Automated Software Testing in the Absence of Specifications

Tao Xie
North Carolina State University
Department of Computer Science

Nov 2005

http://www.csc.ncsu.edu/faculty/xie/

Why Automate Testing?

- Software testing is important
  - Software errors cost the U.S. economy about $59.5 billion each year (0.6% of the GDP) [NIST 02]
  - Improving testing infrastructure could save 1/3 cost
- Software testing is costly
  - Account for even half the total cost of software development [Beizer 90]
- Automated testing reduces manual testing effort
  - Test execution: JUnit framework
  - Test generation: Parasoft Jtest, Agitar Agitator, etc.
  - Test-behavior checking: Parasoft Jtest, Agitar Agitator, etc.

Automated Specification-Based Testing

- Test-input generation
  - preconditions
  - class invariants
- Test-behavior checking
  - postcondtions
  - class invariants
- Tool examples
  - Parasoft Jtest, TestEra [Marinov et al. 01], Korat [Boyapati et al. 02], AsmlT [Grieskamp et al. 02], ASTOOT [Doong et al. 94], JML [Cheon et al. 02], etc.

Specs often don’t exist in practice

Approaches

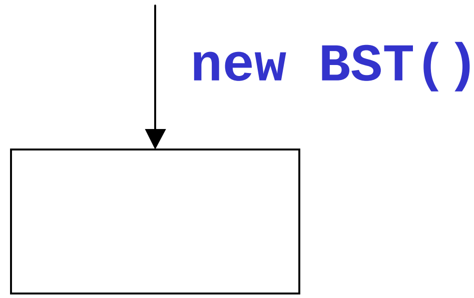
- Test-Input Generation
  - Method-sequence exploration
  - Concrete-state exploration [ASE 04]
  - Symbolic-state exploration [TACAS 05]
- Test-Behavior Checking
  - Test selection based on new behavior [ASE 03]
  - Test selection based on special behavior [ISSRE 05]
  - Test abstraction for overall behavior [ICFEM 04]

Binary Search Tree Example

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

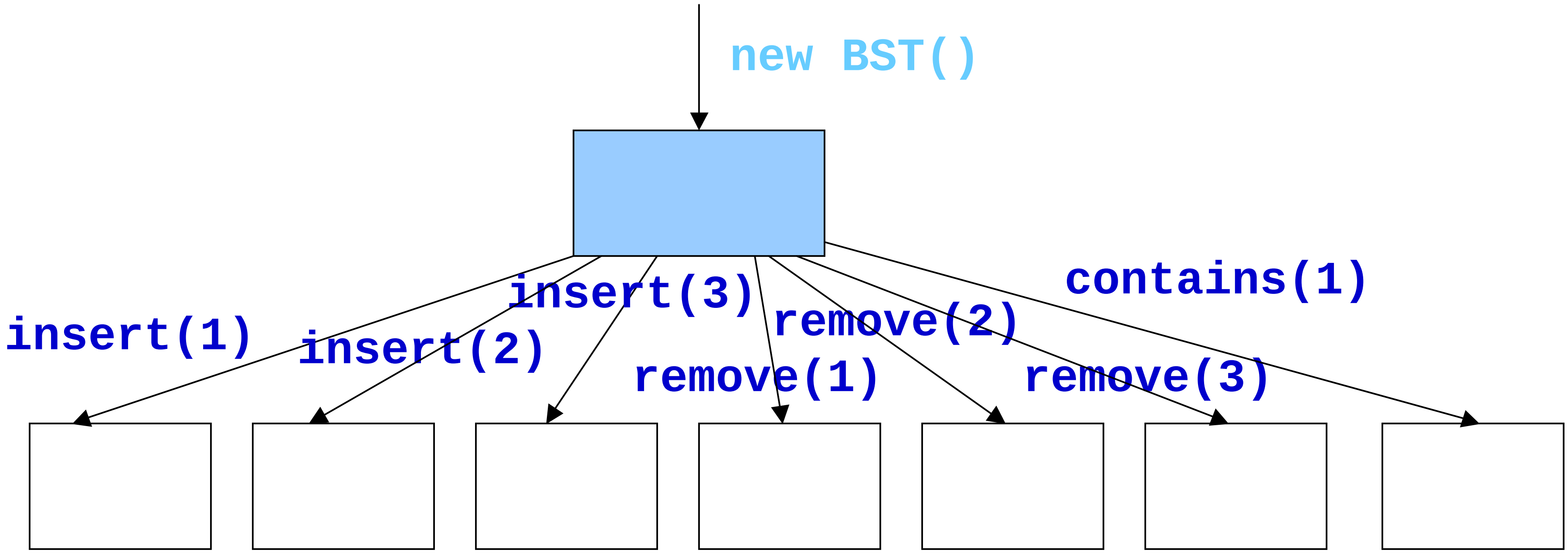
Exploring Method Sequences

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



Exploring Method Sequences

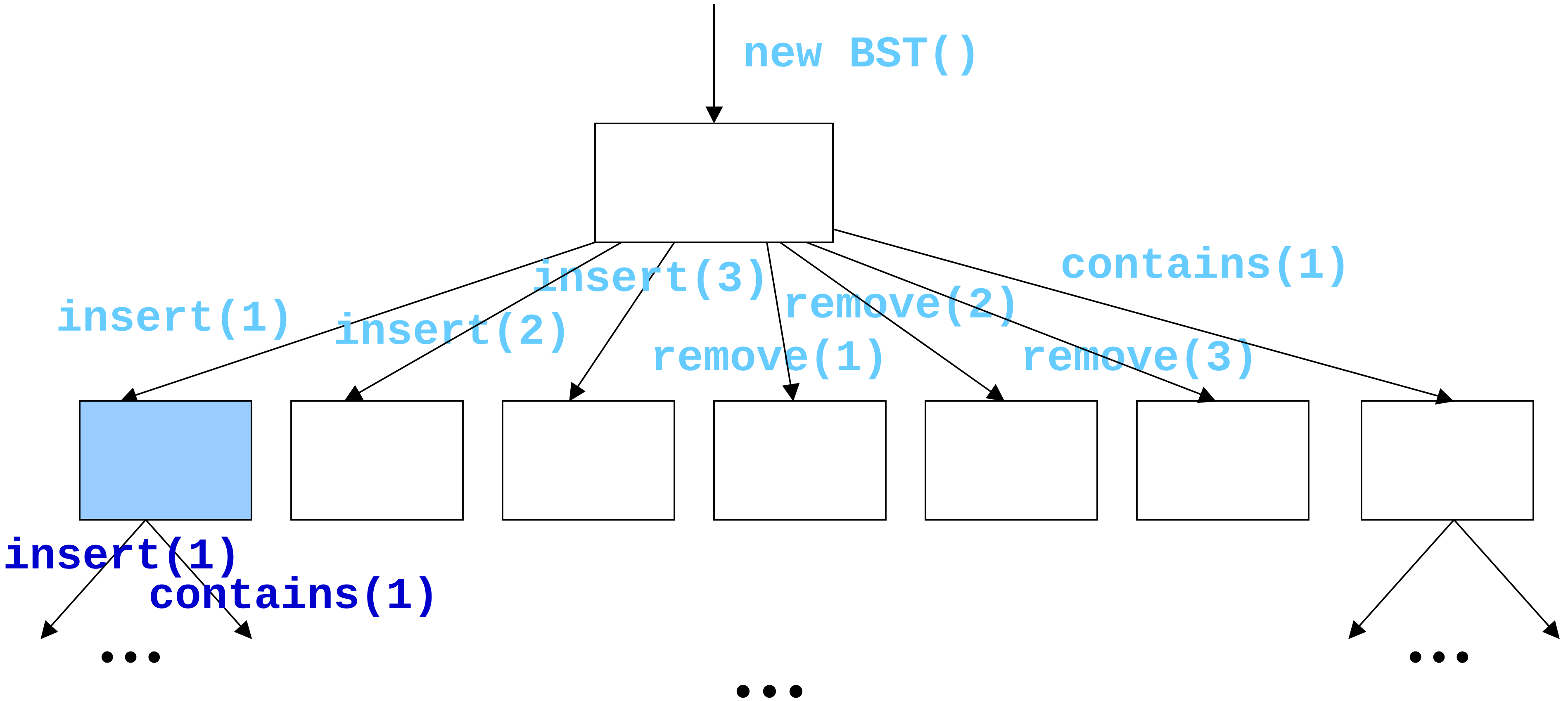
- Method arguments: `insert(1)`, `insert(2)`, `insert(3)`, `remove(1)`, `remove(2)`, `remove(3)`, `contains(1)`



Iteration 1

Exploring Method Sequences

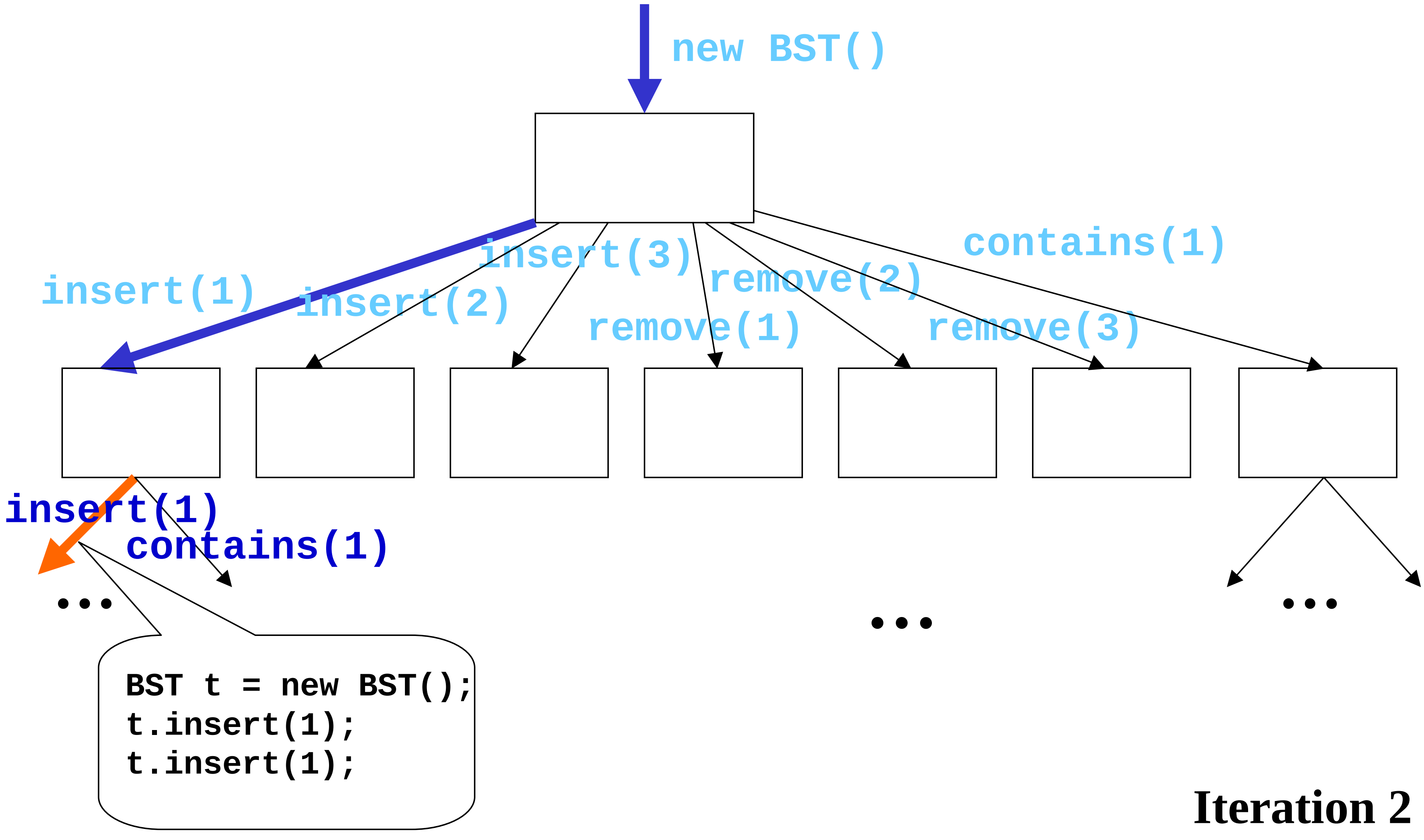
- Method arguments: **`insert(1), insert(2), insert(3), remove(1), remove(2), remove(3), contains(1)`**



