

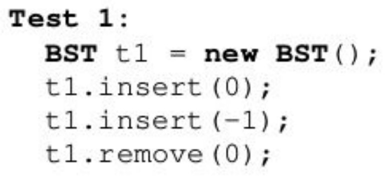
Proposed Solution - Symstra

Symstra is a framework that creates method sequences using symbolic execution. It explores all possible method sequences with symbolic variables for primitive-type arguments, representing all possible values for each argument. Symstra uses symbolic execution to work with symbolic states, which include these symbolic variables.

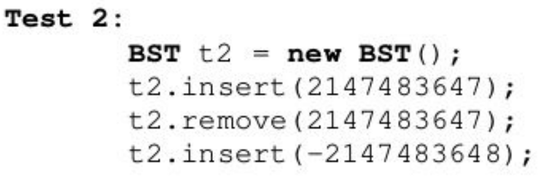
How it works



```
class BST implements Set {
    Node root;
    static class Node {
        int value;
        Node left;
        Node right;
    }
    void insert(int value) { ... }
    void remove(int value) { ... }
    bool contains(int value) { ... }
}
```



```
Test 1:
  BST t1 = new BST();
  t1.insert(0);
  t1.insert(-1);
  t1.remove(0);
```



```
Test 2:
      BST t2 = new BST();
      t2.insert(2147483647);
      t2.remove(2147483647);
      t2.insert(-2147483648);
```

Tools usually test by checking output correctness, which depends on design-by-contract annotations translated into run-time assertions, or model-based testing. If these aren't present, they simply check code robustness by running tests for uncaught exceptions. Some tools can also explore all method sequences up to a set length.

What arguments to use for method calls?

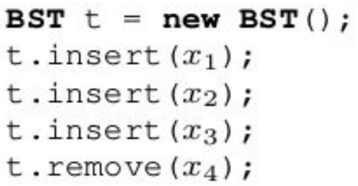
How to determine equivalent tests?

Symbolic execution operates on a symbolic state consisting of:

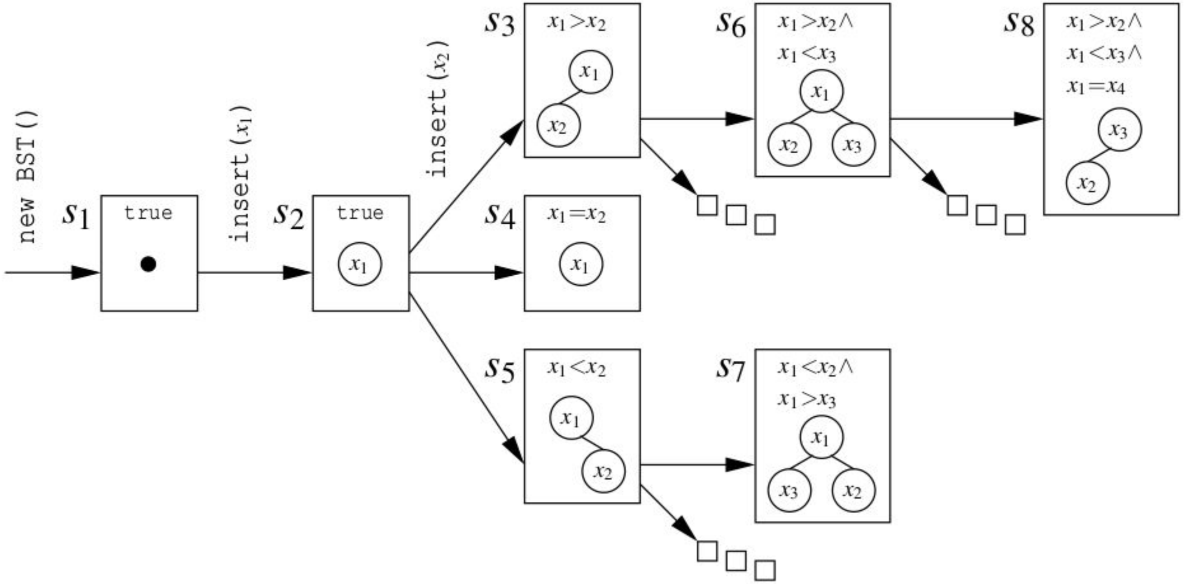
- A constraint, called the path condition, that must hold for execution to reach a certain point
- A heap containing symbolic variables

