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

Tao Xie[1], Darko Marinov[2], Wolfram Schulte[3] and David Notkin[1]

[1] University of Washington
[2] University of Illinois at Urbana-Champaign
[3] Microsoft Research

Motivations

- Object-oriented unit tests consist of sequences of method invocations.

- Behavior of an invocation depends on the method’s arguments and the state of the receiver at the beginning of the invocation.

- Automated test-input generation needs to produce:
  - Method sequences building relevant receiver object states
  - Relevant method arguments

Motivations

- Object-oriented unit tests consist of sequences of method invocations.

- Behavior of an invocation depends on the method's arguments and the state of the receiver at the beginning of the invocation.

- Automated test-input generation needs to produce:
  - Method sequences building relevant receiver object states
  - Relevant method arguments

Symstra achieves both tasks using symbolic execution of method sequences with symbolic arguments

Outline

- Motivations
- Example
- Test generation by exploring concrete states
- Symstra: exploring symbolic states
- Evaluation
- Conclusion

Binary Search Tree Example

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

Previous Test-Generation Approaches

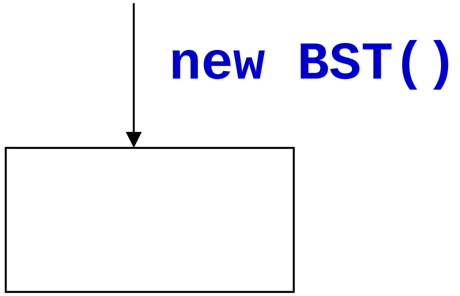
- Straightforward approach: generate all (bounded) possible sequences of calls to the methods under test
  - too many and many are redundant [Xie et al. 04]

```
Test 1:                  Test 2:
BST t1 = new BST();      BST t2 = new BST();
t1.size();               t2.size();
                         t2.size();
```

- Concrete-state exploration approach [Willem et al. 04, Xie et al. 04]
  - assume a given set of method call arguments
  - explore new receiver-object states with method calls (in breadth-first manner)

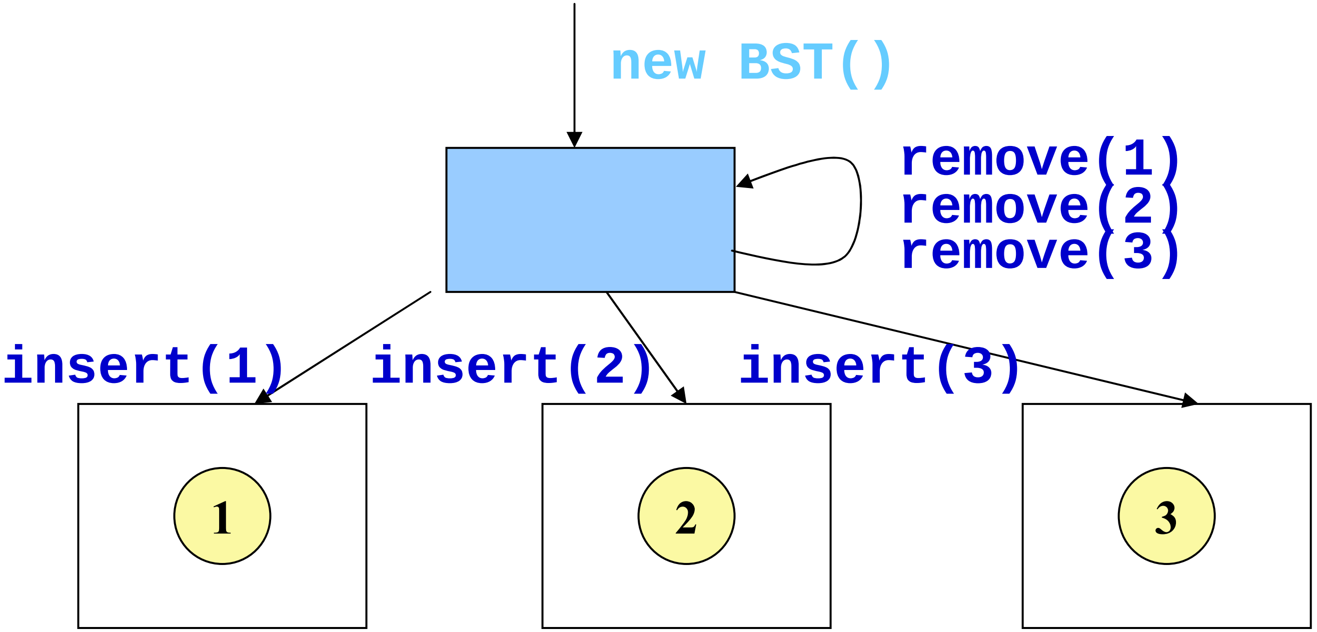
Exploring Concrete States

- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`



Exploring Concrete States

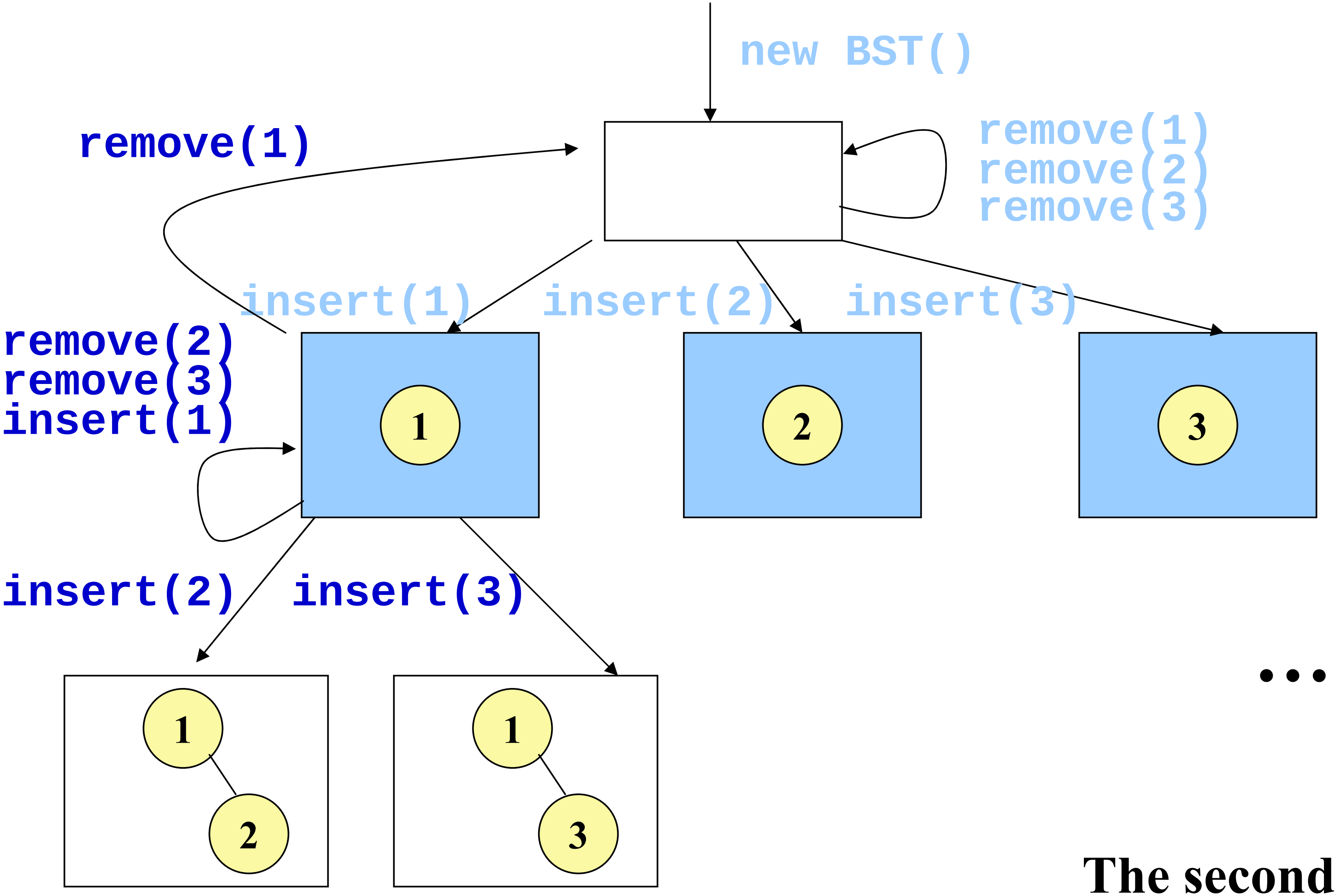
- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`



