Automated Test-Input Generation

Tao Xie

North Carolina State University
Department of Computer Science
Nov 2005

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

Why Automate Testing?

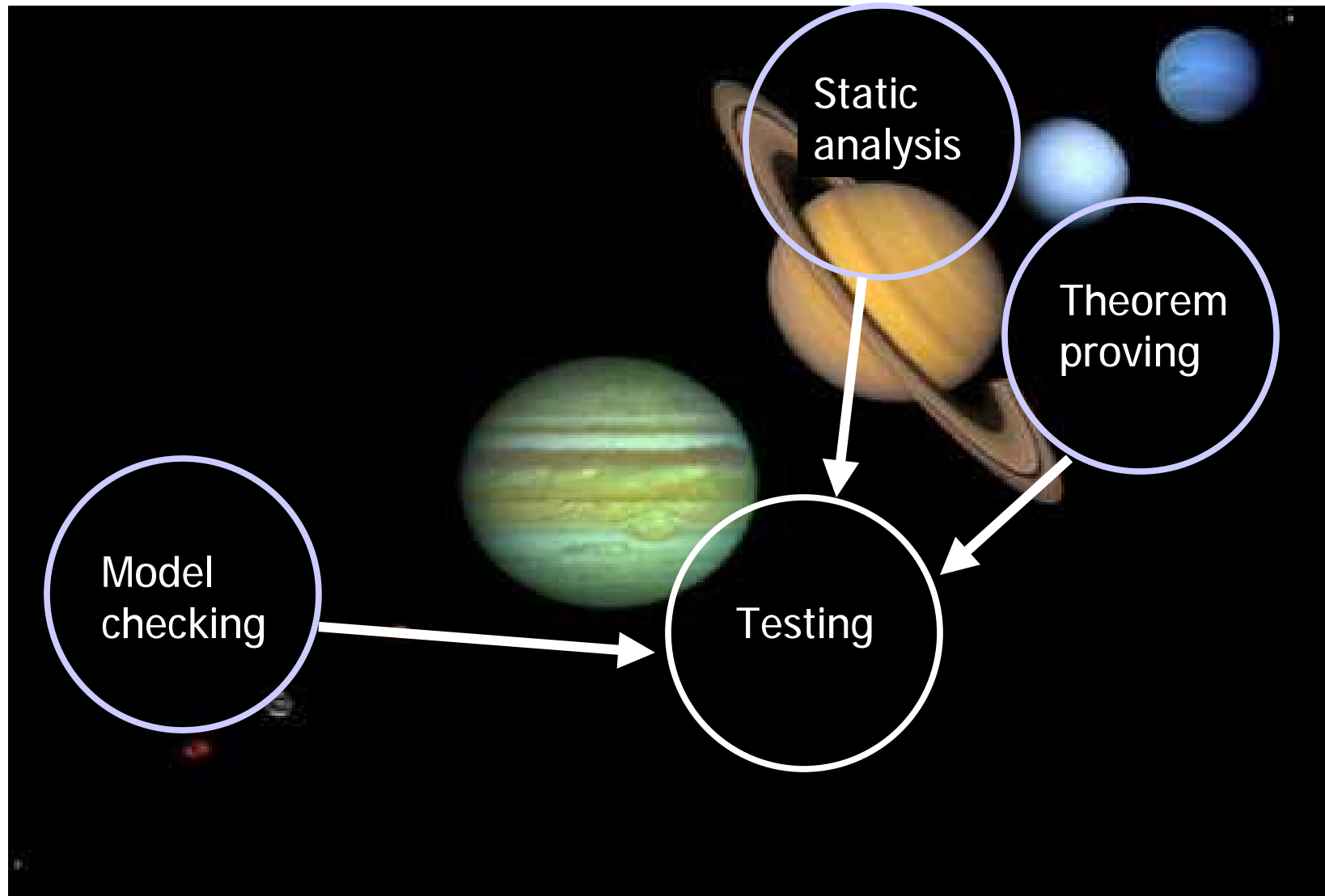- Software testing is important
  - Software errors cost the U.S. economy about $59.5 billion each year (0.6% of the GDP) [NIST 02]
  - Improving testing infrastructure could save 1/3 cost