Iteration 2

Generating Tests from Exploration

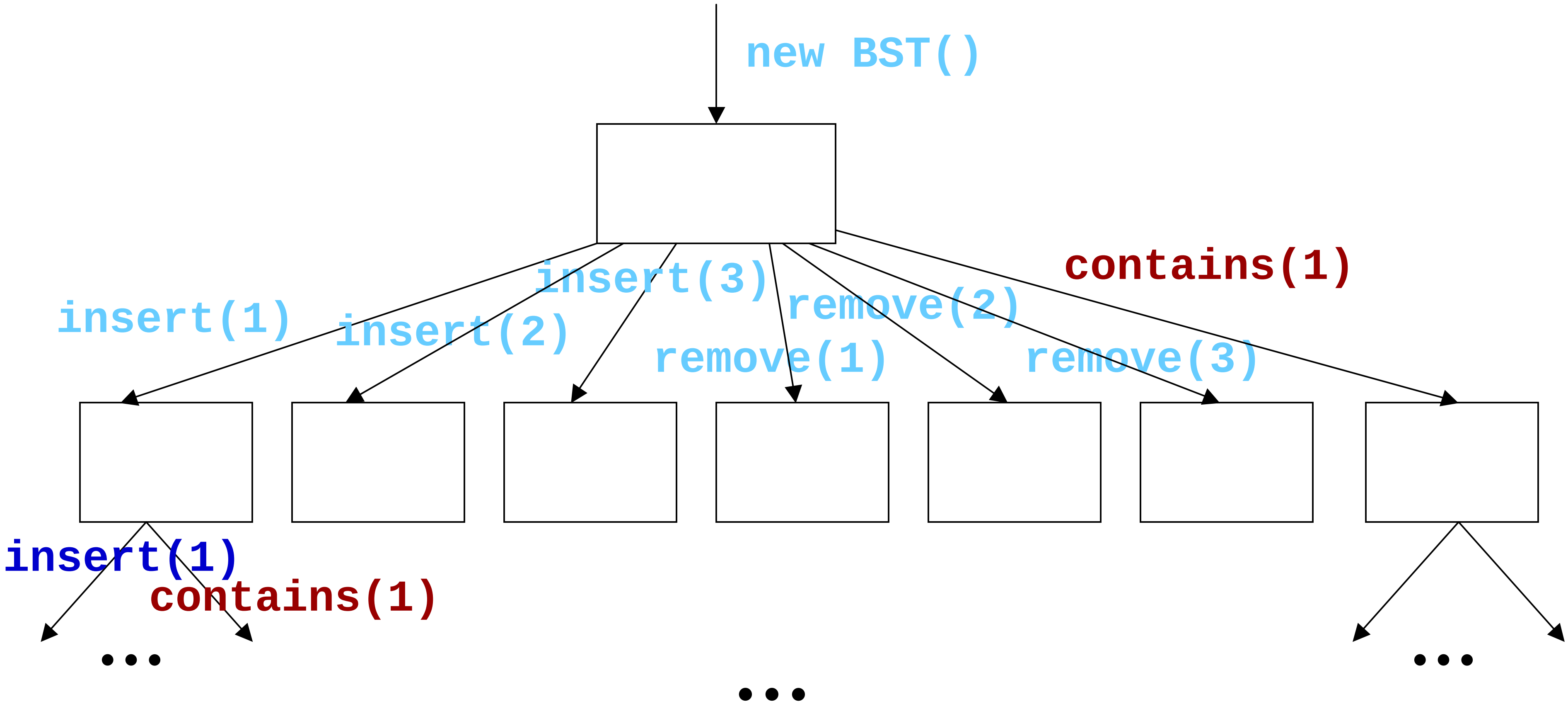
- Collect method sequence along the shortest path (constructor-call edge à each method-call edge)



Iteration 2

Pruning State-Preserving Methods

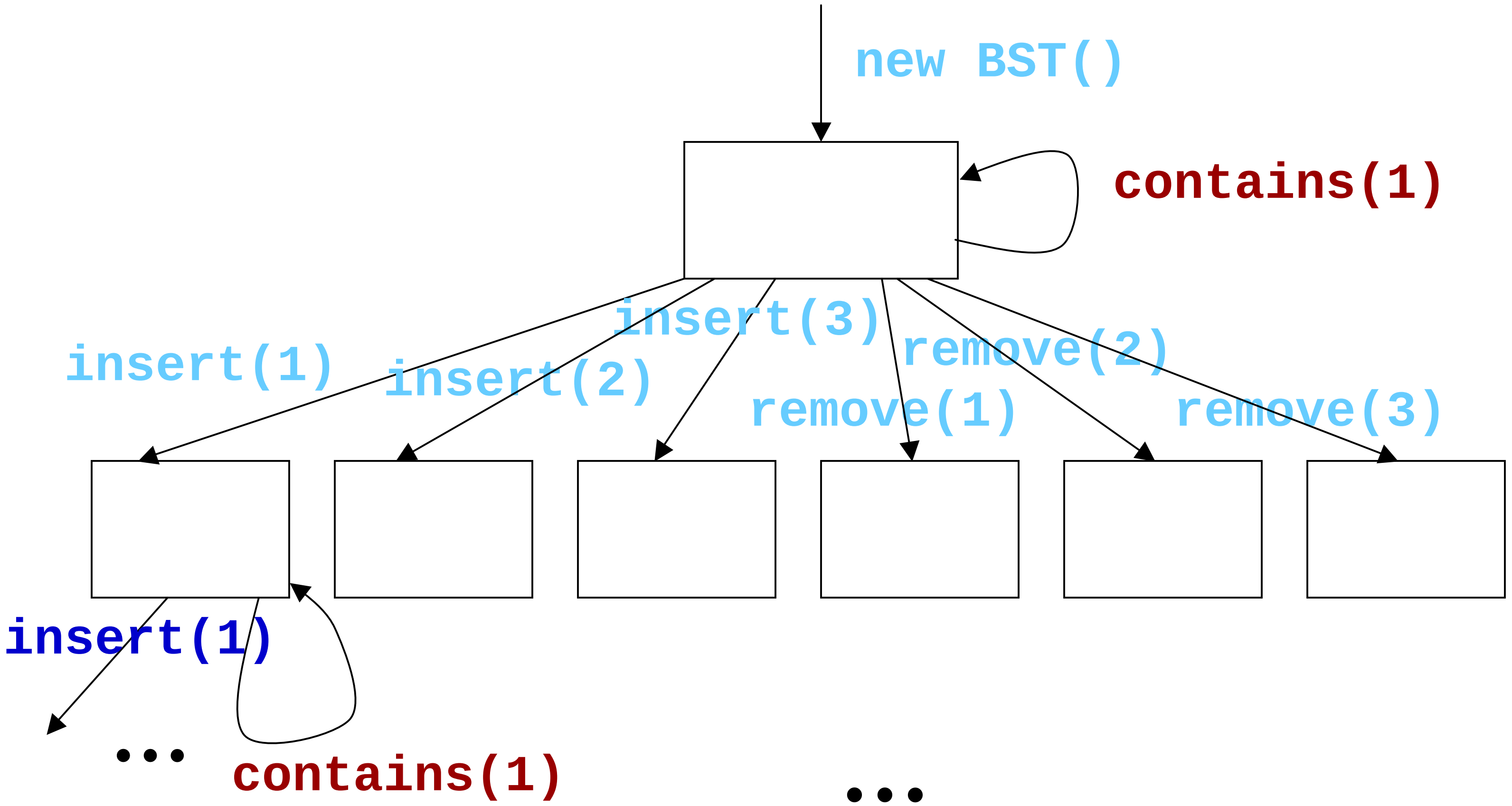
- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3), contains(1)`



Iteration 2

Pruning State-Preserving Methods

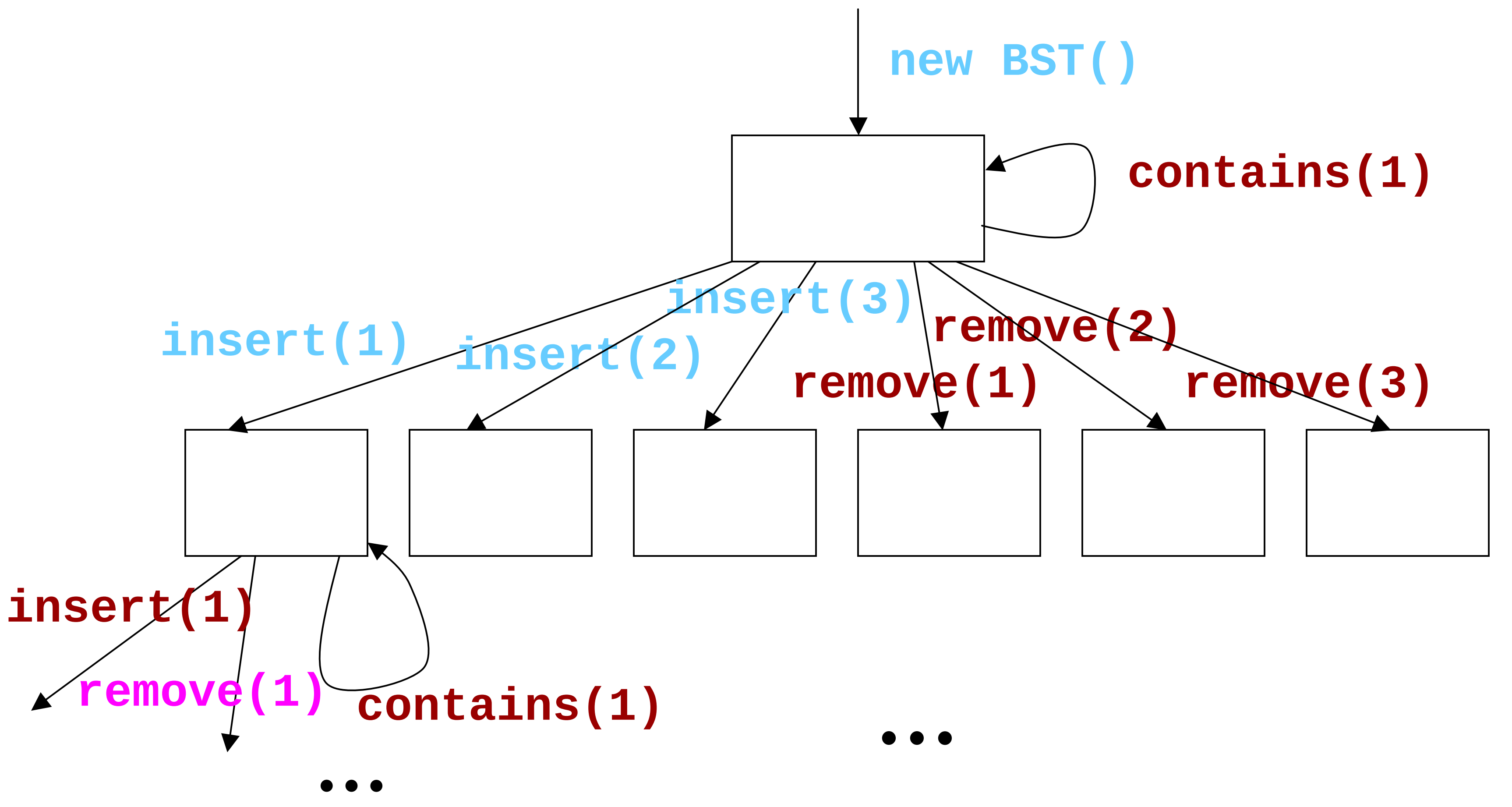
- Method arguments: **`insert(1), insert(2), insert(3), remove(1), remove(2), remove(3), contains(1)`**



Iteration 2

Observation

- Some method sequences lead receiver object states back to earlier explored states



Iteration 2

Rationale

- Focus on each method execution individually
- When method executions are deterministic, unnecessary to test a method with the same inputs (same inputs $\Rightarrow$ same behavior)
  - method inputs: incoming program states
    - receiver-object state: transitively-reachable-field values
    - arguments

Binary Search Tree Example

```
public class BST implements Set {
  //@ invariant       // class invariant for BST
  //@ repOk();
   Node root;
   int size;
   public void insert (int value) { … }
   public void remove (int value) { … }
   public bool contains (int value) { … }
}
```

- If receiver-object states are directly constructed, we need to have a way to know valid object states
  - defined by a Java predicate: **`repOK`**