The first iteration

Exploring Concrete States

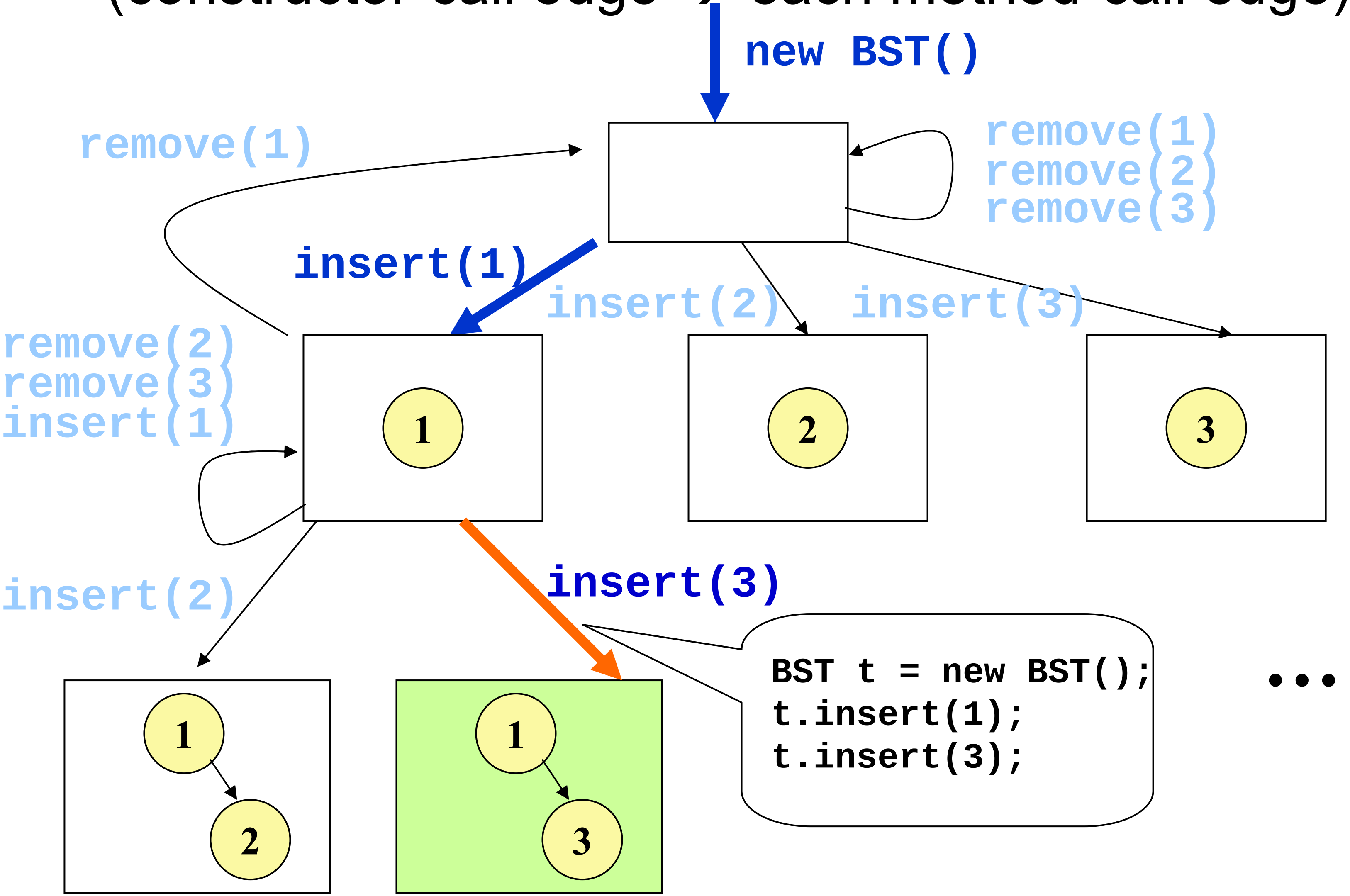
- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`



The second iteration

Generating Tests from Exploration

- Collect method sequence along the shortest path (constructor-call edge → each method-call edge)

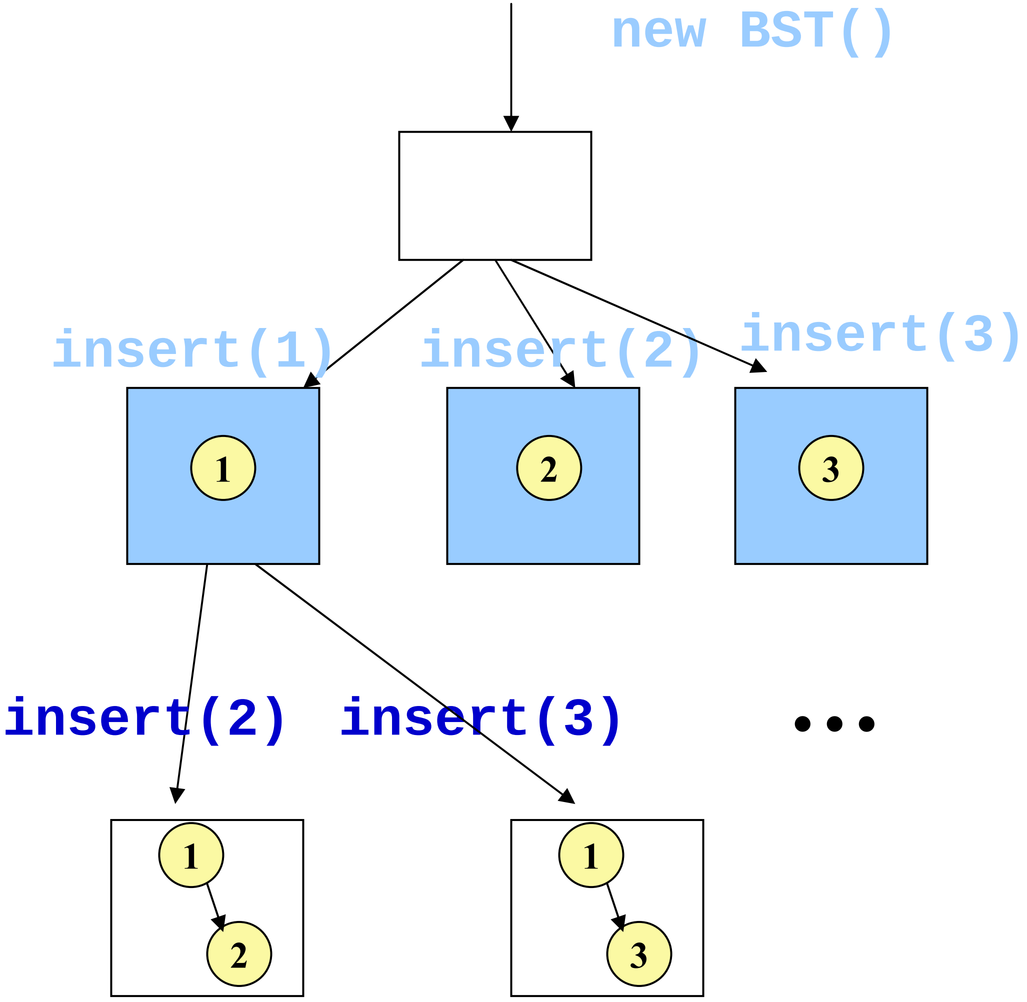


```
BST t = new BST();
t.insert(1);
t.insert(3);
```