- 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 inspection: Agitar Agitator

Software Quality Assurance

Static
analysis

Theorem
proving

Model
checking

Testing
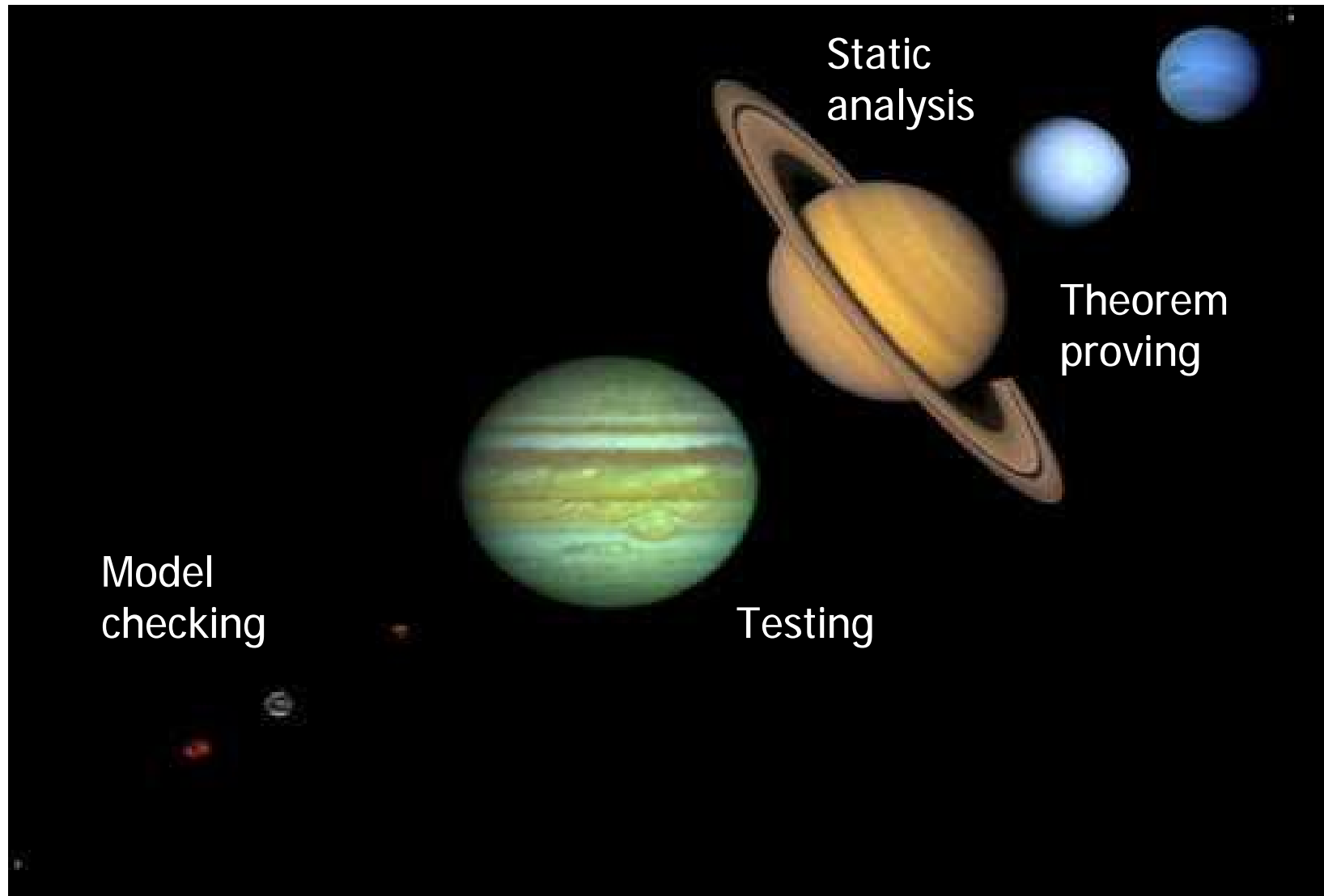
Adapted from Doron Peled's slide