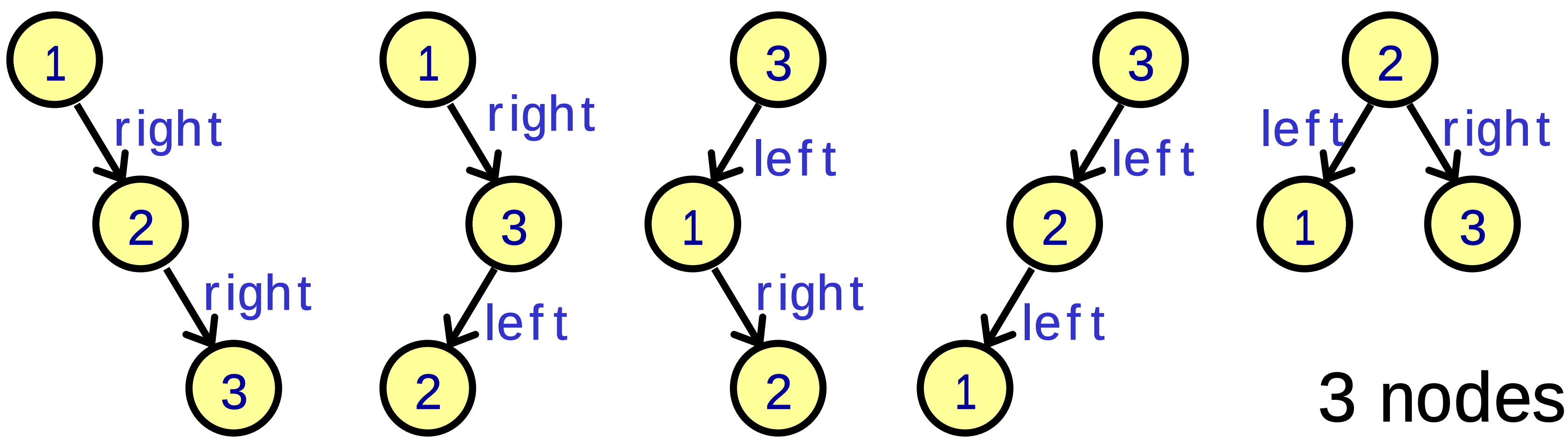
repOk (Class Invariant)

```
boolean repOk() {
  if (root == null) return size == 0;  // empty tree has size 0
  Set visited = new HashSet(); visited.add(root);
  List workList = new LinkedList(); workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node)workList.removeFirst();
    if (current.left != null) {
      if (!visited.add(current.left)) return false; // acyclicity
      workList.add(current.left);
    }
    if (current.right != null) {
      if (!visited.add(current.right)) return false; // acyclicity
      workList.add(current.right);
    }
  }
  if (visited.size() != size) return false; // consistency of size
  if (!isOrdered(root)) return false;       // data is ordered
  return true;
}
```

Korat

[Boyapati et al. ISSTA 02]

- Given predicate *p* and finitization *f*, generate all inputs for which *p* returns "*true*"
  - uses finitization to define input space
    - e.g., defines #nodes and what values can be on a BST node.
  - systematically explores valid input space
    - prunes input space using field accesses



3 nodes

Node vals: {1, 2, 3}

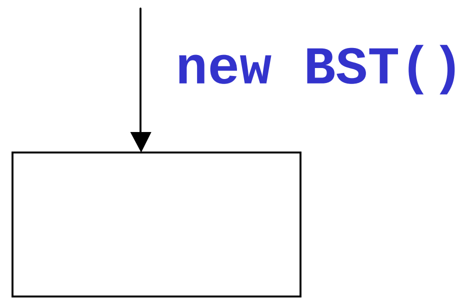
What if `repOK` is not there

- Then direct construction of valid object states seem impossible

- Solution: fall back to building valid object states with method sequences but in a smarter way
  - method-sequence exploration
    - assume a state-modifying method leads to a new object state
  - explicit-state exploration
    - inspect whether an object state is actually new (defined by transitively-reachable-field values)

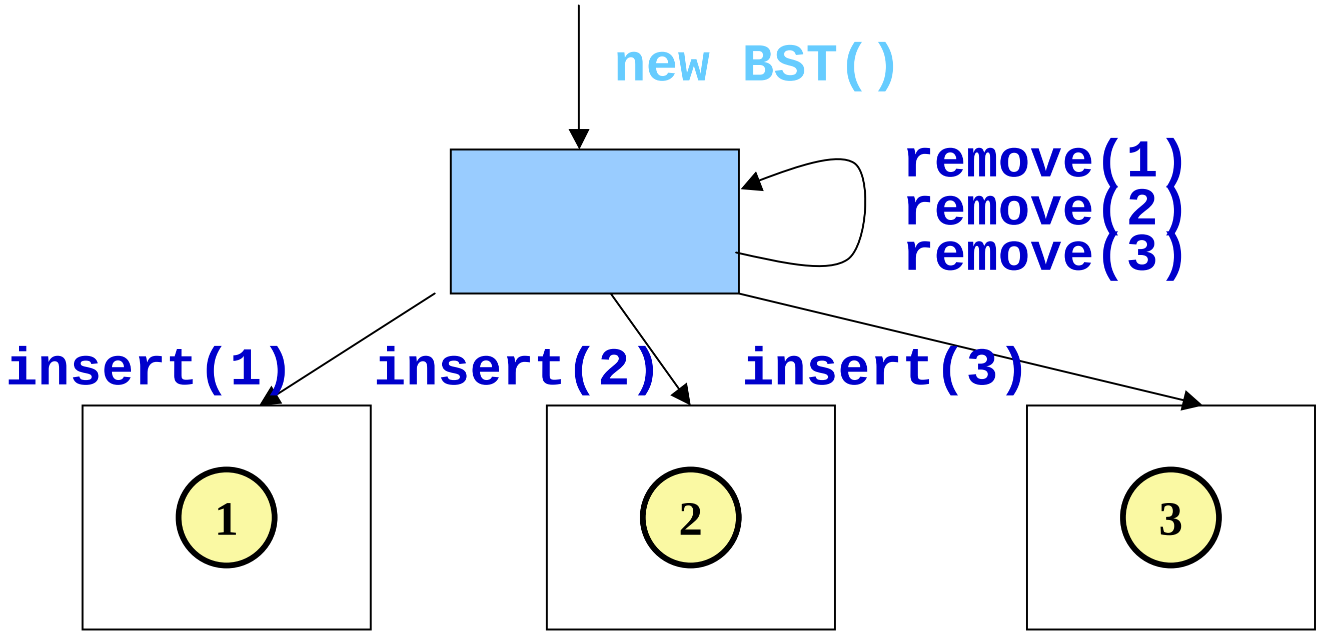
Exploring Concrete States

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



Exploring Concrete States

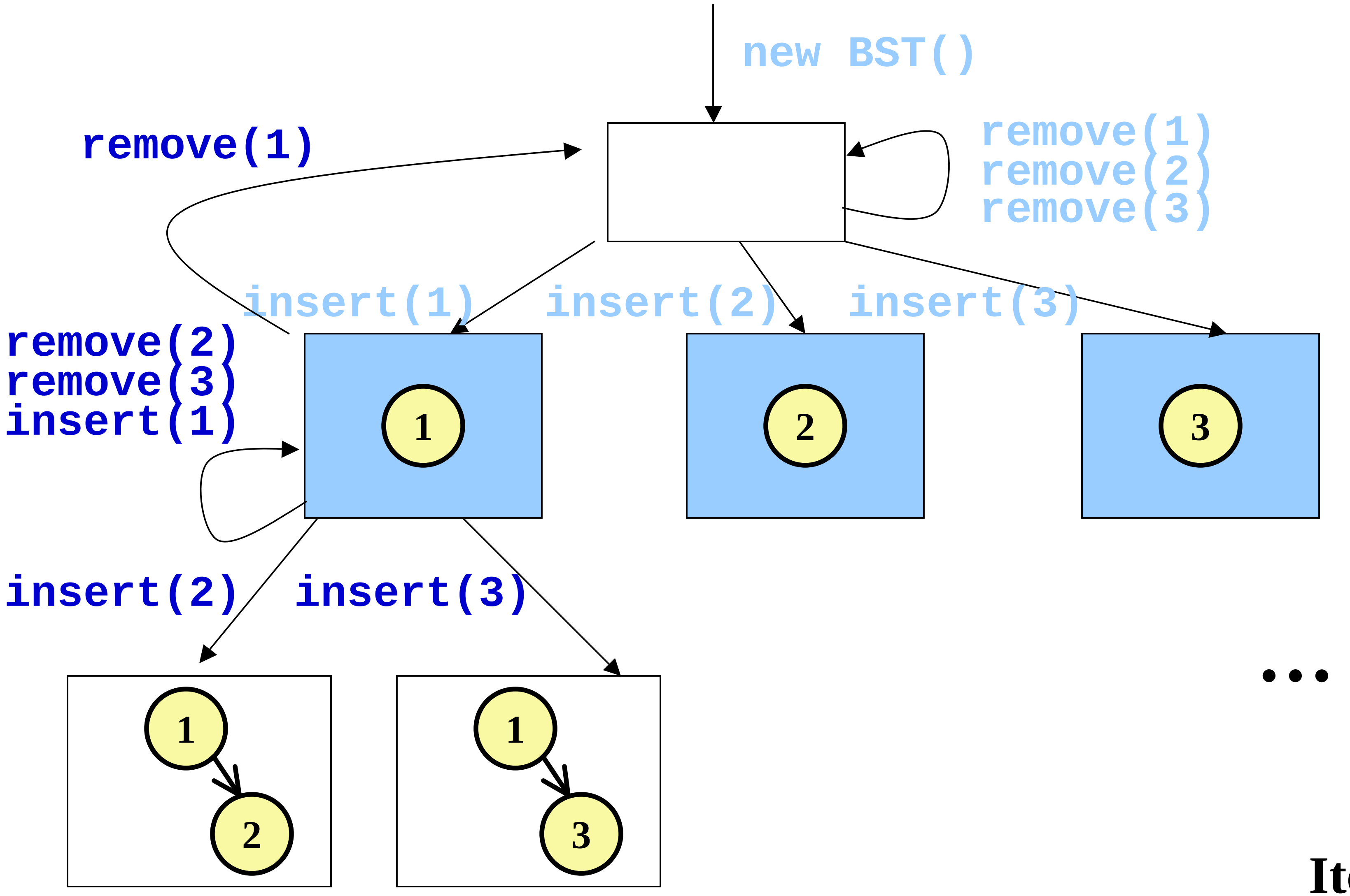
- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`



Iteration 1

Exploring Concrete States

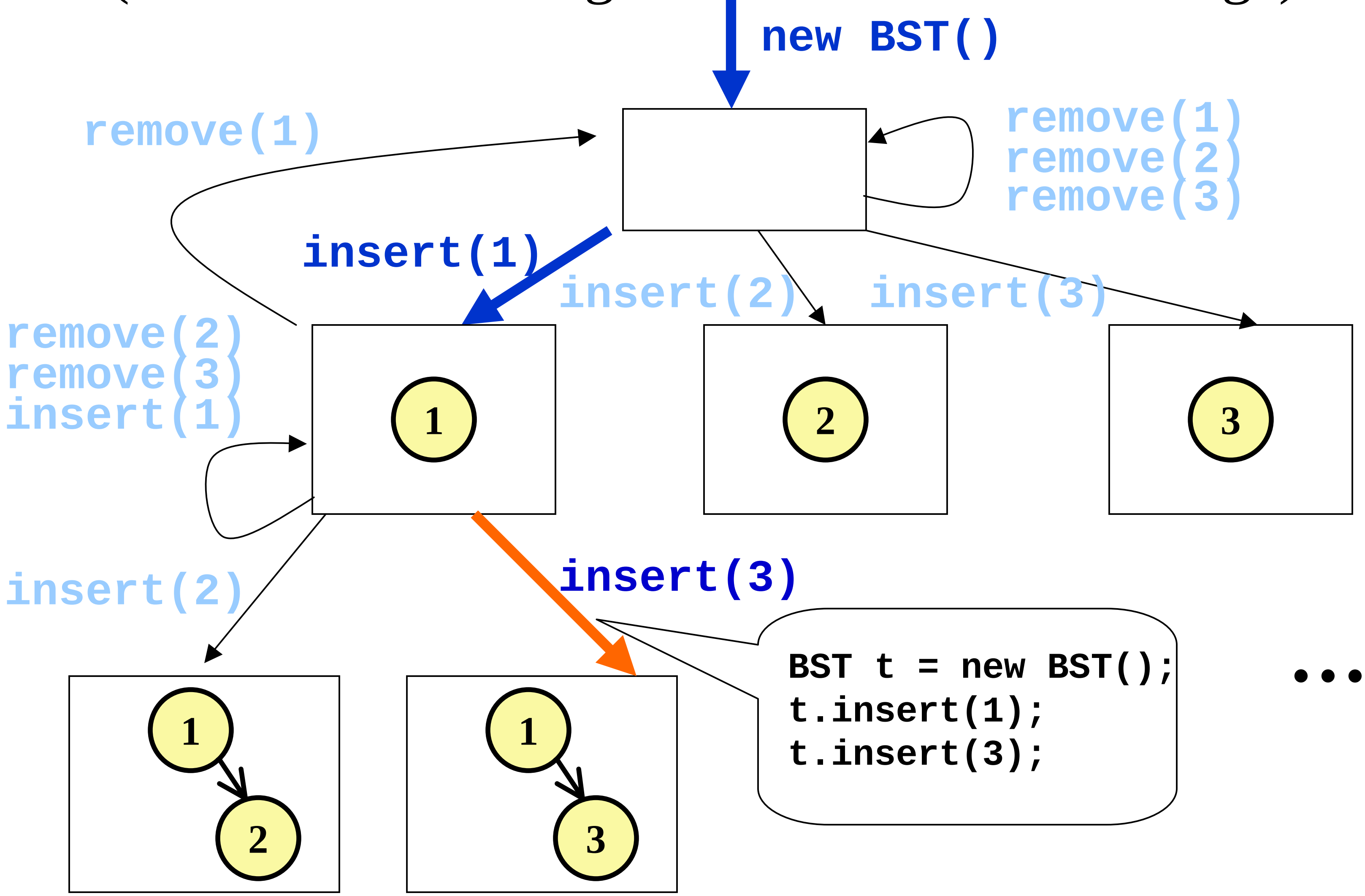
- Method arguments: **`insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`**