Issues of Concrete-State Exploration

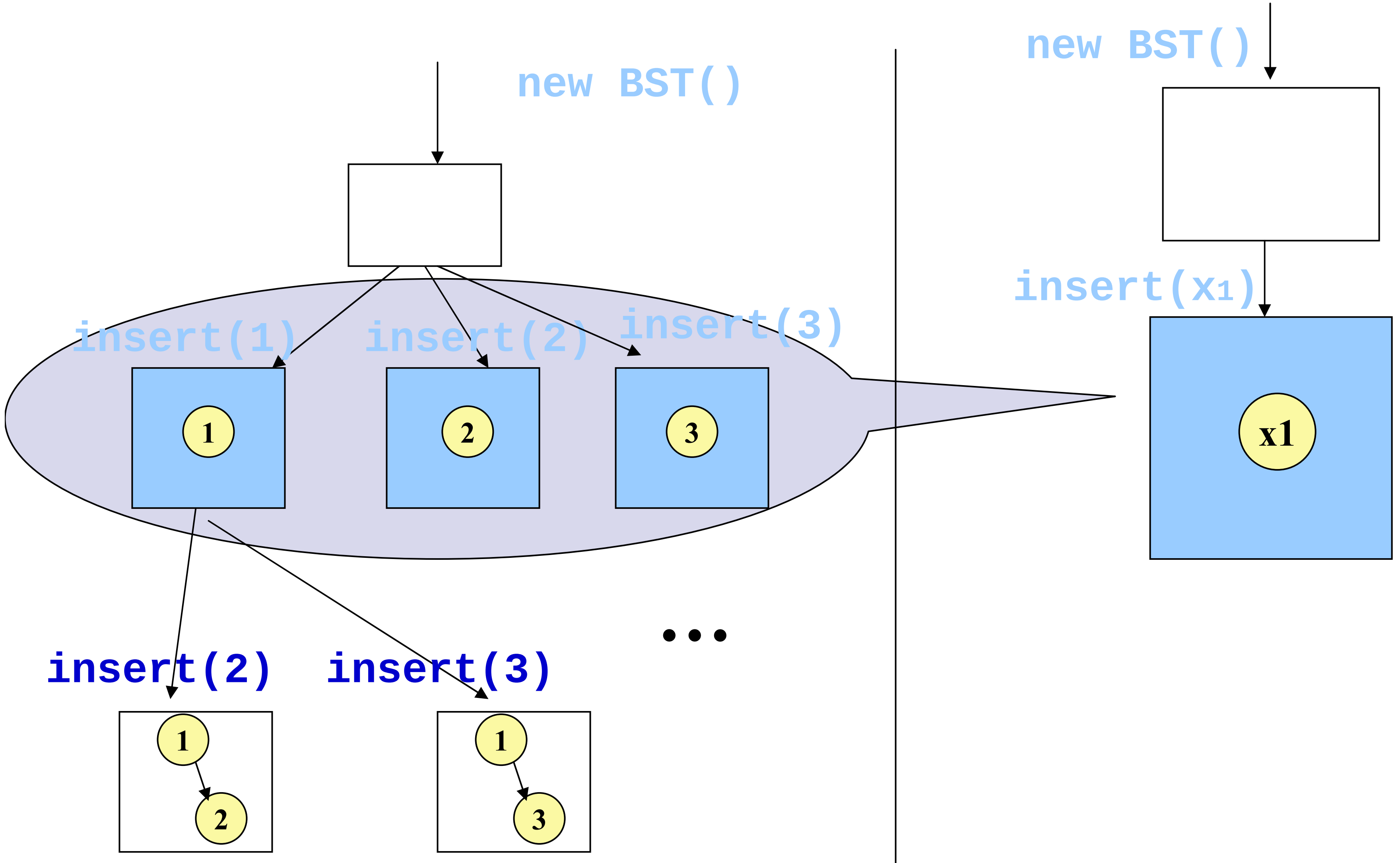
- State explosion (still)
  - need at least *N* different **`insert`** arguments to reach a BST with size *N*
  - run out of memory when *N* reaches 7
- Relevant-argument determination
  - assume a set of given relevant arguments
    - e.g., **`insert(1), insert(2), insert(3), etc.`**