Software Quality Assurance

Binary Search Tree Example

```java
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) { … }
}
```

Approaches

- Method-sequence exploration
  - Parasoft Jtest 4.5 and others

- Specification-based state exploration
  - TestEra [Marinov&Khurshid ASE 01], Korat [Boyapati et al. ISSTA 02]

- Concrete-state exploration
  - Rostra [Xie et al. ASE 04], NASA JPF [Visser et al. ISSTA 04]

- Symbolic-state exploration
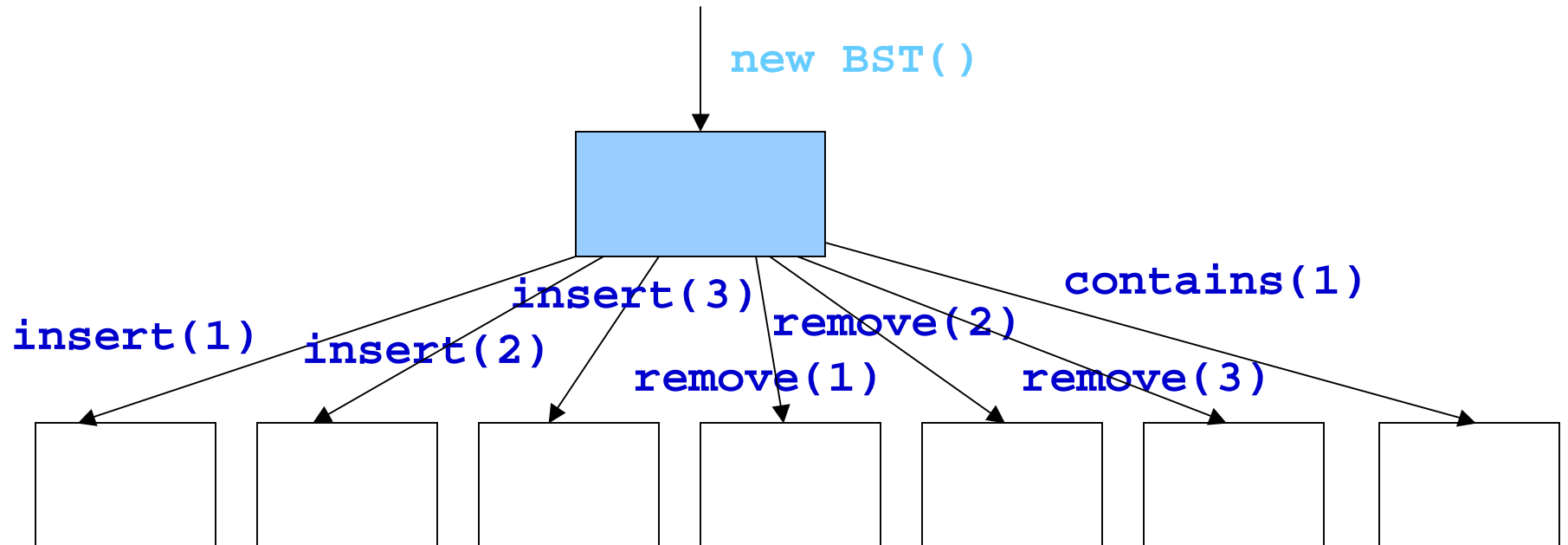 - Symstra [Xie et al. TACAS 05]