Iteration 2

Generating Tests from Exploration

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



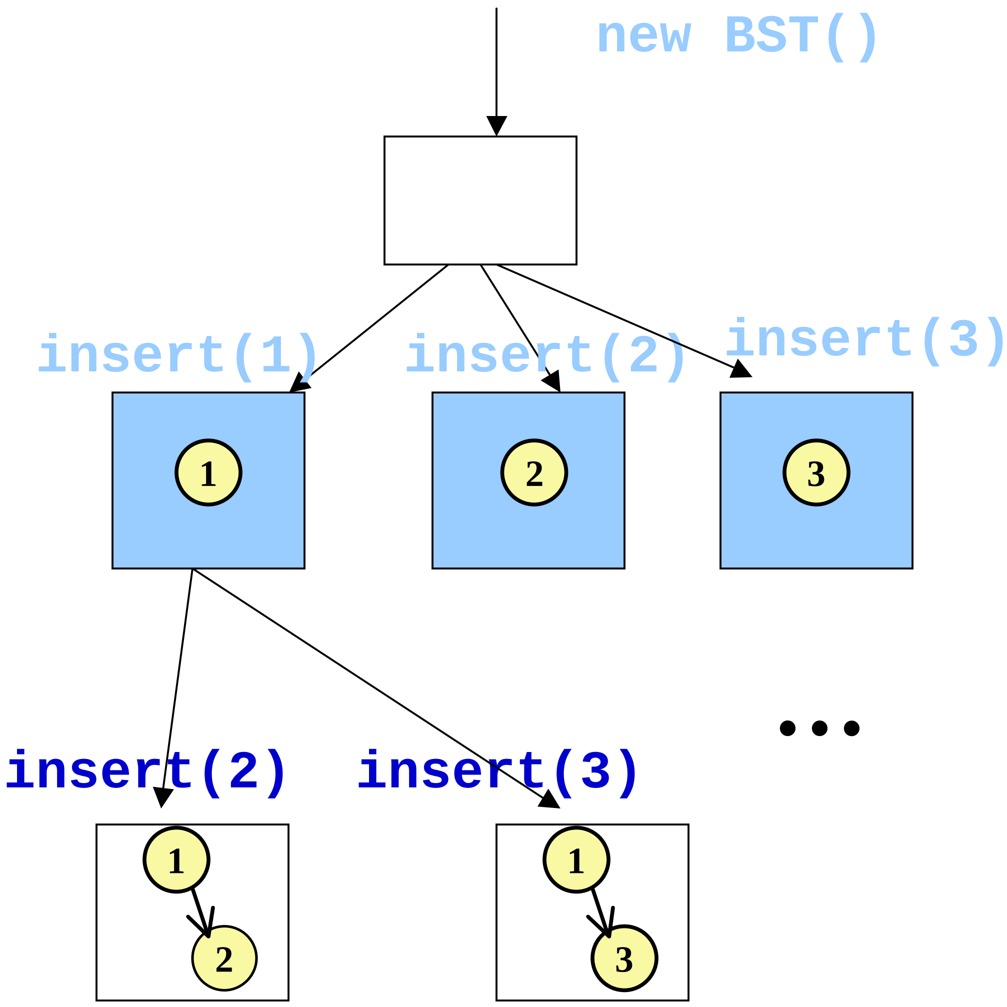
Improvement over Method-Sequence Exploration

- An industrial tool adopting previous approach based on method sequences
  - Parasoft Jtest 4.5 www.parasoft.com
    - Generate tests with method-call lengths up to three
- Use Jtest to generate tests for 11 Java classes from various sources
  - most are complex data structures
- Apply Rostra on the Jtest-generated tests
  - 90% of generated tests are redundant, i.e., 90% tests contain no new method inputs

Issues of Concrete-State Exploration

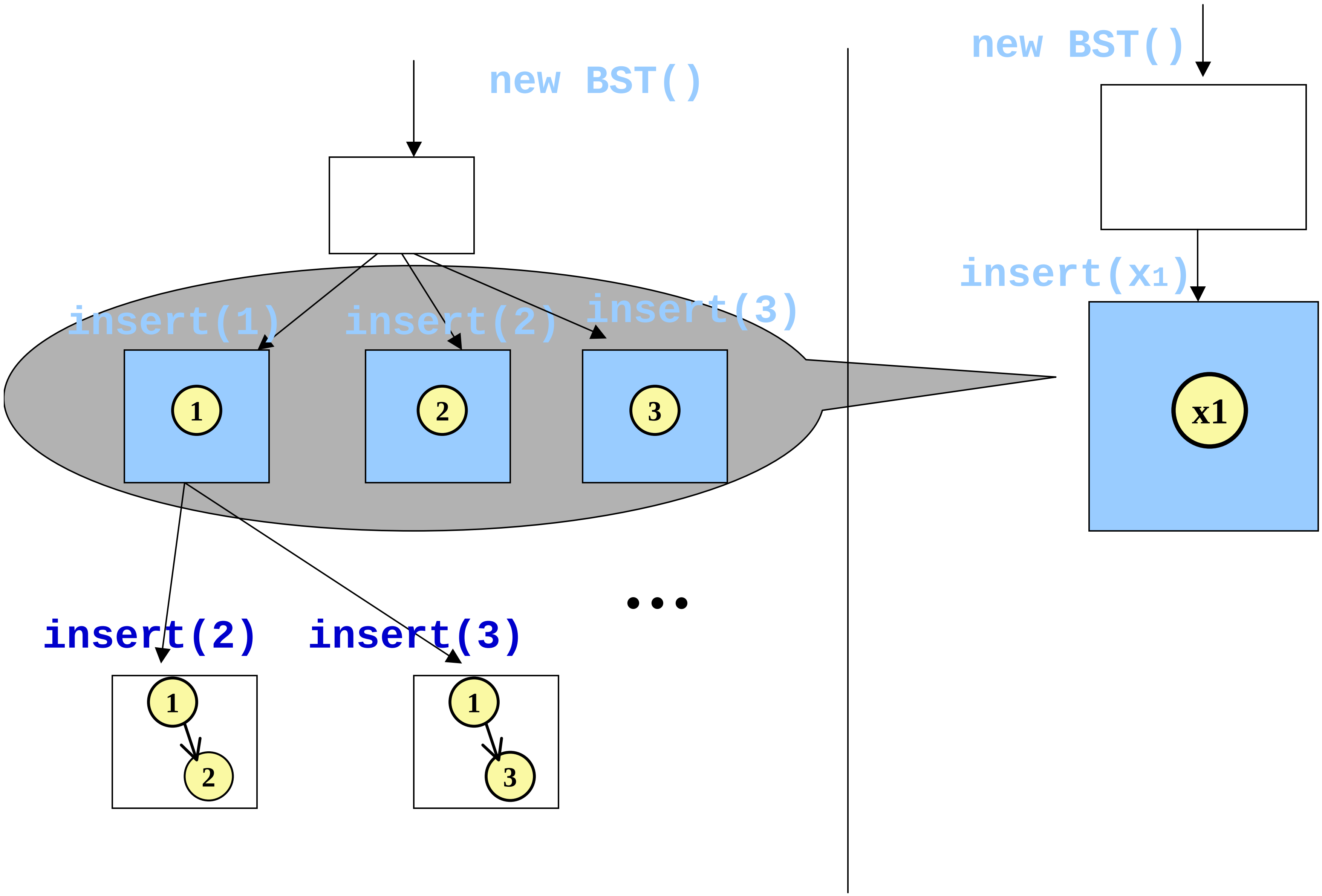
- State explosion
  - need at least *N* different **`insert`** arguments to reach a BST with size *N*
  - run out of memory when *N* reaches 7
- Relevant-argument determination
  - assume a set of given relevant arguments
    - e.g., **`insert(1), insert(2), insert(3), etc.`**

Exploring Concrete States



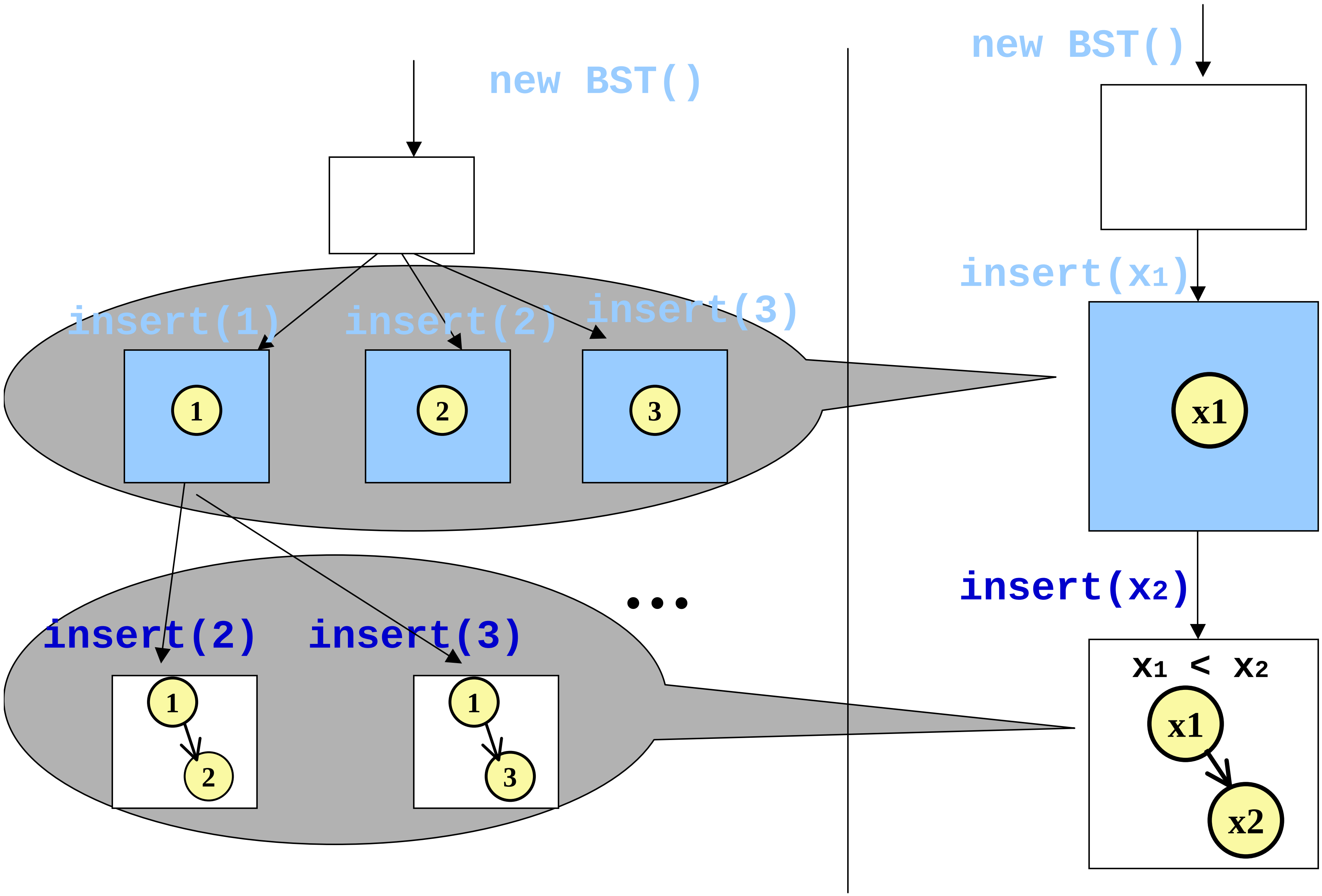
Iteration 2

State Abstraction: Symbolic States



Iteration 2

State Abstraction: Symbolic States

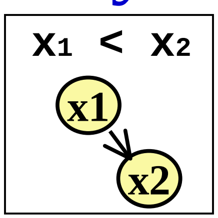


Iteration 2

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)
                   ...;
```

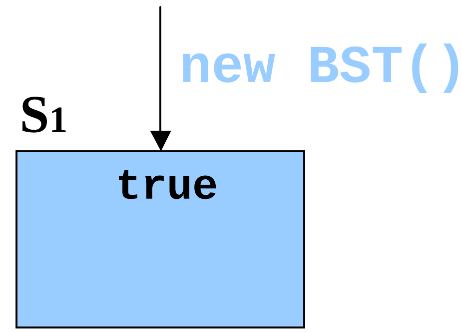
- Build a path condition for each path
  - conjunct conditionals or their negations
- Produce symbolic states (<heap, path condition>)
  - e.g., 

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++;
}
```