When symbolic execution encounters a branch, it explores both outcomes, adding the branch condition or its negation to the constraint as needed.

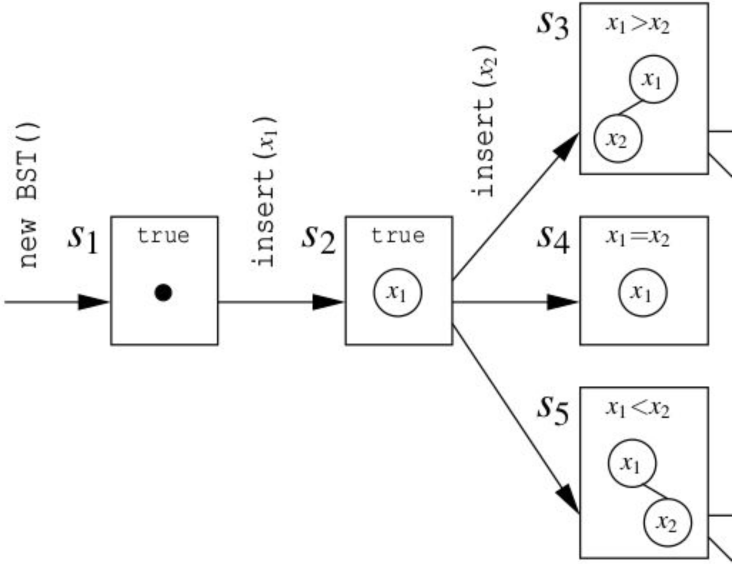
Sequence with concrete arguments => one state
Sequence with symbolic arguments => several states (execution tree)

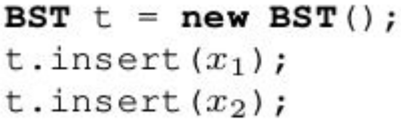


```
BST t = new BST();
t.insert(x_1);
t.insert(x_2);
t.insert(x_3);
t.remove(x_4);
```



- Part of the execution tree

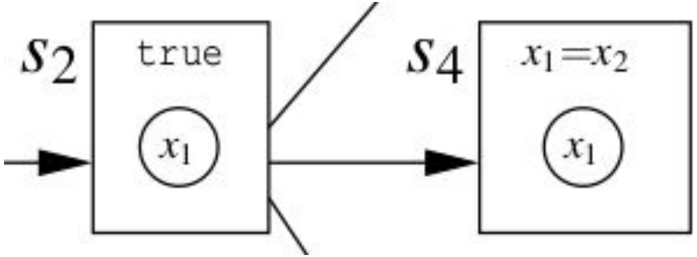




```
BST t = new BST();
t.insert(x1);
t.insert(x2);
```

Each state has a heap and a constraint that must hold for that heap to be created.

1. The constructor first creates an empty tree.
2. Insert: adds the element x1 to the tree.
3. Insert: produces states s3 , s4 , and s5 : if x1 = x2 , the tree does not change, and if x2 > x1 (or x2 < x1), x2 is added in the right (or left) subtree.