Exploring Method Sequences

- Method arguments: `insert(1), insert(2), insert(3),`
`remove(1), remove(2), remove(3), contains(1)`
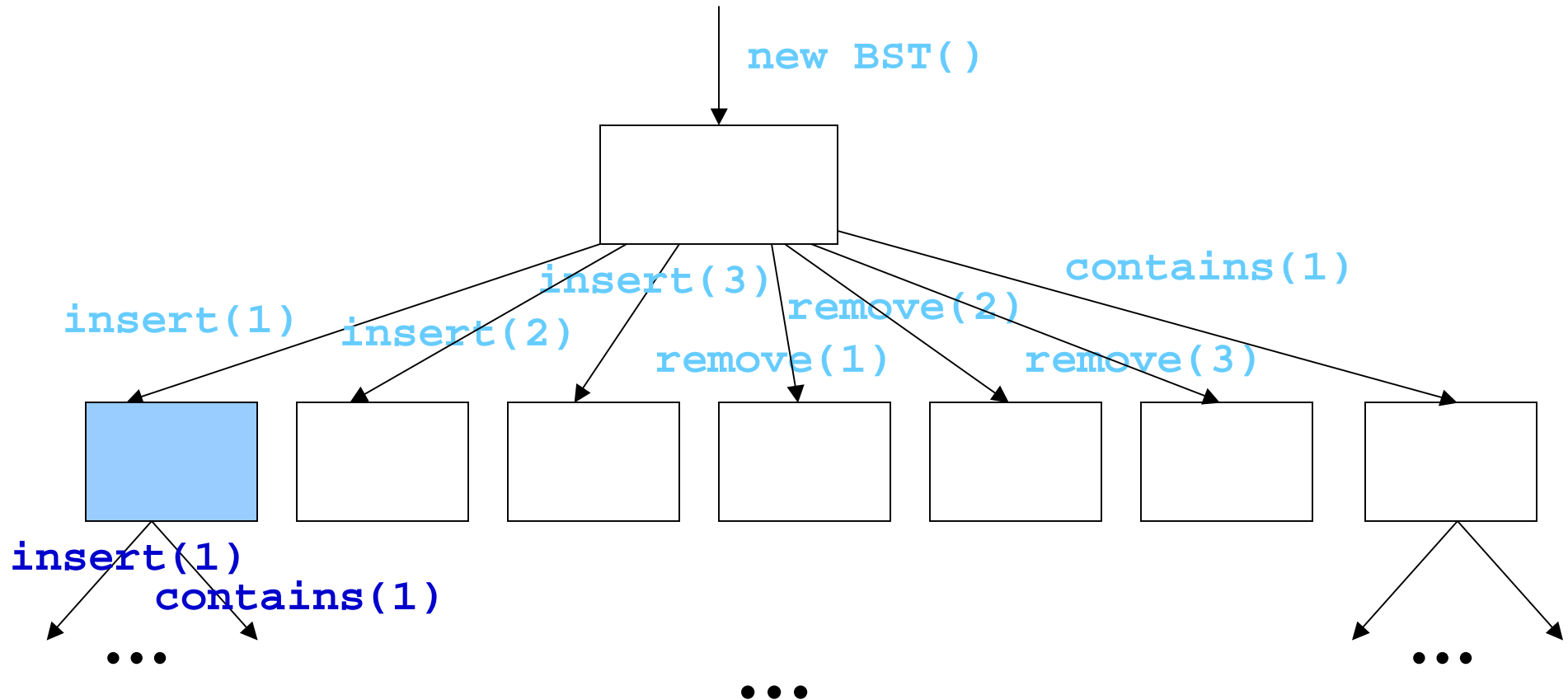
`new BST()`
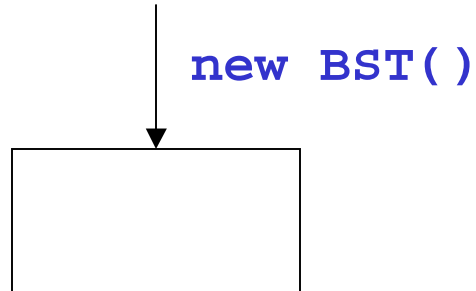
Exploring Method Sequences