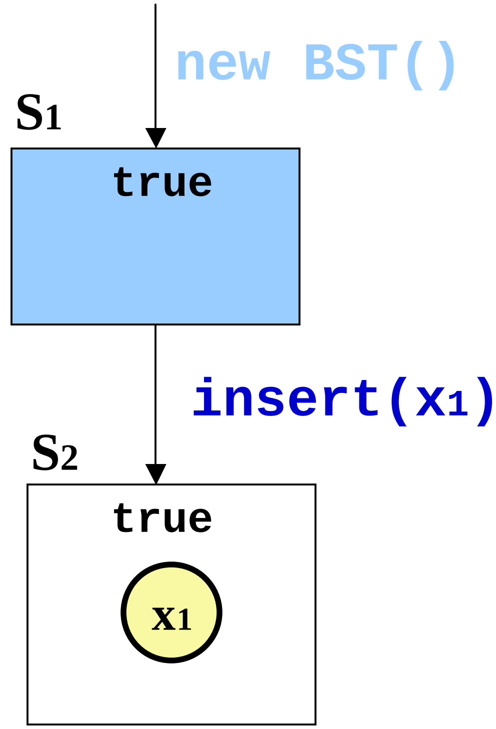
Exploring Symbolic States

```
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

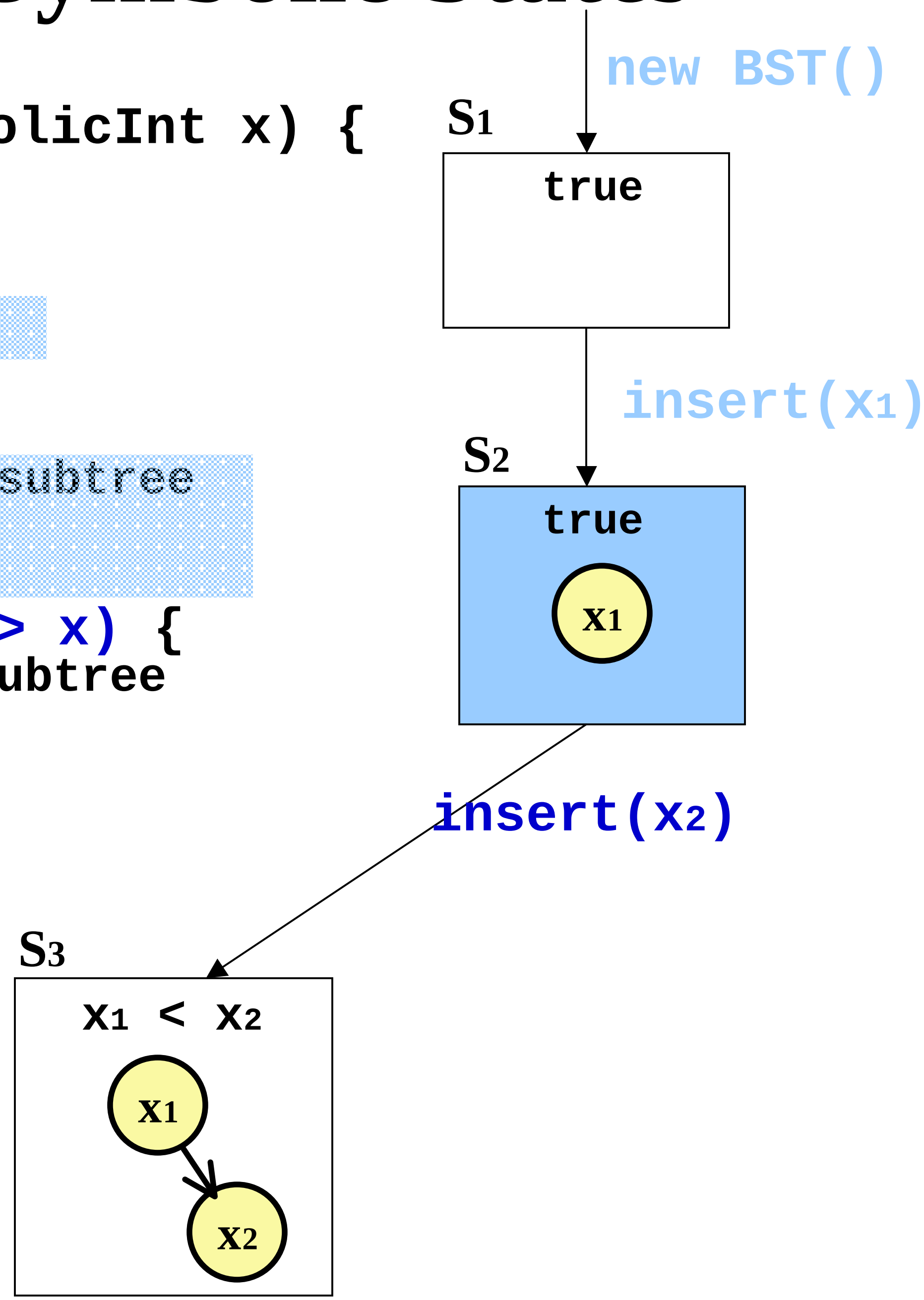
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

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

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



Iteration 1

Exploring Symbolic States

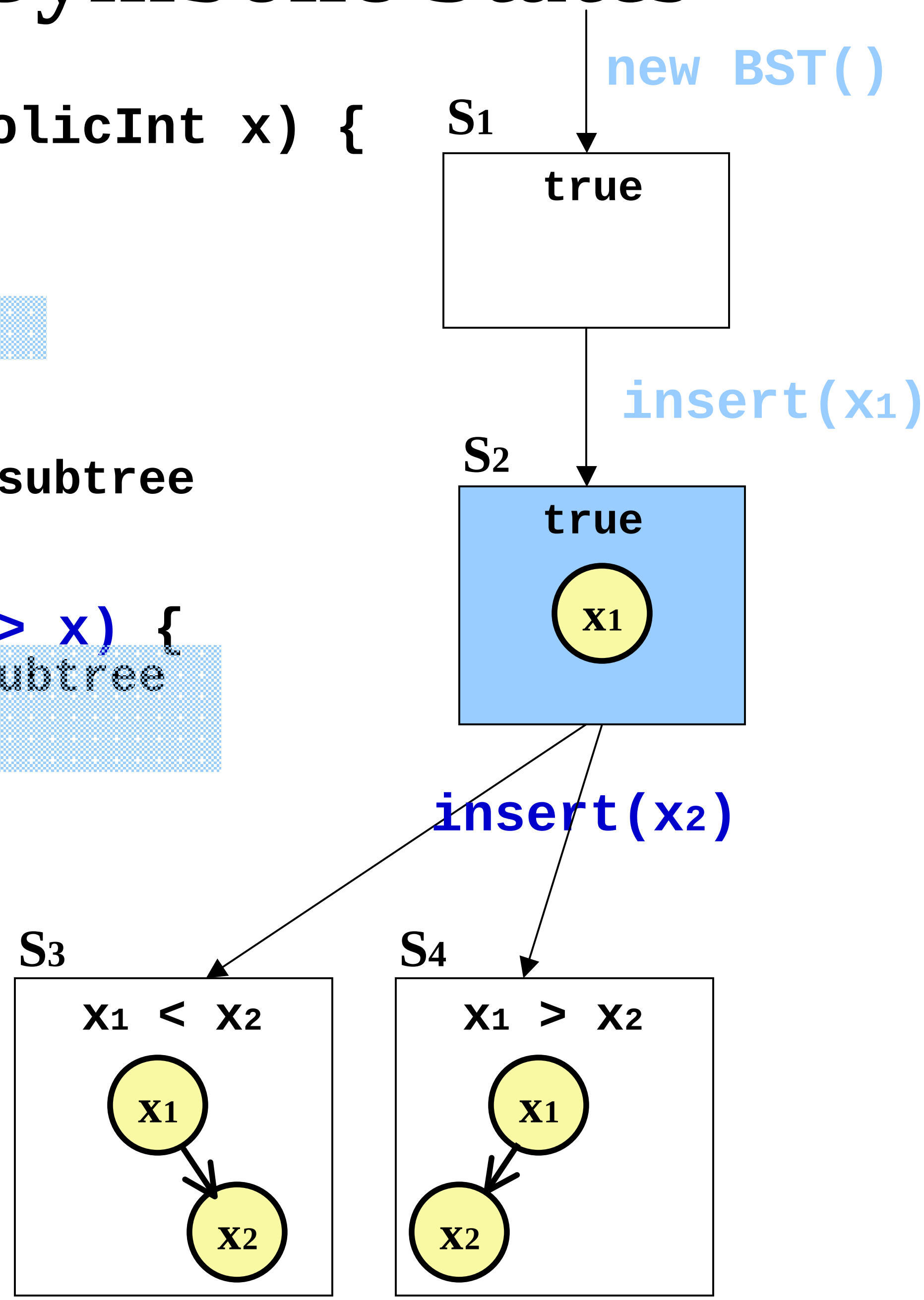
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
     //explore the right subtree
      ...

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

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



Iteration 2

Exploring Symbolic States

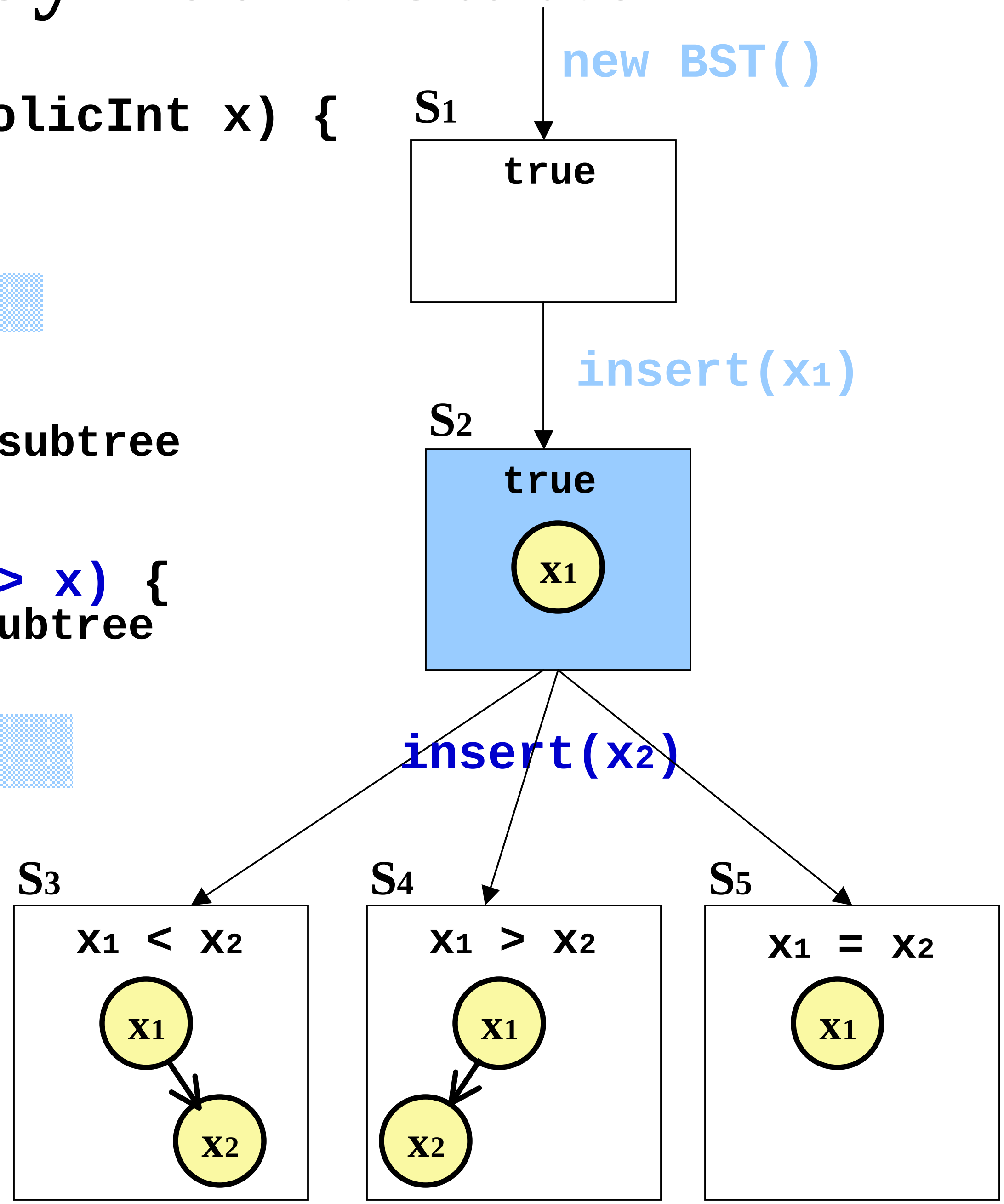
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

    } else if (t.value > x) {
      //explore the left subtree
       ...
    } else return;
   }
 }
 size++;
}
```