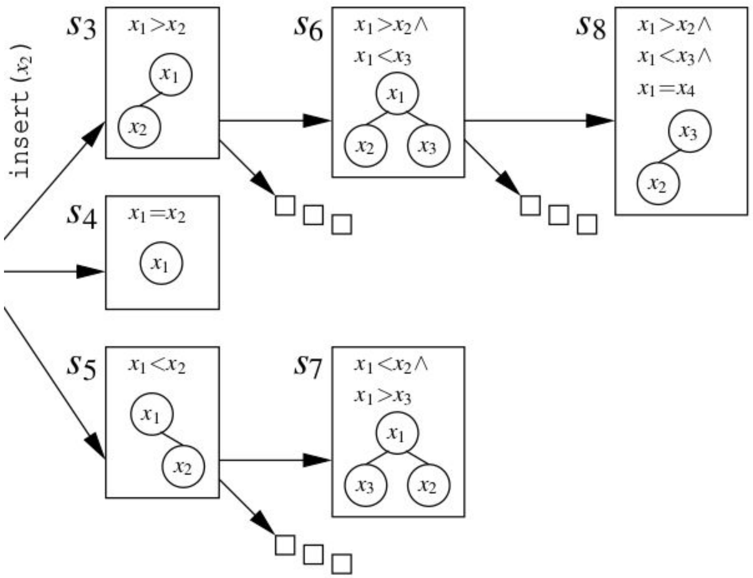


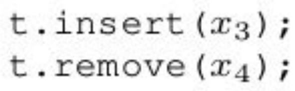
- ➔ Syntactically different, semantically the same

  can be turned into the same concrete heaps by giving x1 and x2 values that meet the constraints.

- ➔ Instead of checking state equivalence, Symstra checks state **subsumption** => s2 subsumes s4 because the set of concrete heaps of s4 is a subset of the set of concrete heaps of s2.

- ➔ Verify that the implication x1 = x2 ⇒ true holds. (current implementation uses Omega library and CVC Lite to check the validity of the implication.)





```
t.insert(x3);
t.remove(x4);
```

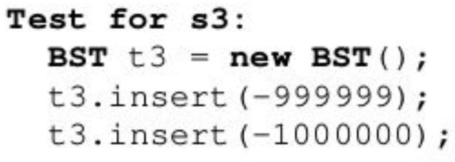
4. Insert: generates several symbolic states. Symstra applies insert only on s3 and s5 (not on s4). We focus on s6 and s7, which are syntactically different but semantically equivalent; swapping variables x2 and x3 results in the same state. Symstra detects this by checking that s6 and s7 are **isomorphic**
5. Remove: s8 is subsumed by a previously explored state, s3

This example has illustrated how Symstra would explore symbolic execution for one particular sequence. Symstra actually exhaustively explores the symbolic execution tree for all sequences up to a given length, pruning the exploration based on subsumption.

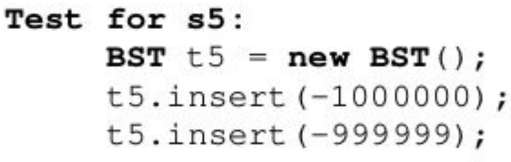
Symbolic state s -> Symstra creates a specific test with concrete arguments to generate a concrete heap for s.

Traverse shortest path from the root of the symbolic execution tree to s and note the method calls encountered.

To determine concrete arguments for these calls, Symstra uses a constraint solver. The current implementation uses the POOC solver.



```
Test for s3:
  BST t3 = new BST();
  t3.insert(-999999);
  t3.insert(-1000000);
```



```
Test for s5:
     BST t5 = new BST();
     t5.insert(-1000000);
     t5.insert(-999999);
```

Idea Symbolic Execution

Symbolic States

State defined as symbolic expressions and constraints.

A symbolic state consists of a constraint and a symbolic heap

Symbolic heaps are graph-based representations of objects and their fields, which can contain symbolic variables

Heap Isomorphism.

Two heaps are isomorphic if they have the same fields for all objects and equal (up to renaming) symbolic expressions for all primitive fields.

Use integer sequences, making the isomorphism test more efficient.

Idea Symbolic Execution

State Subsumption

A symbolic state subsumes another if every concrete heap of the second state can be replaced by a concrete heap of the first state.