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

Iteration 1

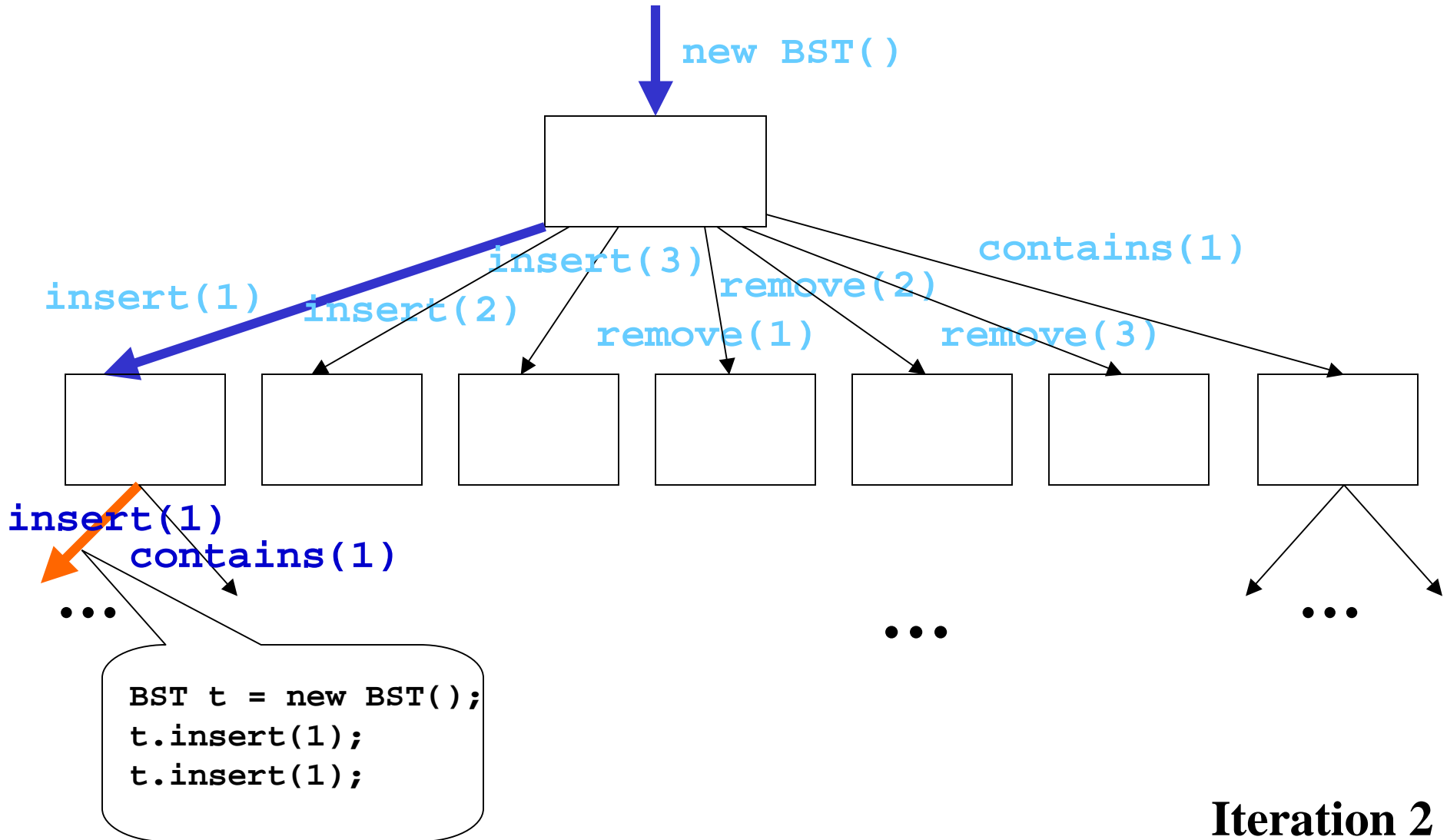# Exploring Method Sequences

- 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)

new BST()

insert(1) insert(2) insert(3) remove(1) remove(2) remove(3) contains(1)

insert(1)
contains(1)