Iteration 2

Exploring Symbolic States

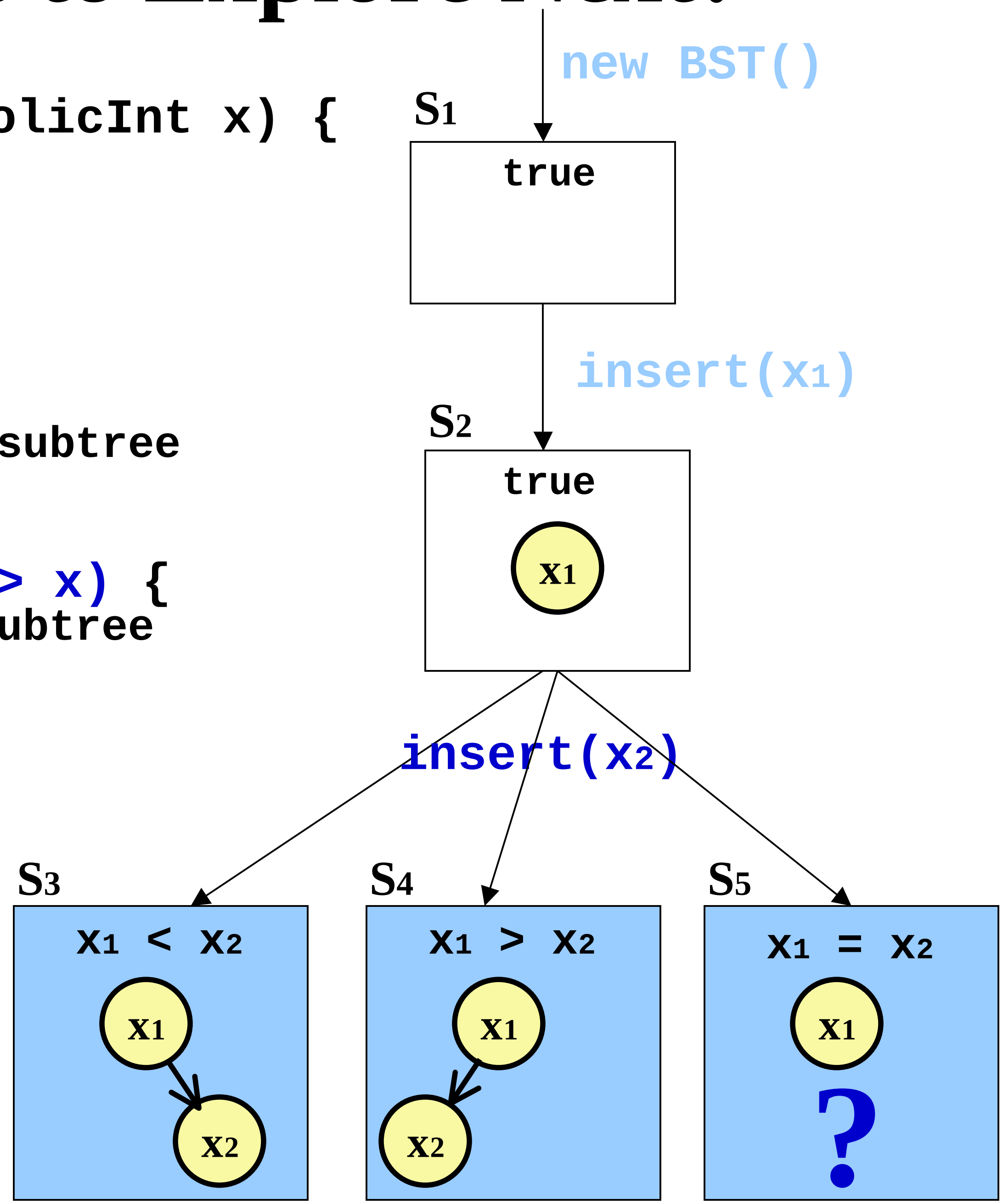
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

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

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



Iteration 2

Which States to Explore Next?

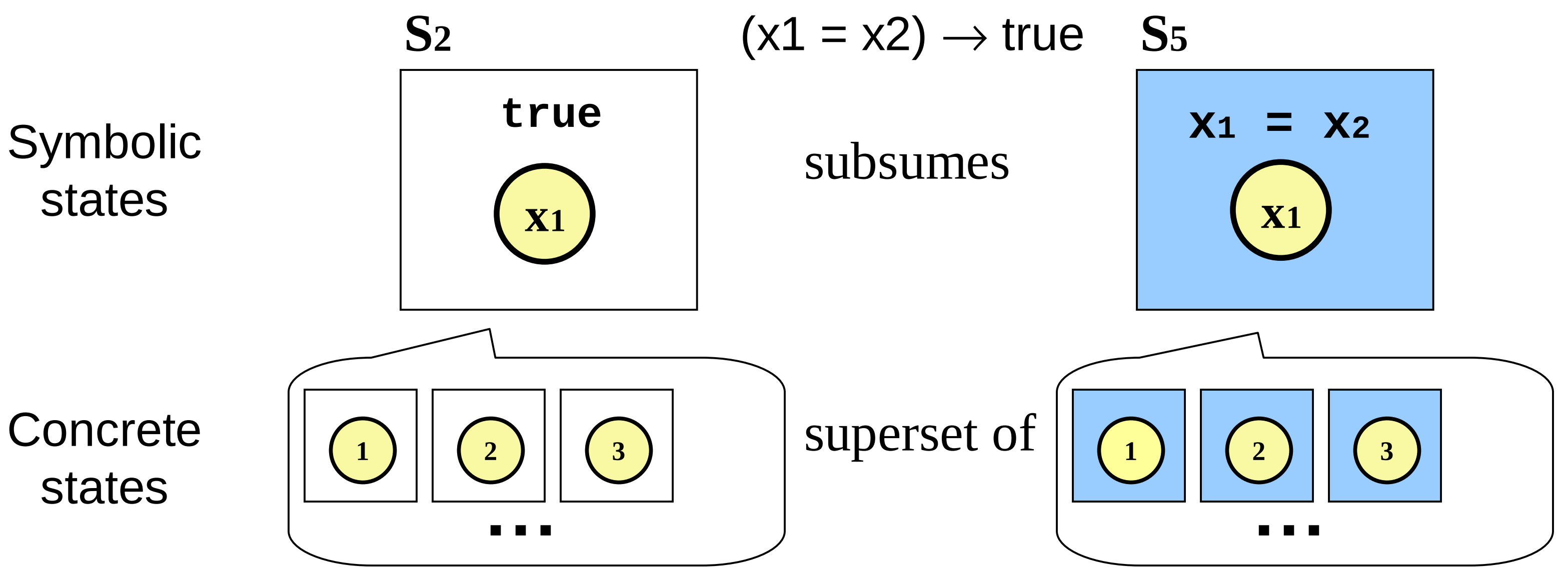
```
public void insert(SymbolicInt x) {
 if (root == null) {
   root = new Node(x);
 } else {
   Node t = root;
   while (true) {
    if (t.value < x) {
      //explore the right subtree
       ...

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

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



Iteration 3

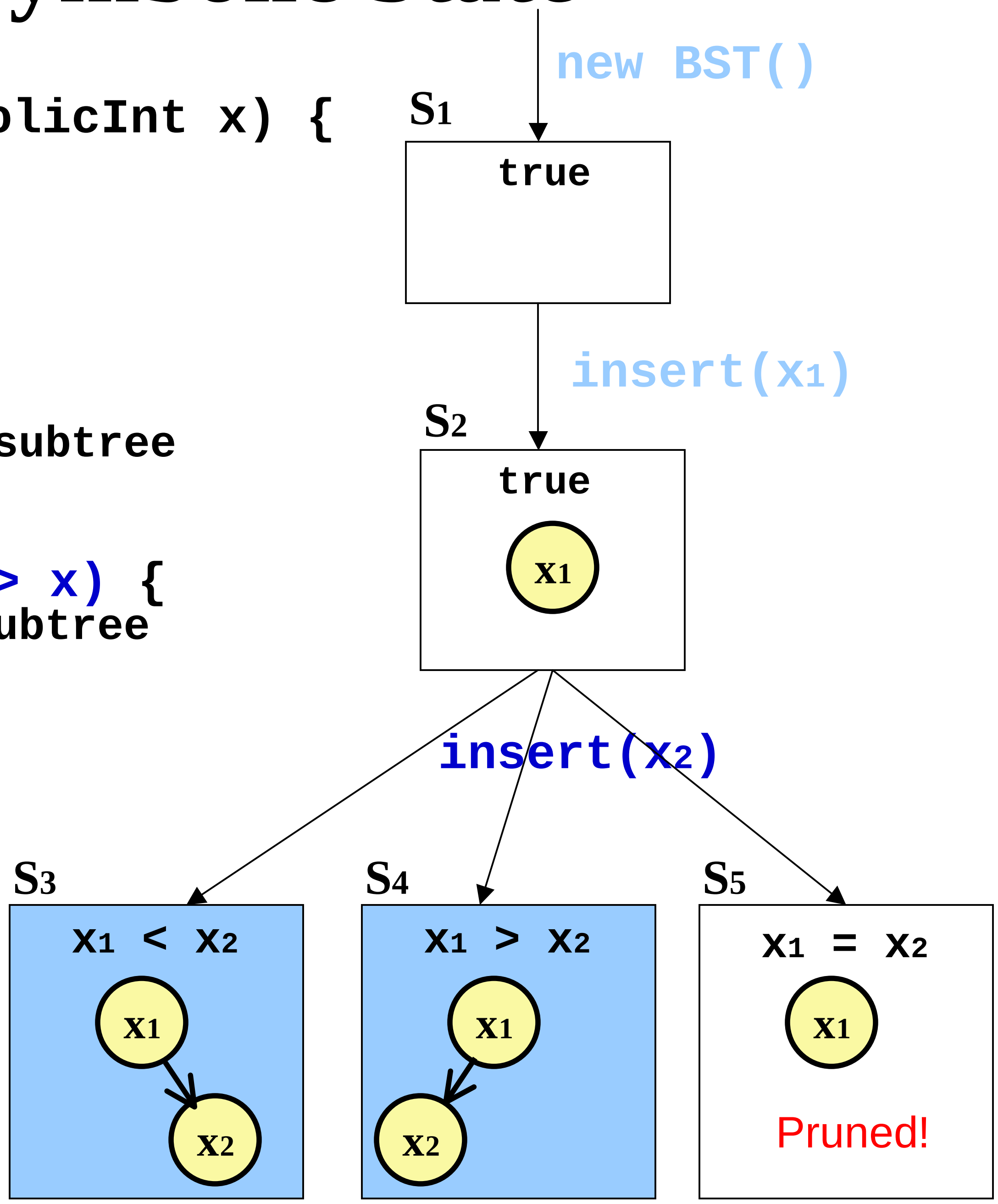
Symbolic State Subsumption

- Symbolic state S_2:<H_2, c_2> subsumes S_5:<H_5, c_5>
  - Heaps $H2$ and $H5$ are isomorphic
  - Path condition $c5 \rightarrow c2$ [checked using CVC Lite, Omega]
- 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++;
}
```