Symstra uses an algorithm to check the constraints to efficiently verify subsumption.

Symbolic Execution

Symbolic execution explores all possible execution paths of a program.

The number of paths can be infinite, so heuristics are used to explore only some paths.

Symstra symbolically executes methods by rewriting the code to operate on symbolic expressions.

Idea Symbolic Execution

Symbolic State Exploration

A symbolic state subsumes another if every concrete heap of Symstra systematically explores the symbolic state space for method sequences.

It uses a breadth-first search and a queue to process states and checks subsumption to prune the exploration.

Concrete Test Generation

During symbolic state exploration, Symstra generates concrete tests that lead to the explored states.

Symstra instantiates symbolic tests using a constraint solver to get concrete arguments for the sequences.

These tests are exported to JUnit test classes and include the associated constraints as comments.

Key Features

VC Lite: An automatic theorem prover for constraint checking.

Omega Library: A faster, but less general, constraint solver for linear arithmetic constraints.

Breadth-First Search: A search strategy for systematic state space exploration.

Subsumption Pruning: Reduces the number of states to be explored by eliminating redundantly subsumed states.

Evaluation

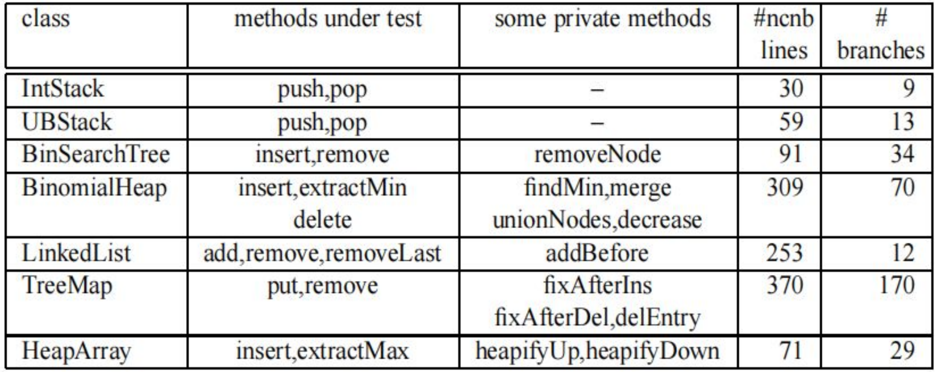
- **Generate tests up to N(1..8) iterations**