Exploring Concrete States



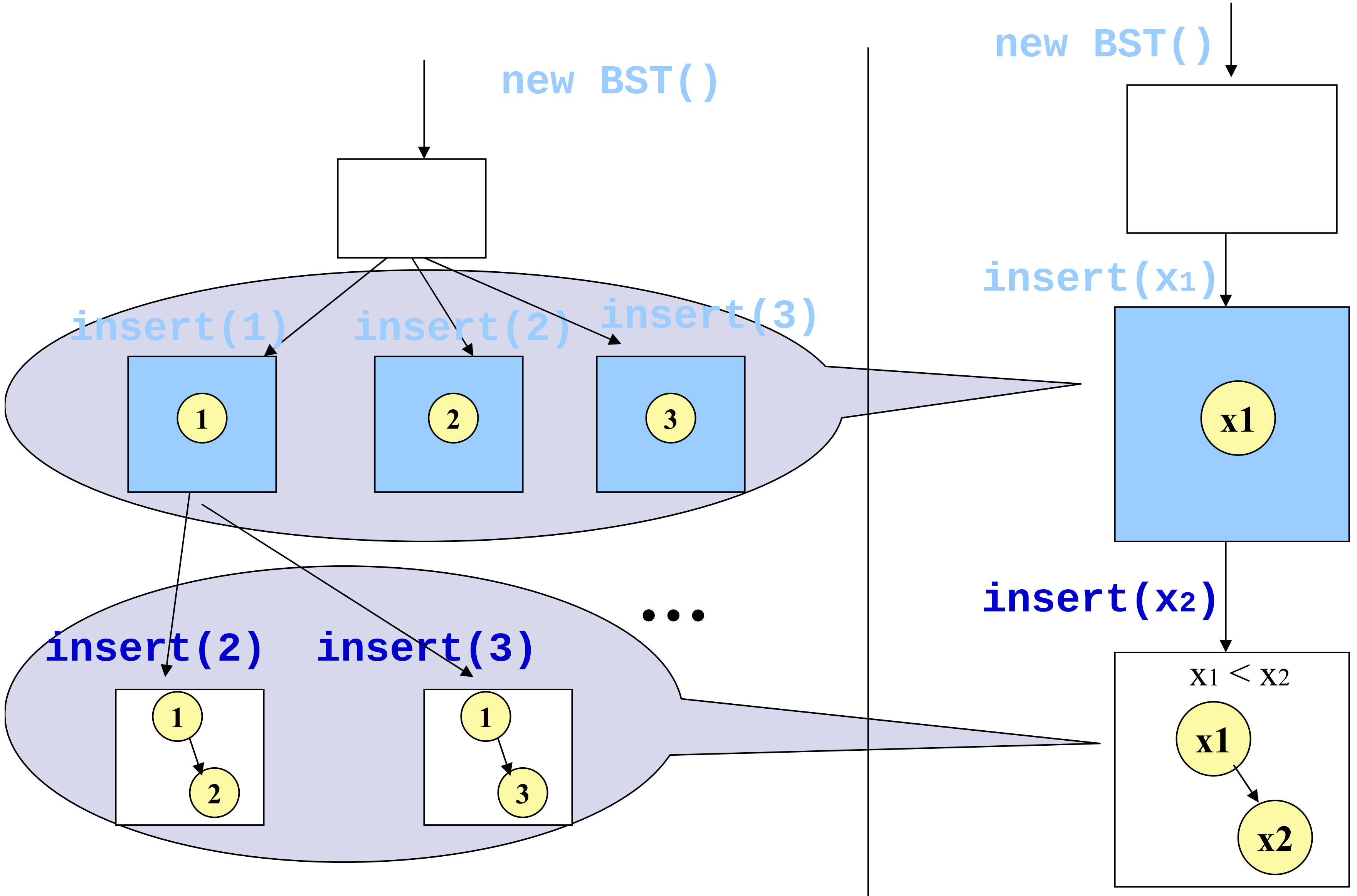
The second iteration

State Abstraction: Symbolic States



The second iteration

State Abstraction: Symbolic States

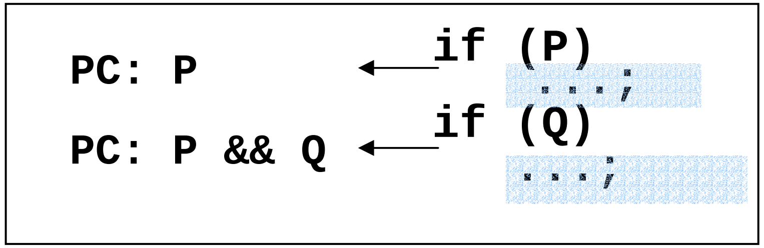


The second iteration

Symbolic Execution

- Execute a method on symbolic input values
  - inputs: `insert(SymbolicInt x)`

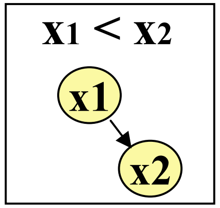
- Explore paths of the method



```
PC: P            <---if (P)
                      ...;
PC: P && Q       <---if (Q)
                     ...;
```

- Build a path condition for each path
  - conjunct conditionals or their negations

- Produce symbolic states (<heap, path condition>)
  - e.g.,

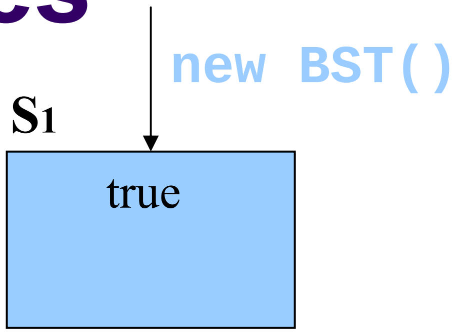


Symbolic Execution Example

```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...

    } else return;
  }
 }
 size++;
}
```

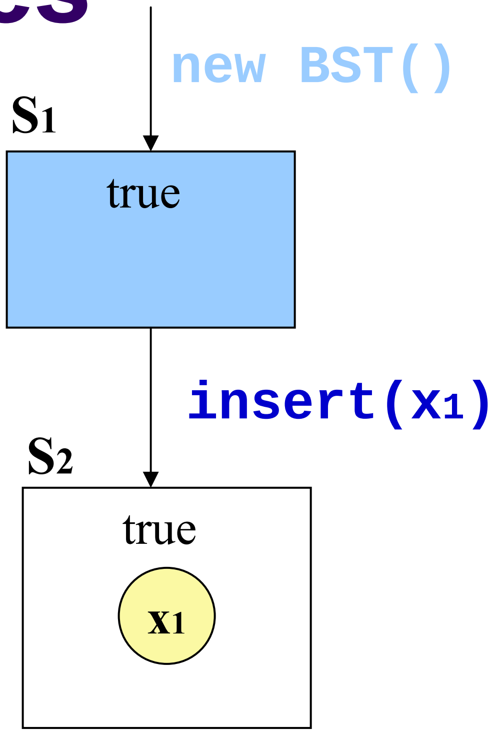
Exploring Symbolic States

```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...

    } else return;
  }
 }
 size++;
}
```



Exploring Symbolic States

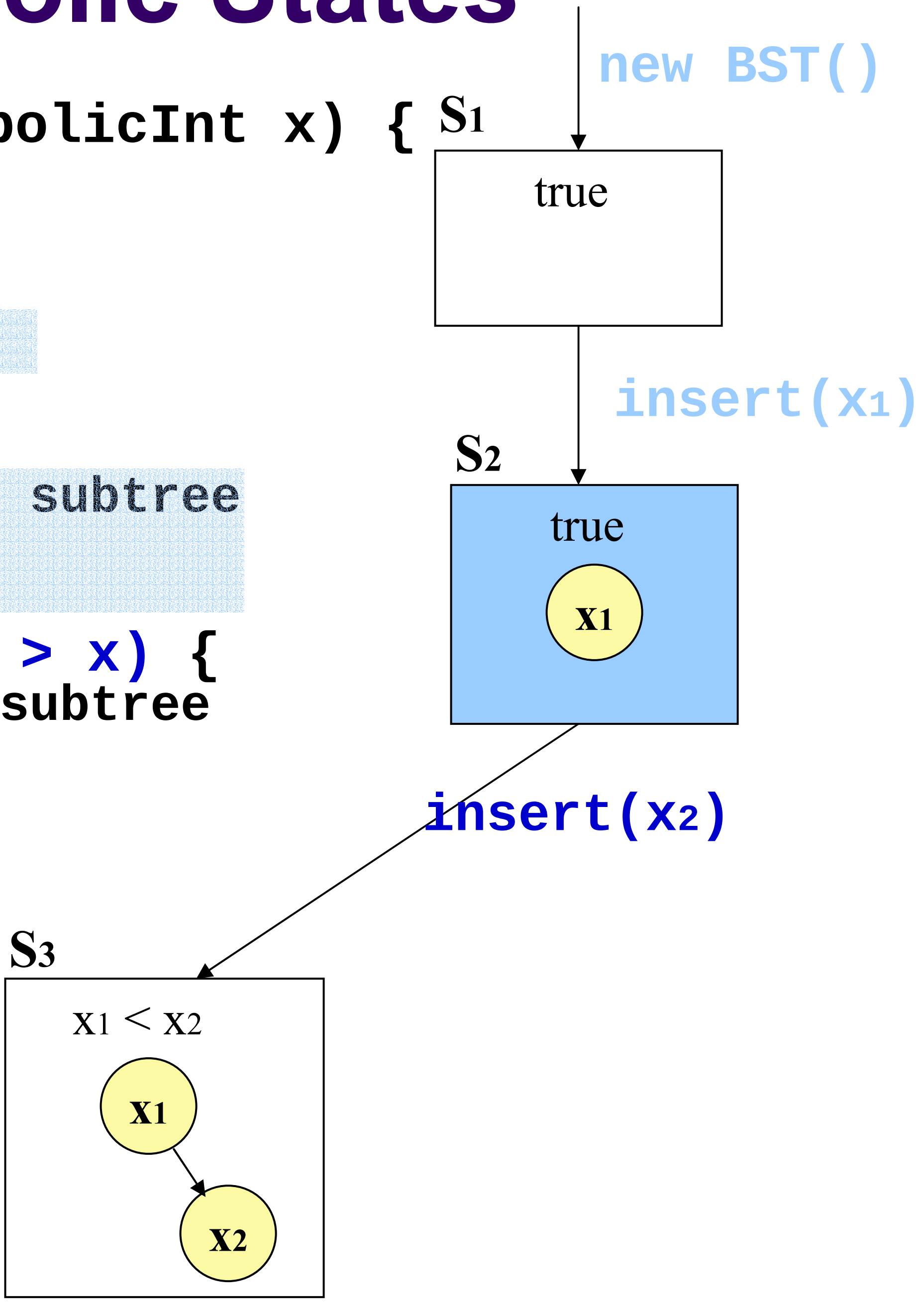
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...