Iteration 3

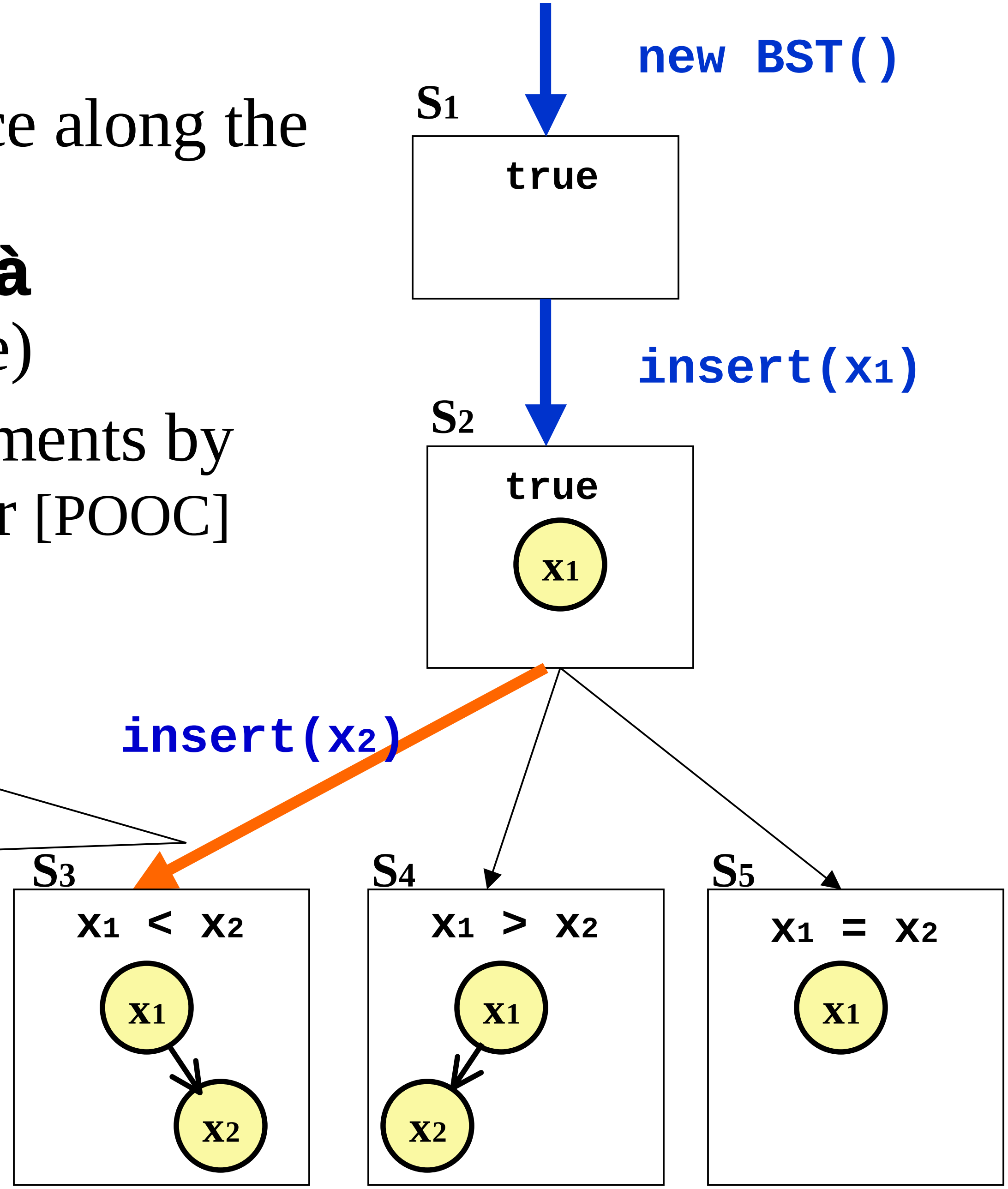
Generating Tests from Exploration

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

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

$x_1 < x_2$

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



Improvement over Concrete-State Exploration

- Focus on the key methods (e.g., `add`, `remove`)
- Generate tests up to 8 iterations
  - Concrete-State vs. Symstra
- Measure #states, time, and branch coverage

- Experimental results show Symstra effectively
  - reduces the state space for exploration
  - reduces the time for achieving branch coverage

Statistics of Some Programs

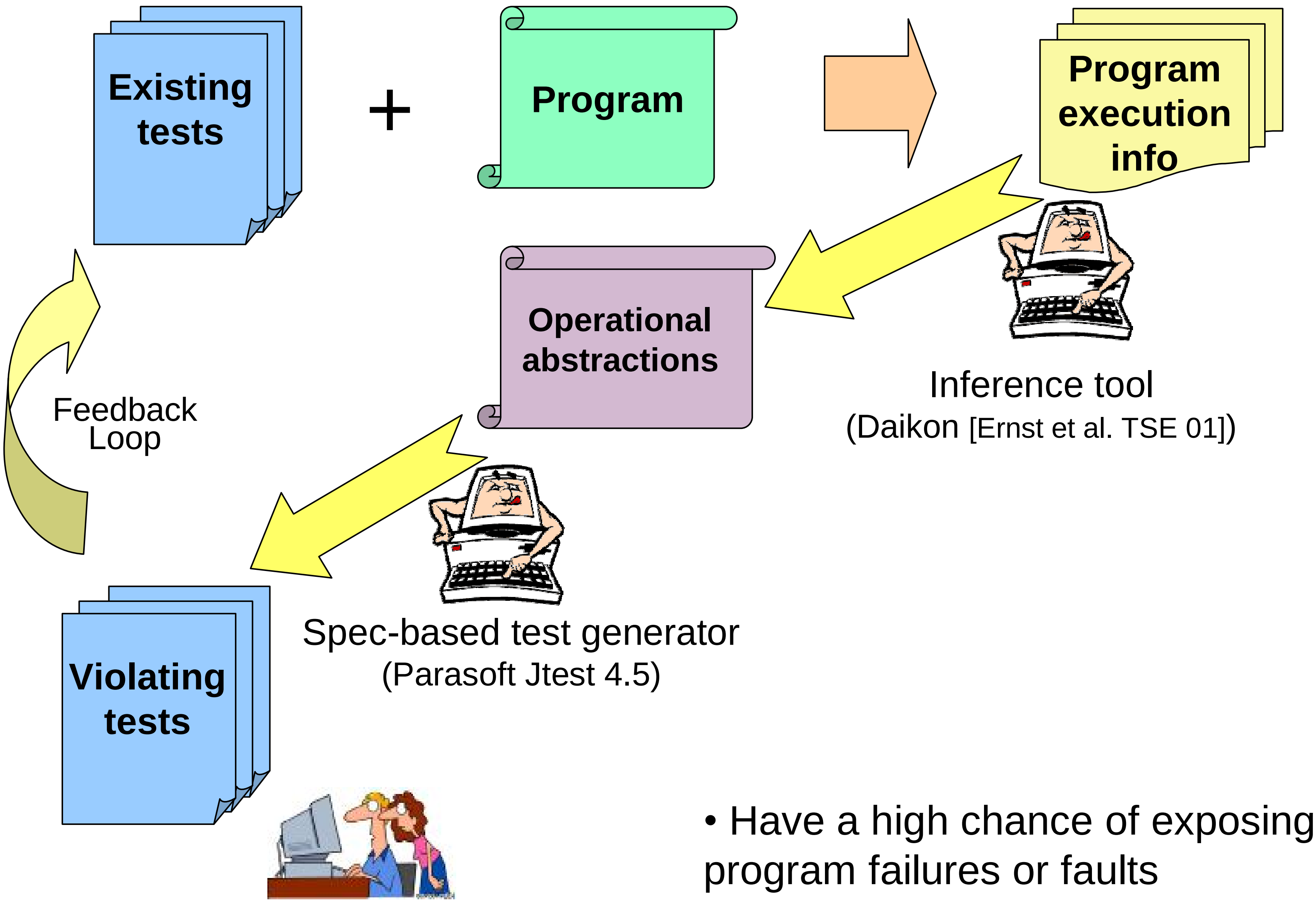
class	N	Concrete-State			Symstra		
		Time (sec)	#states	%cov (branch)	Time (sec)	#states	%cov (branch)
BinarySearchTree	6	23	731	100	29	197	100
	7	Out of Memory			137	626	100
	8				318	1458	100
BinomialHeap	6	51	3036	84	3	7	84
	7	Out of Memory			4	8	90
	8				9	9	91
LinkedList	6	412	9331	100	0.6	7	100
	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

Approaches

- Test-Input Generation
  - Method-sequence exploration
  - Concrete-state exploration [ASE 04]
  - Symbolic-state exploration [TACAS 05]
  - Concolic-state exploration
    DART [Godefroid et al. PLDI 05], EGT [Cadar&Engler SPIN 05], CUTE [Sen et al. FSE 05]

- Test-Behavior Checking
  - Test selection based on new behavior [ASE 03]
  - Test selection based on special behavior [ISSRE 05]
  - Test abstraction for overall behavior [ICFEM 04]

Test Selection based on New Behavior



• Have a high chance of exposing program failures or faults

Example

```
class Stack {
   ...
 public int top(){
    if (numberOfElements < 1) {
      System.out.println("Empty Stack");
      return -1;
    } else {
      return elems[numberOfElements-1];
    }
 }
```

Daikon generates from manually-created tests :

```
@post: [($result == -1) ó (this.numberOfElements == 0)]
```

Jtest generates a violating test input:

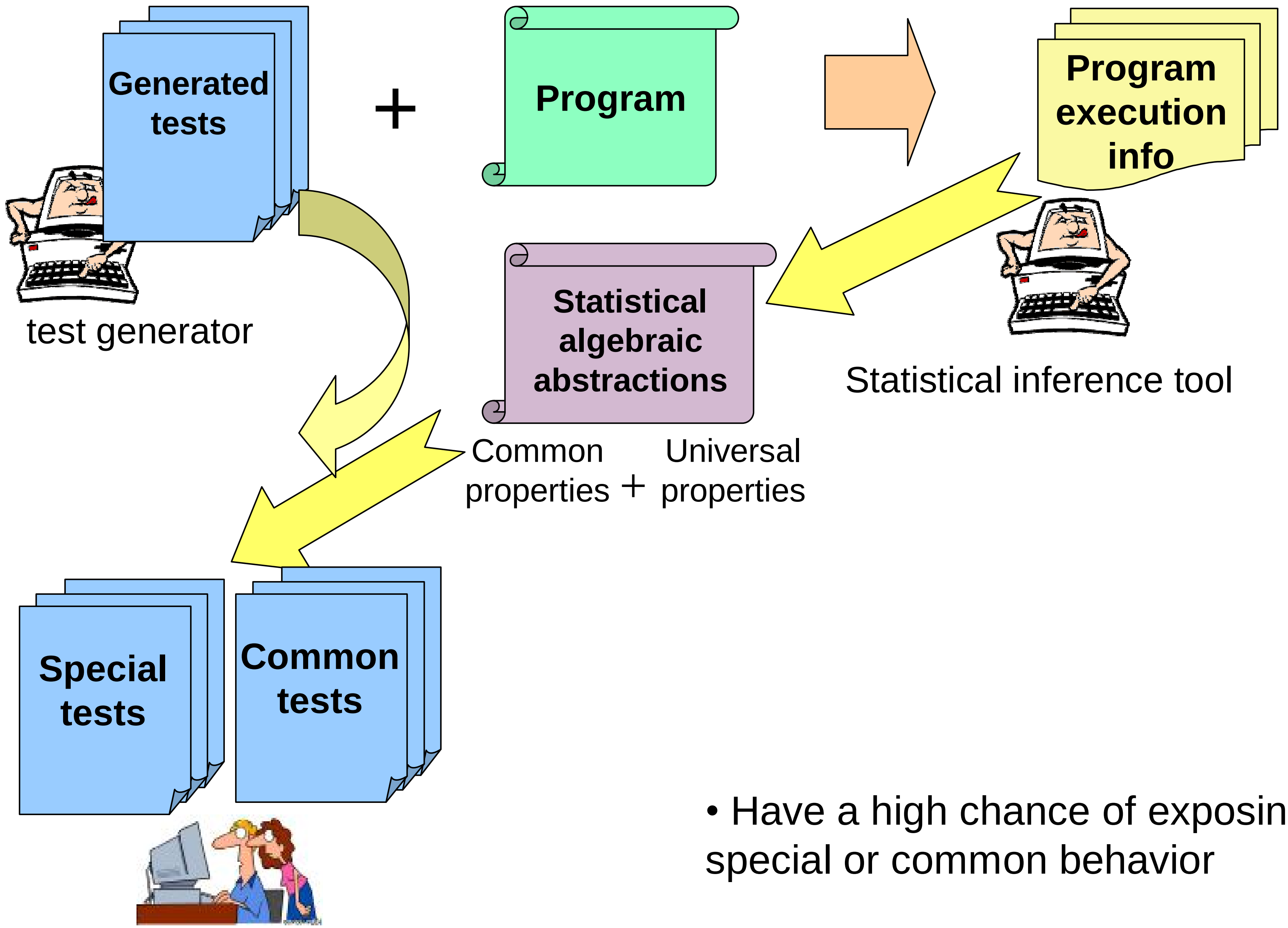
```
uniqueBoundedStack THIS = new uniqueBoundedStack ();
THIS.push (-1);
int RETVAL = THIS.top ();
```