  Concrete-State(Rostra) vs. Symstra

- **Pentium IV 2.8 GHz, Java 2 JVM with 512 MB**

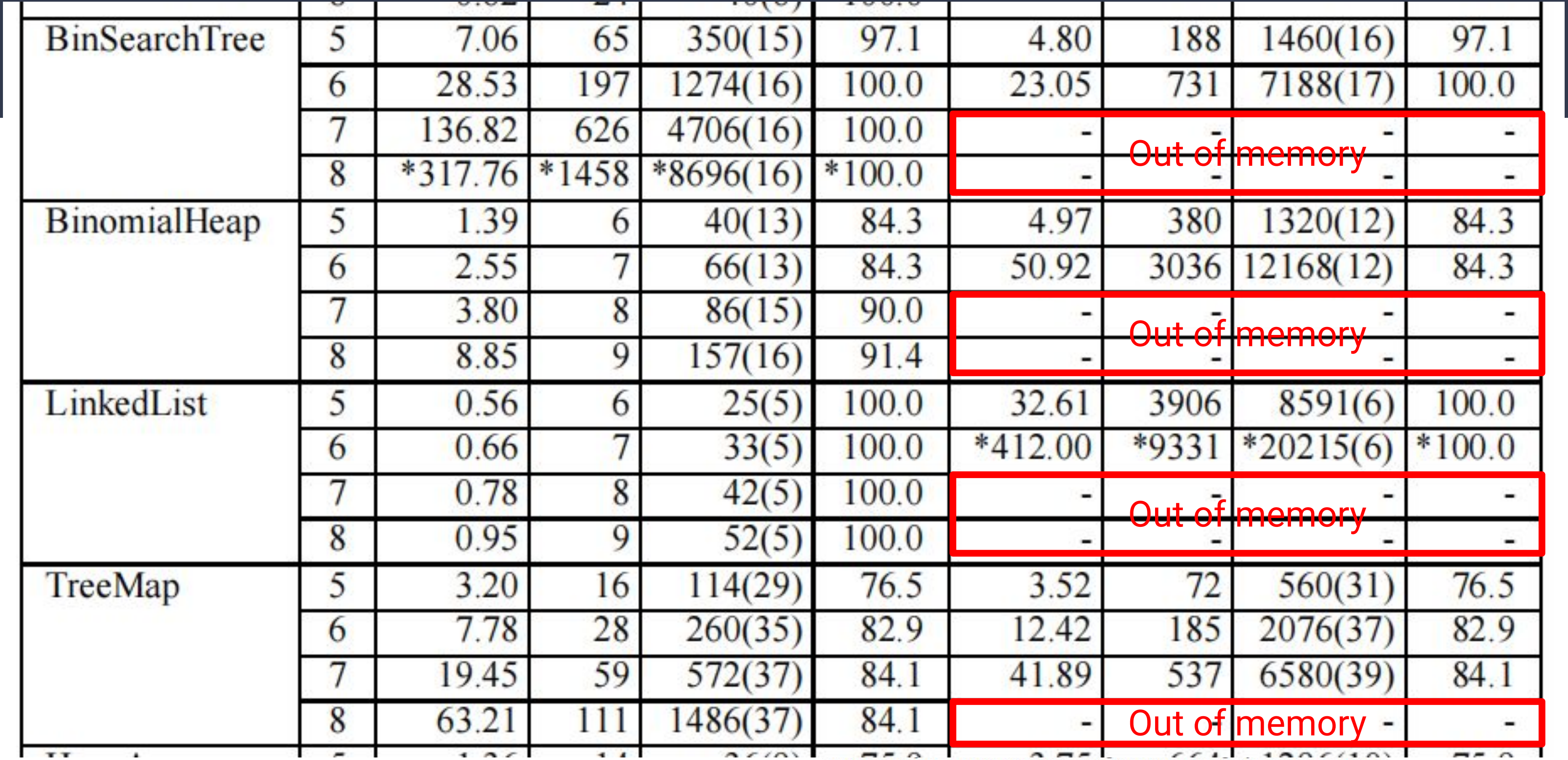
- **Focus on the key methods (e.g., add,remove) of seven Java classes from various sources**

  most are complex data structures



class	methods under test	some private methods	#ncnb lines	# branches
IntStack	push,pop	–	30	9
UBStack	push,pop	–	59	13
BinSearchTree	insert,remove	removeNode	91	34
BinomialHeap	insert,extractMin delete	findMin,merge unionNodes,decrease	309	70
LinkedList	add,remove,removeLast	addBefore	253	12
TreeMap	put,remove	fixAfterIns fixAfterDel,delEntry	370	170
HeapArray	insert,extractMax	heapifyUp,heapifyDown	71	29