    } else return;
  }
 }
 size++;
}
```



The first iteration

Exploring Symbolic States

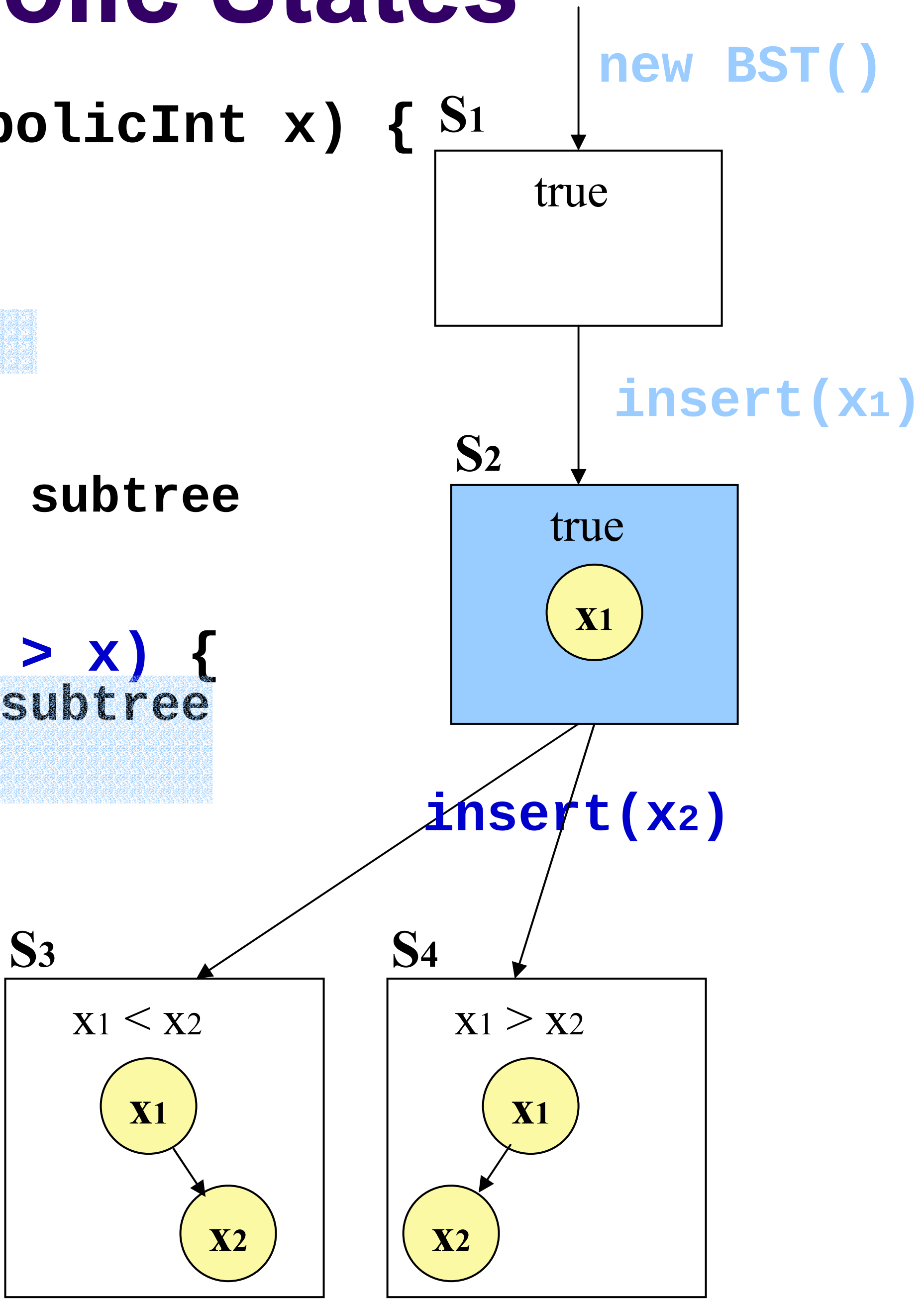
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
     //explore the right subtree
      ...

    } else if (t.value > x) {
     //explore the left subtree
      ...

    } else return;
  }
 }
 size++;
}
```



The second iteration

Exploring Symbolic States

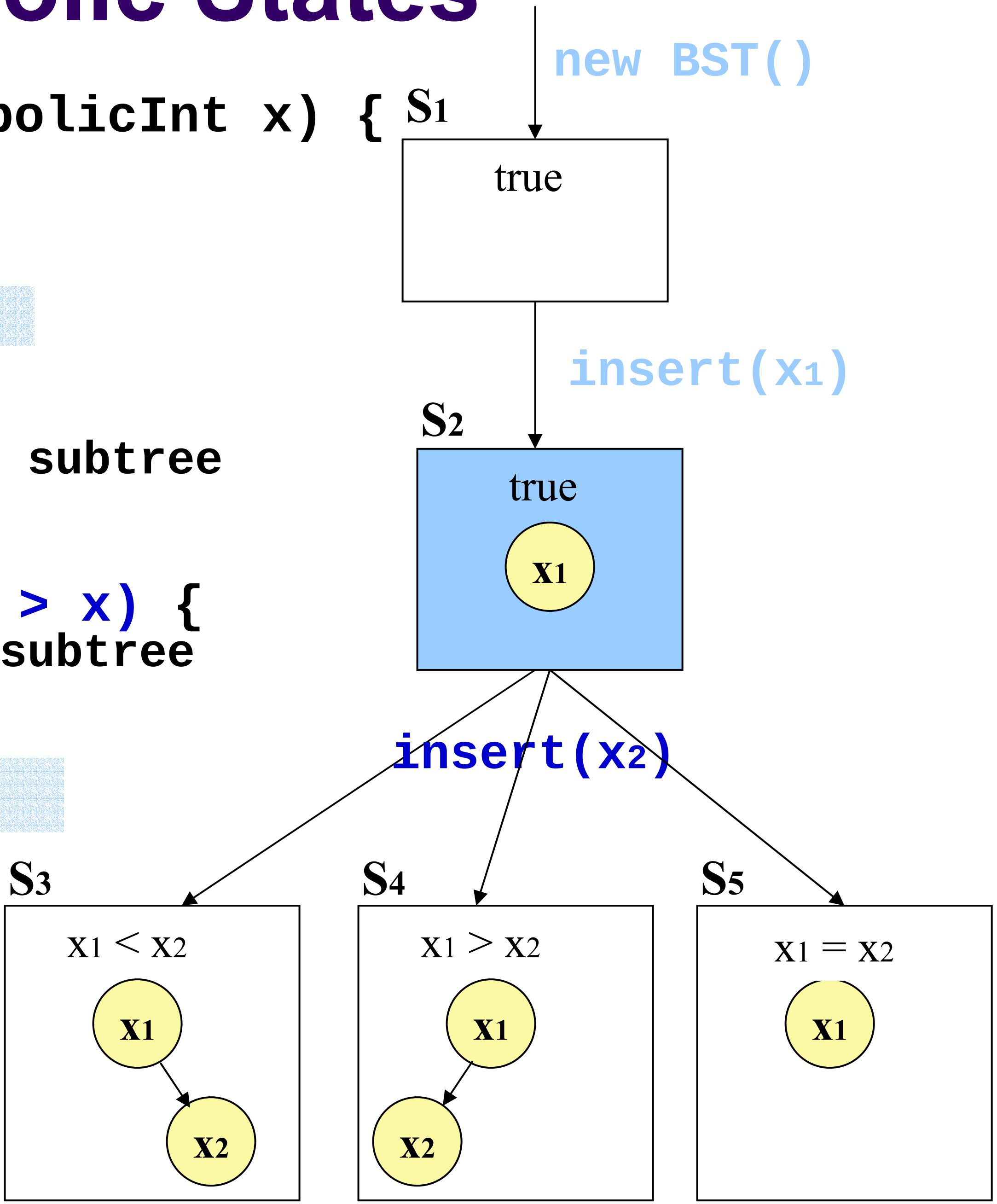
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...