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

Iteration 2

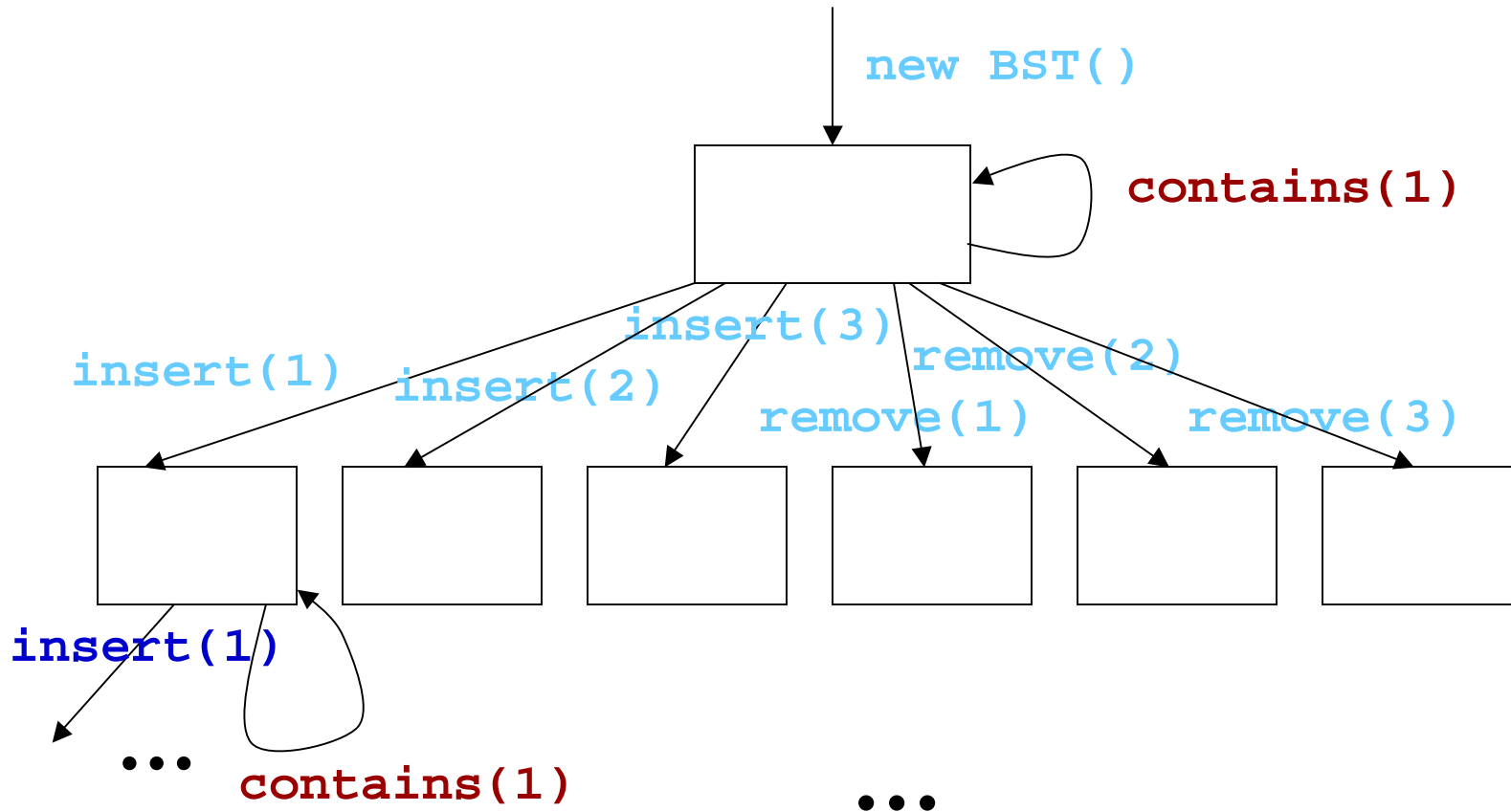# Pruning