Table 1. Experimental subjects



BinSearchTree	5	7.06	65	350(15)	97.1	4.80	188	1460(16)	97.1
	6	28.53	197	1274(16)	100.0	23.05	731	7188(17)	100.0
	7	136.82	626	4706(16)	100.0	-	-	-	-
	8	*317.76	*1458	*8696(16)	*100.0	-	-	-	-
BinomialHeap	5	1.39	6	40(13)	84.3	4.97	380	1320(12)	84.3
	6	2.55	7	66(13)	84.3	50.92	3036	12168(12)	84.3
	7	3.80	8	86(15)	90.0	-	-	-	-
	8	8.85	9	157(16)	91.4	-	-	-	-
LinkedList	5	0.56	6	25(5)	100.0	32.61	3906	8591(6)	100.0
	6	0.66	7	33(5)	100.0	*412.00	*9331	*20215(6)	*100.0
	7	0.78	8	42(5)	100.0	-	-	-	-
	8	0.95	9	52(5)	100.0	-	-	-	-
TreeMap	5	3.20	16	114(29)	76.5	3.52	72	560(31)	76.5
	6	7.78	28	260(35)	82.9	12.42	185	2076(37)	82.9
	7	19.45	59	572(37)	84.1	41.89	537	6580(39)	84.1
	8	63.21	111	1486(37)	84.1	-	-	-	-

Out of memory

Out of memory

Out of memory

Out of memory

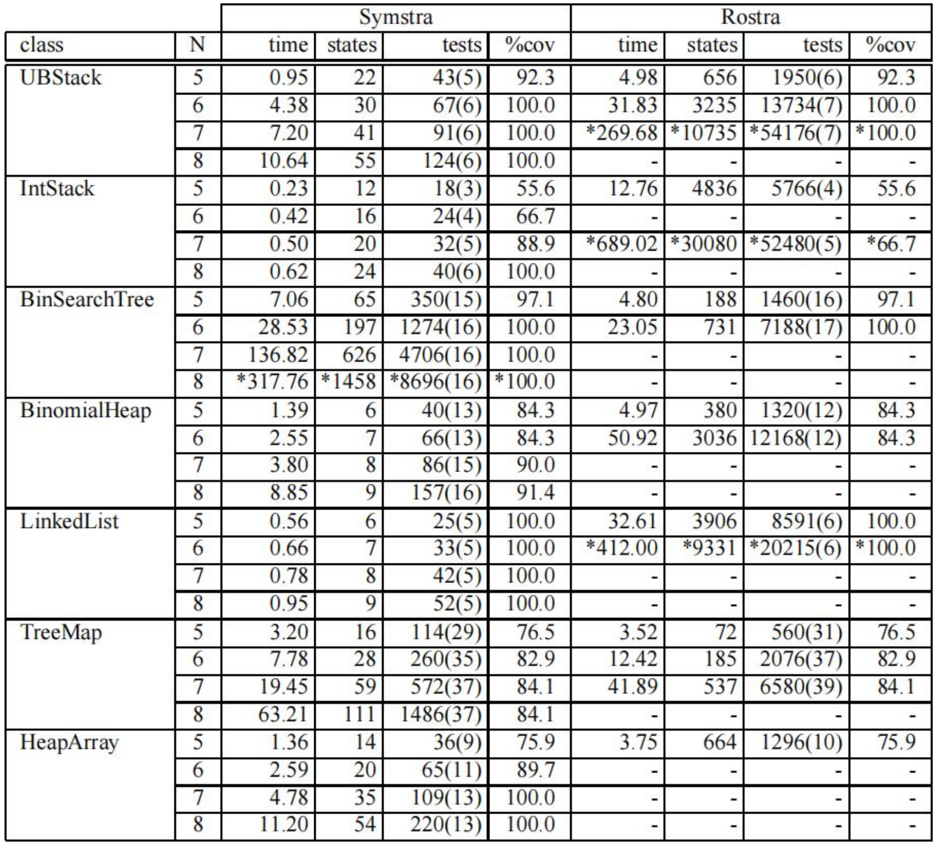
Evaluation

The Result

Experimental results show Symstra effectively:

reduces the state space for exploration

reduces the time for achieving code coverage



		Symstra				Rostra			
class	N	time	states	tests	%cov	time	states	tests	%cov
UBStack	5	0.95	22	43(5)	92.3	4.98	656	1950(6)	92.3
	6	4.38	30	67(6)	100.0	31.83	3235	13734(7)	100.0
	7	7.20	41	91(6)	100.0	*269.68	*10735	*54176(7)	*100.0
	8	10.64	55	124(6)	100.0	-	-	-	-
IntStack	5	0.23	12	18(3)	55.6	12.76	4836	5766(4)	55.6
	6	0.42	16	24(4)	66.7	-	-	-	-
	7	0.50	20	32(5)	88.9	*689.02	*30080	*52480(5)	*66.7
	8	0.62	24	40(6)	100.0	-	-	-	-
BinSearchTree	5	7.06	65	350(15)	97.1	4.80	188	1460(16)	97.1
	6	28.53	197	1274(16)	100.0	23.05	731	7188(17)	100.0
	7	136.82	626	4706(16)	100.0	-	-	-	-
	8	*317.76	*1458	*8696(16)	*100.0	-	-	-	-
BinomialHeap	5	1.39	6	40(13)	84.3	4.97	380	1320(12)	84.3
	6	2.55	7	66(13)	84.3	50.92	3036	12168(12)	84.3
	7	3.80	8	86(15)	90.0	-	-	-	-
	8	8.85	9	157(16)	91.4	-	-	-	-
LinkedList	5	0.56	6	25(5)	100.0	32.61	3906	8591(6)	100.0
	6	0.66	7	33(5)	100.0	*412.00	*9331	*20215(6)	*100.0
	7	0.78	8	42(5)	100.0	-	-	-	-
	8	0.95	9	52(5)	100.0	-	-	-	-
TreeMap	5	3.20	16	114(29)	76.5	3.52	72	560(31)	76.5
	6	7.78	28	260(35)	82.9	12.42	185	2076(37)	82.9
	7	19.45	59	572(37)	84.1	41.89	537	6580(39)	84.1
	8	63.21	111	1486(37)	84.1	-	-	-	-
HeapArray	5	1.36	14	36(9)	75.9	3.75	664	1296(10)	75.9
	6	2.59	20	65(11)	89.7	-	-	-	-
	7	4.78	35	109(13)	100.0	-	-	-	-
	8	11.20	54	220(13)	100.0	-	-	-	-

Table 2. Experimental results of test generation using Symstra and Rostra

contributions

- **Symbolic Sequence Exploration:** Symstra uses symbolic execution to exhaustively generate method sequences with symbolic variables for primitive and reference-type arguments.

- **Symbolic State Comparison:** This new technique allows Symstra to prune object state exploration, thereby speeding up test generation without losing exhaustiveness. Specifically, this pruning preserves intra-method path coverage.

- **Implementation:** We described an implementation of a test generation tool for Symstra that is able to handle dynamically allocated structures, method conditions, and symbolic data but currently lacks concurrency support.

- **Evaluation:** Symstra generates tests faster and achieves better branch coverage than existing techniques, working with ordinary Java implementations without additional user-provided methods.

Discussion and Future Work

Specifications

Symstra utilizes specifications written in Java Modelling Language (JML) to ensure method sequences satisfy pre-conditions and class invariants, enabling bug detection when violated.

- Operating at the bytecode level, Symstra performs black-box testing to explore states.

Discussion and Future Work

Performance

By employing state subsumption, Symstra explores one or more symbolic states with isomorphic heaps, with plans to explore exactly one union symbolic state per isomorphic heap.

- reducing explored states without sacrificing exhaustiveness,
- facilitating exploration of longer method sequences.

Discussion and Future Work

Limitation

Symbolic execution has limitations, particularly with handling array indexes as symbolic variables.

- Two solutions

Conclusion

Automated test-input generation needs to produce:

- method sequences building relevant receiver object states
- relevant method arguments

Symstra exhaustively explores method sequences with symbolic arguments

- prune exploration based on state subsumption
- generate concrete arguments using a constraint solver

The experimental results show Symstra's effectiveness over the existing concrete-state exploration approaches

Questions