    } else return;
  }
 }
 size++;
}
```



The second iteration

Exploring Symbolic States

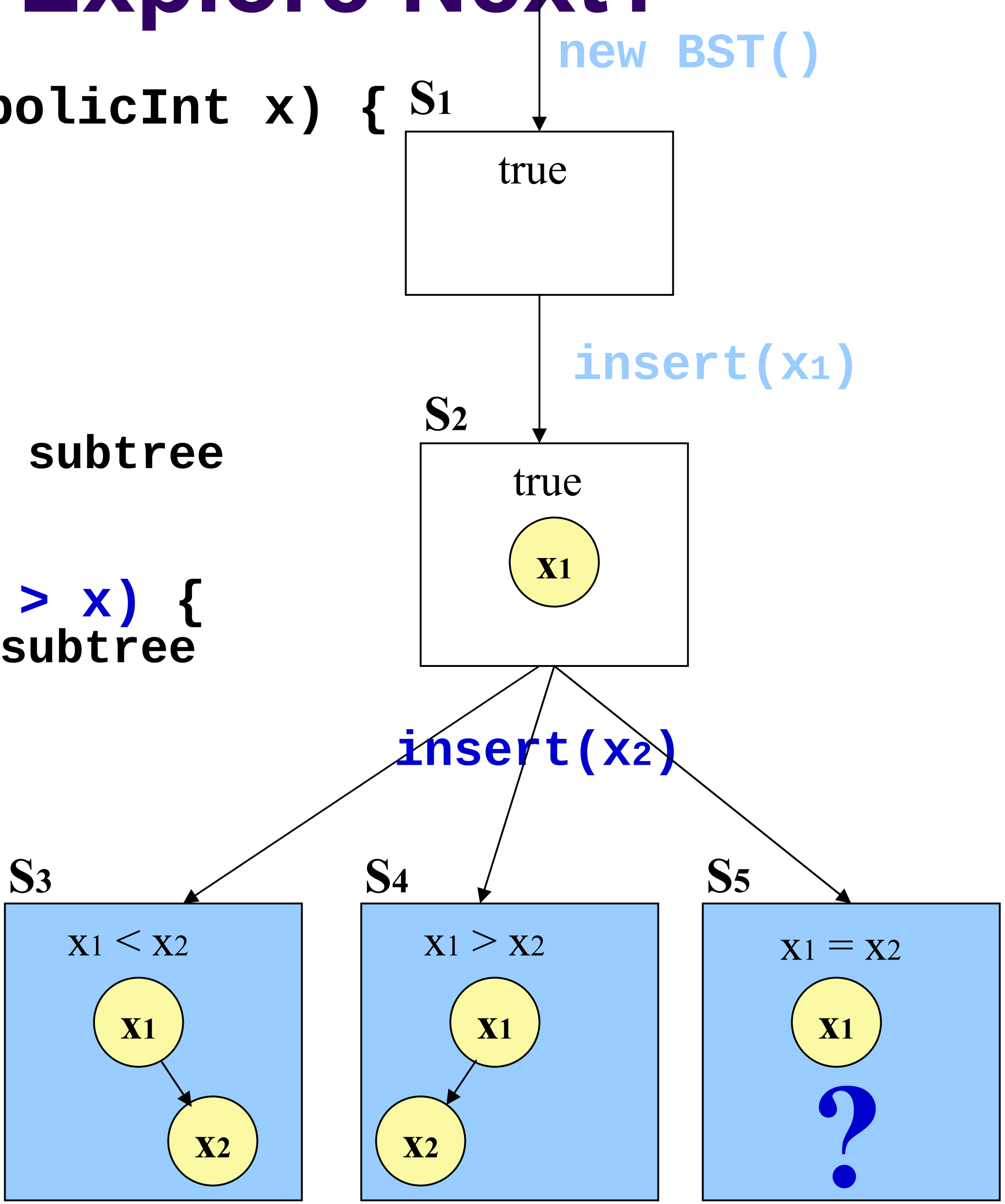
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
     //explore the right subtree
      ...

    } else if (t.value > x) {
     //explore the left subtree
      ...

    } else return;
  }
 }
 size++;
}
```



The second iteration

Which States to Explore Next?

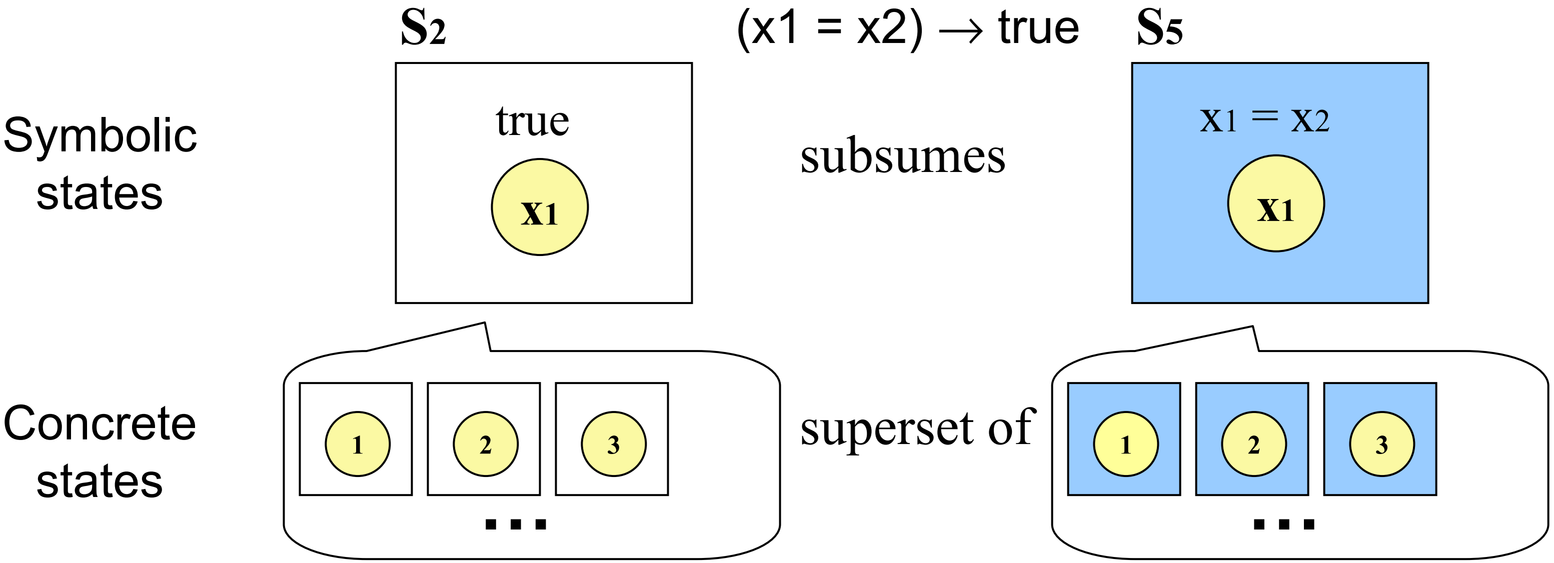
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
     //explore the right subtree
      ...