Agitar Agitator

- Many awards and successful user stories
- Version 1.5 released in June 2004
  - Automatically generate initial tests
  - Infer Daikon-invariant-like observations
  - Developers confirm these observations to assertions
  - Generate more tests to violate these inferred & confirmed observations

Agitar® SOFTWARE | Develop Software With Confidence

Test Selection based on Special Behavior



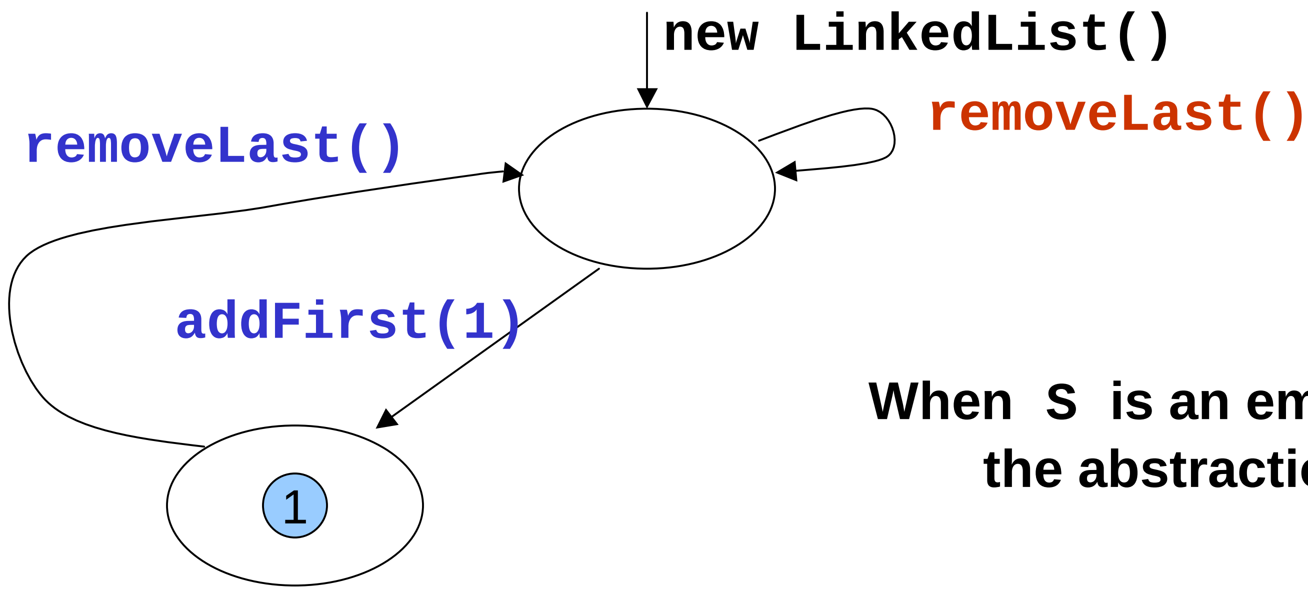
- Have a high chance of exposing special or common behavior

Example

- `removeLast(addFirst(S, e).state).state == addFirst(removeLast(S).state, e).state`

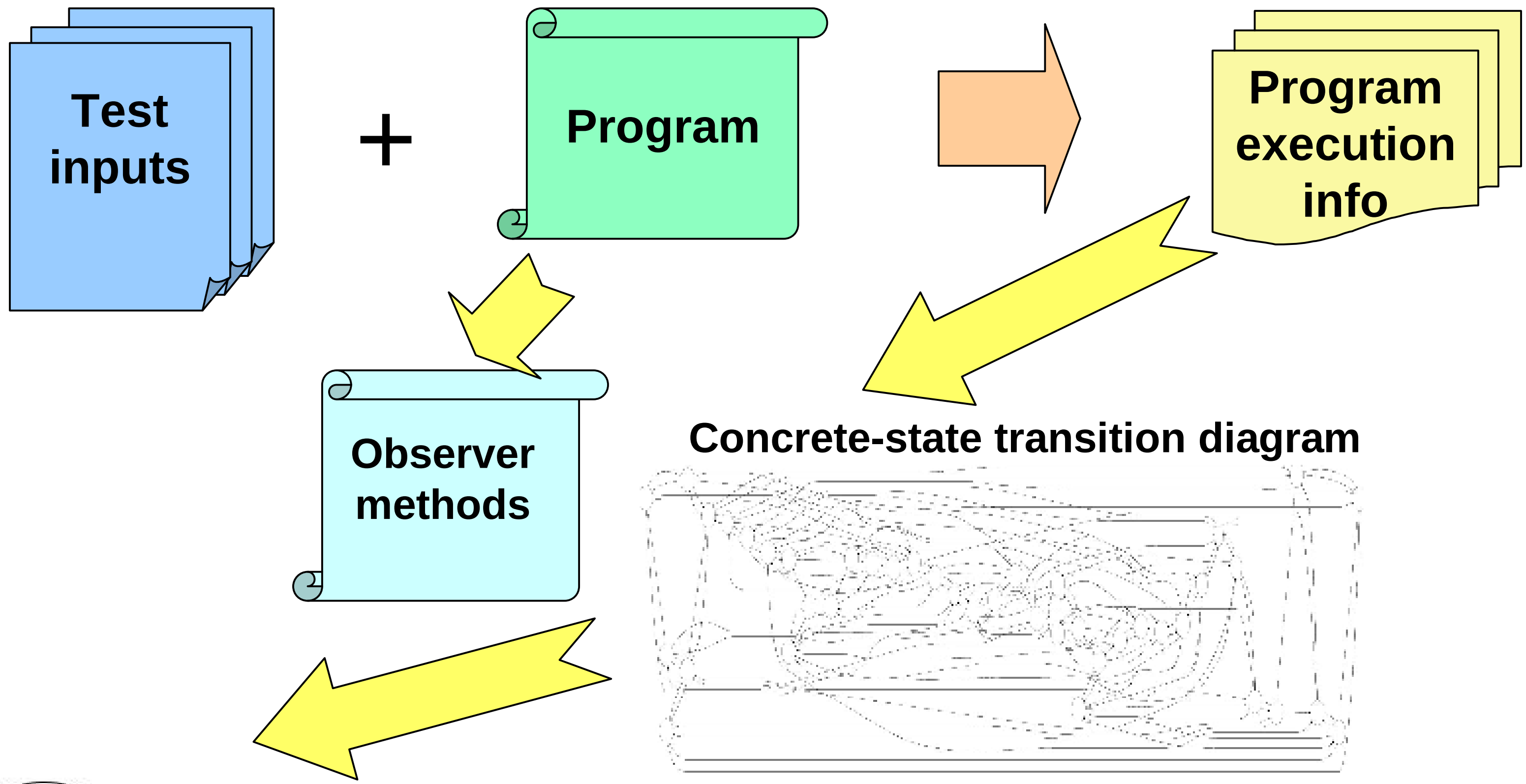
  117 satisfying instances

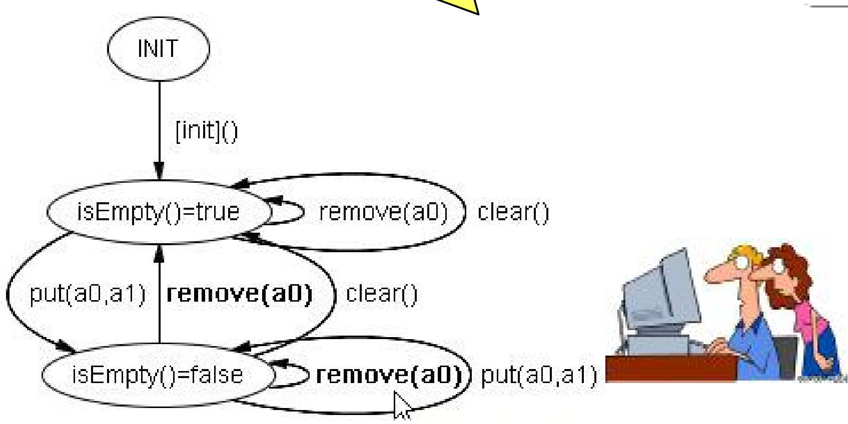
  3 violating instances



When `S` is an empty `LinkedList`, the abstraction is violated.

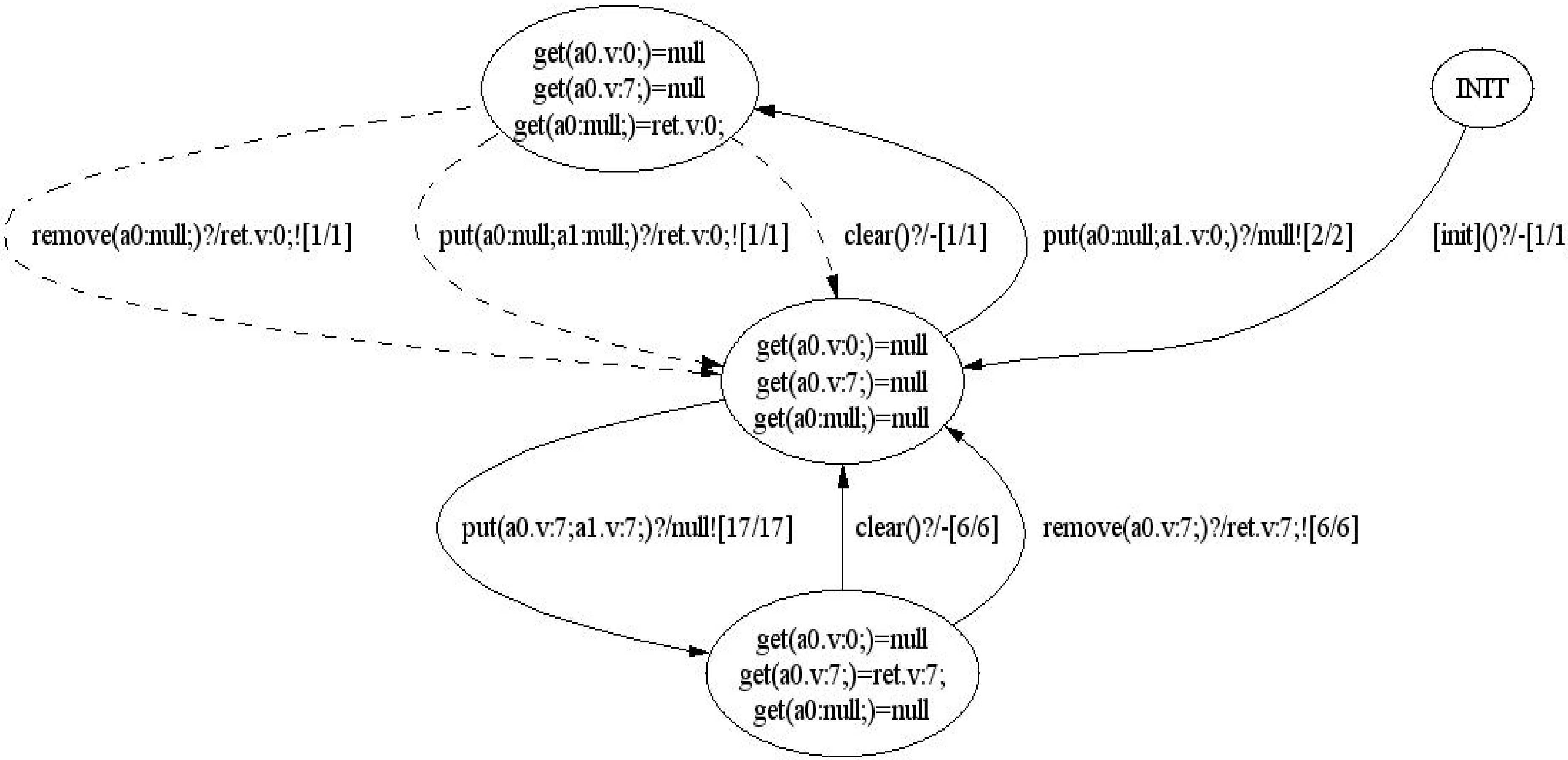
Test Abstraction for Overall Behavior





• Help expose and diagnose program problems, understand interface

Example



- **Suspicious transition:** **`put(a0:null;a1:null;)?/ret.v:0![1/1]`**
- **Expose an error in Java API doc for HashMap**

Summary

- Automated software testing to reduce manual efforts
- Specifications to help test-input generation and test-behavior checking
  - but they often don't exist
- "test-infer-test-infer…"
  - test-input generation in the absence of class invariants
    - state representation: concrete vs. symbolic [ASE 04, TACAS 05]
  - test-behavior checking
    - axiomatic abstractions: test selection [ASE 03]
    - algebraic abstractions: test selection [ISSRE 05]
    - abstract object-state machines: test abstraction [ICFEM 04]

Questions?