    } else if (t.value > x) {
     //explore the left subtree
      ...

    } else return;
  }
 }
 size++;
}
```



The third iteration

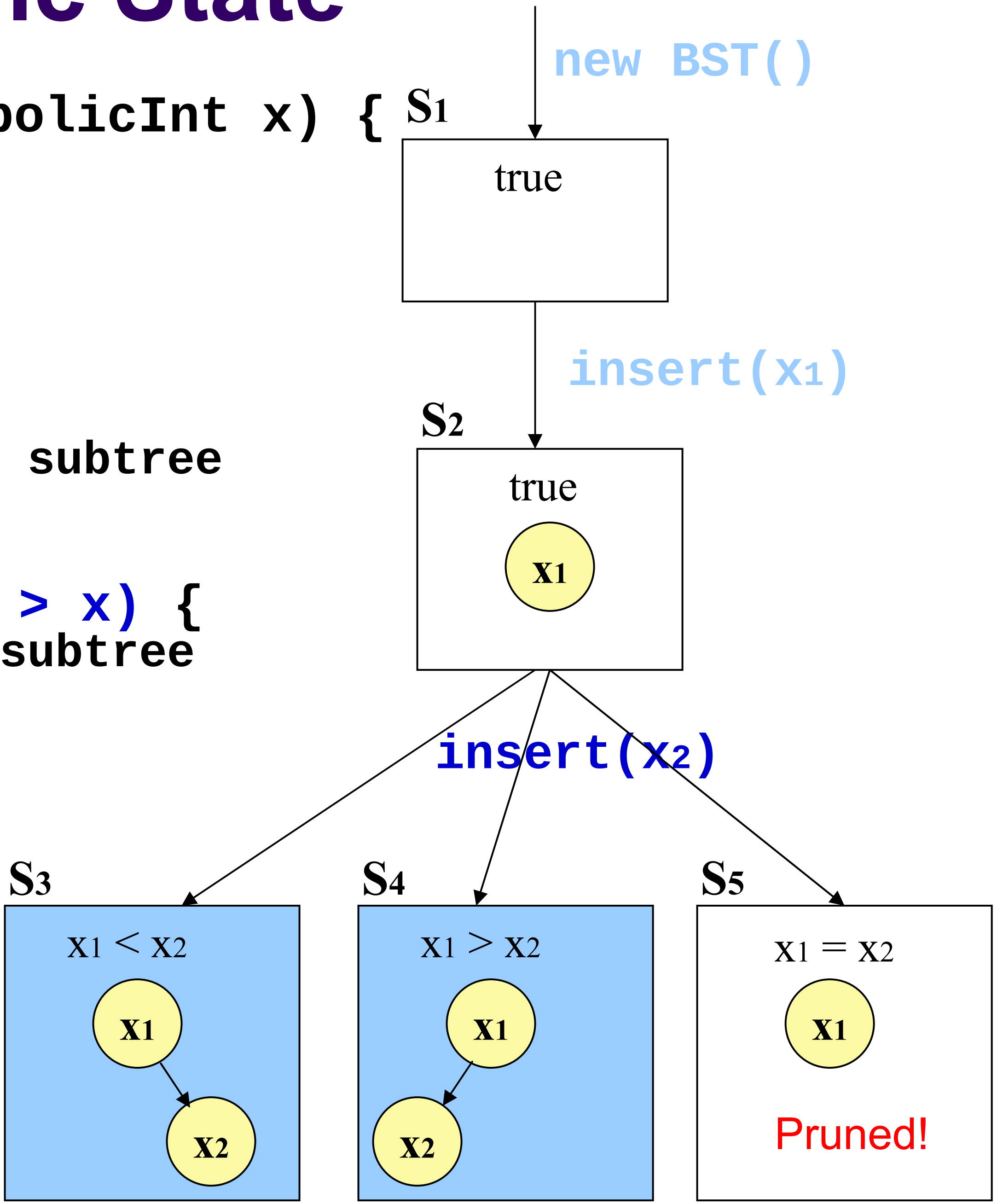
Symbolic State Subsumption

- Symbolic state S_2:<H_2, c_2> subsumes S_5:<H_5, c_5>
  - Heaps H_2 and H_5 are isomorphic
  - Path condition $c_5 \rightarrow c_2$ [CVC Lite, Omega]
- Concrete states represented by S_2 are a superset of concrete states represented by S_5
- If S_2 has been explored, S_5 is pruned.
  - Still guarantee path coverage within a method



Pruning Symbolic State

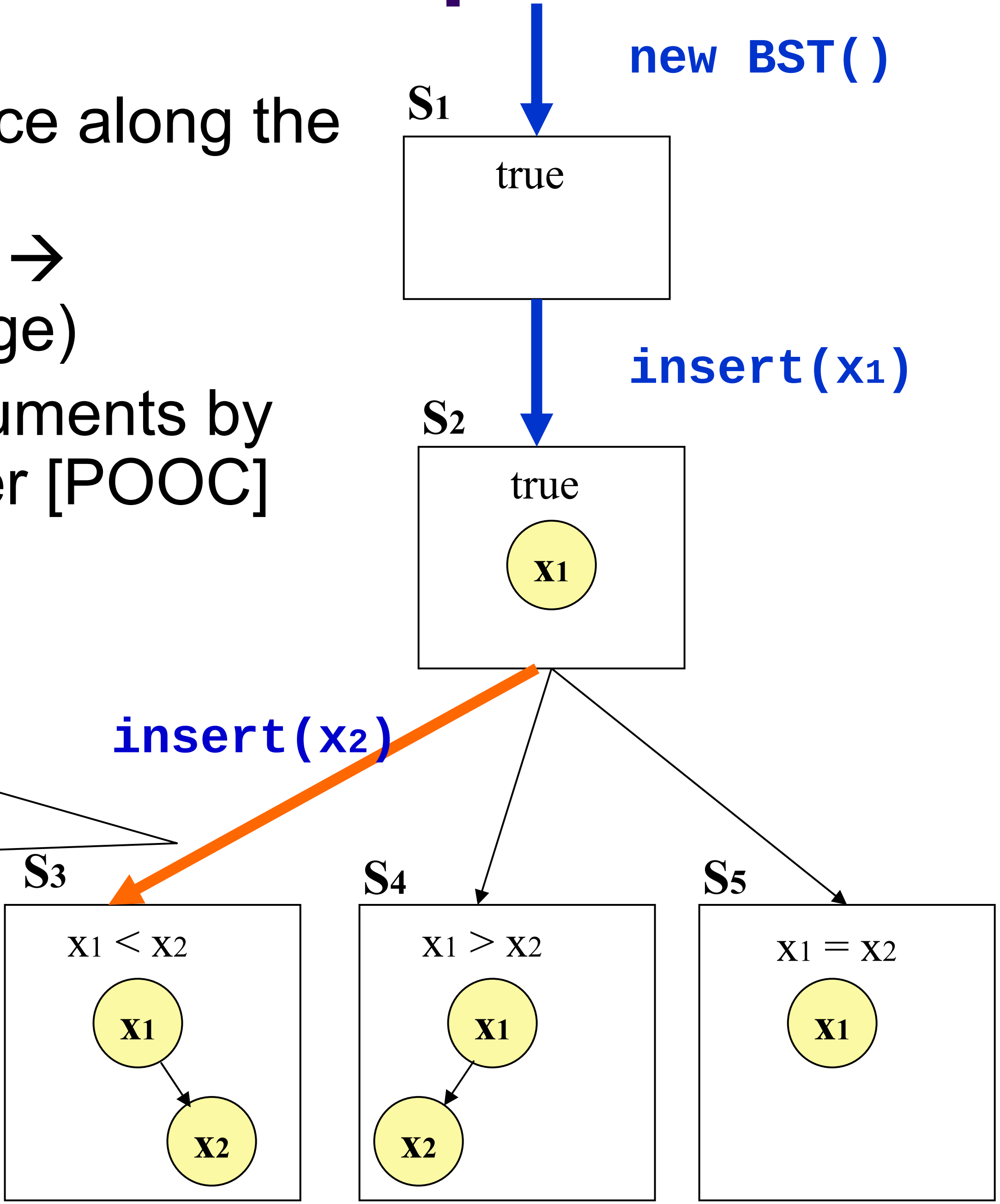
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...

    } else return;
  }
 }
 size++;
}
```



The third iteration

Generating Tests from Exploration

- Collect method sequence along the shortest path
  (constructor-call edge → each method-call edge)
- Generate concrete arguments by using a constraint solver [POOC]



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

x1 < x2

```
BST t = new BST();
t.insert(-1000000);
t.insert(-999999);
```

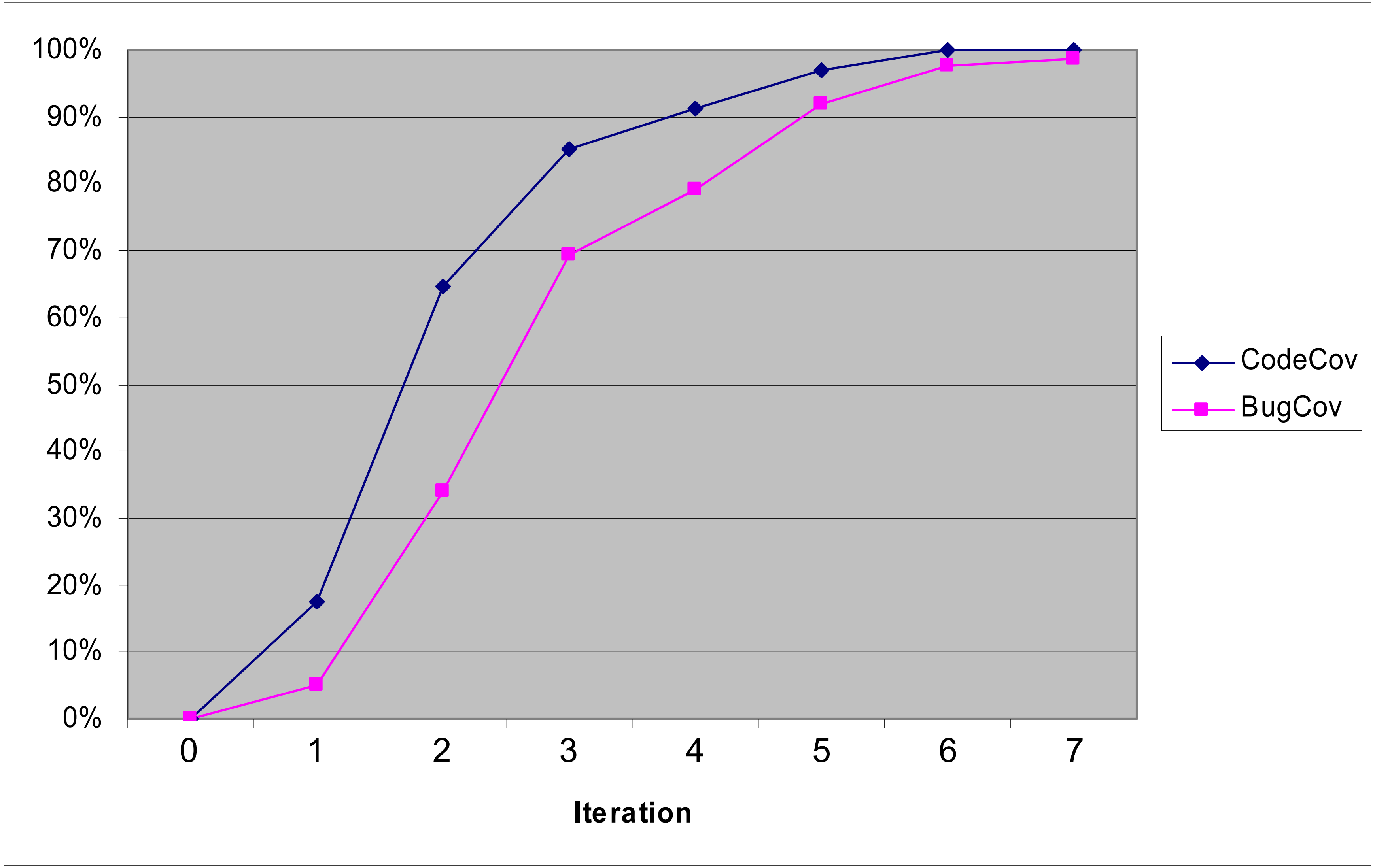
Evaluation

- Generate tests up to *N* (1..8) iterations
    - Concrete-State vs. Symstra
- Focus on the key methods (e.g., `add`, `remove`) of seven Java classes from various sources
    - most are complex data structures
- Measure #states, time, and code coverage
- Pentium IV 2.8 GHz, Java 2 JVM with 512 MB
- Experimental results show Symstra effectively
    - reduces the state space for exploration
    - reduces the time for achieving code coverage

Statistics of Some Programs

class	N	Concrete-State			Symstra		
		Time (sec)	#states	%cov	Time (sec)	#states	%cov
BinarySearchTree	6	23	731	100	29	197	100
	7	**Out of Memory**			137	626	100
	8				318	1458	100
BinomialHeap	6	51	3036	84	3	7	84
	7	**Out of Memory**			4	8	90
	8				9	9	91
LinkedList	6	412	9331	100	0.6	7	100
	7	**Out of Memory**			0.8	8	100
	8				1	9	100
TreeMap	6	12	185	83	8	28	83
	7	42	537	84	19	59	84
	8	**Out of Memory**			63	111	84

Code Coverage and (Seeded-)Bug Coverage with Iterations (Binary Search Tree)



Related Work

Directly construct new valid object states

- Generating tests with concrete-state construction

  e.g., TestEra [Marinov&Khurshid 01] and Korat [Boyapati et al. 02]

  - require specifications or **`repOK`** (class invariants)

- Generating tests with symbolic execution

  e.g. NASA Java Pathfinder [Khurshid et al. 03, Visser et al. 04]

  - require **`repOK`** (class invariants)

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?