State-Preserving Methods

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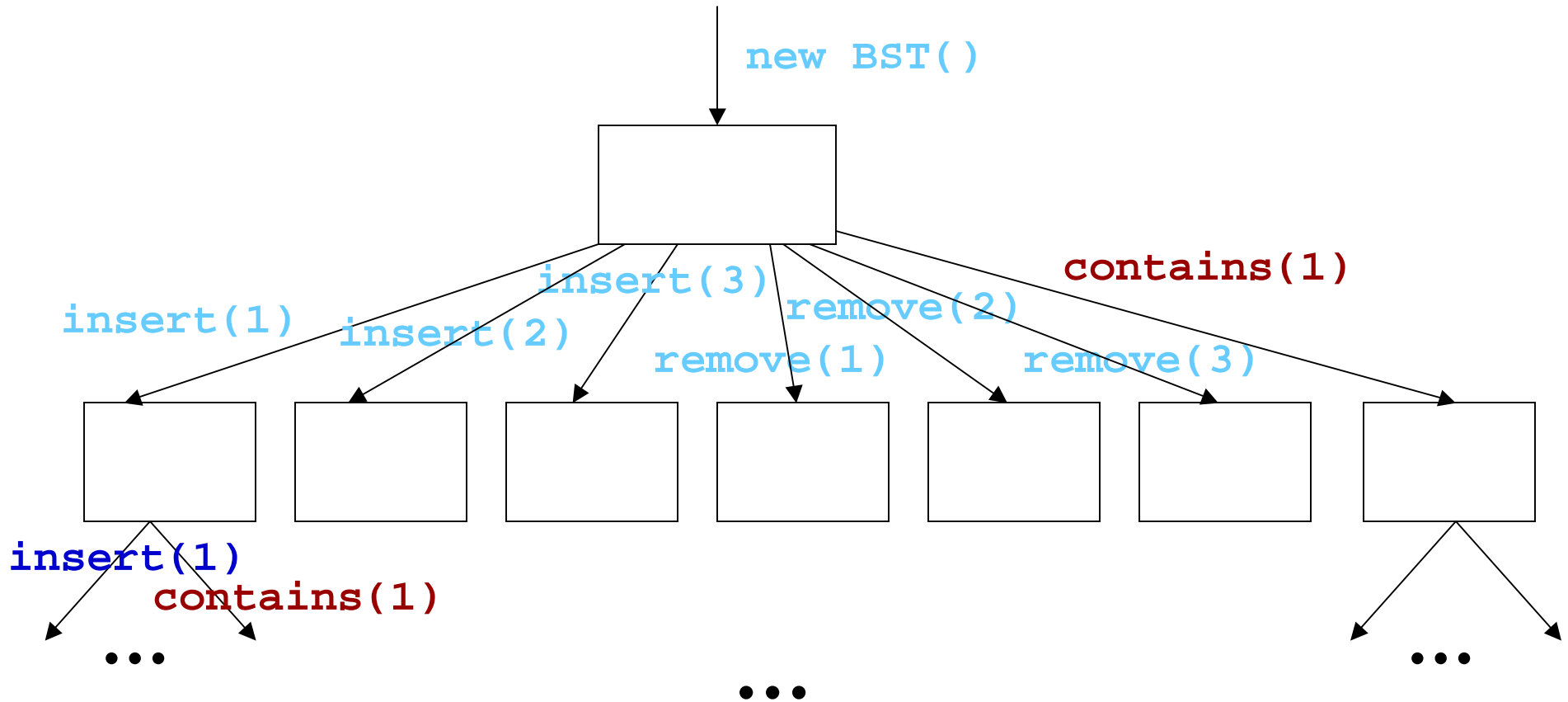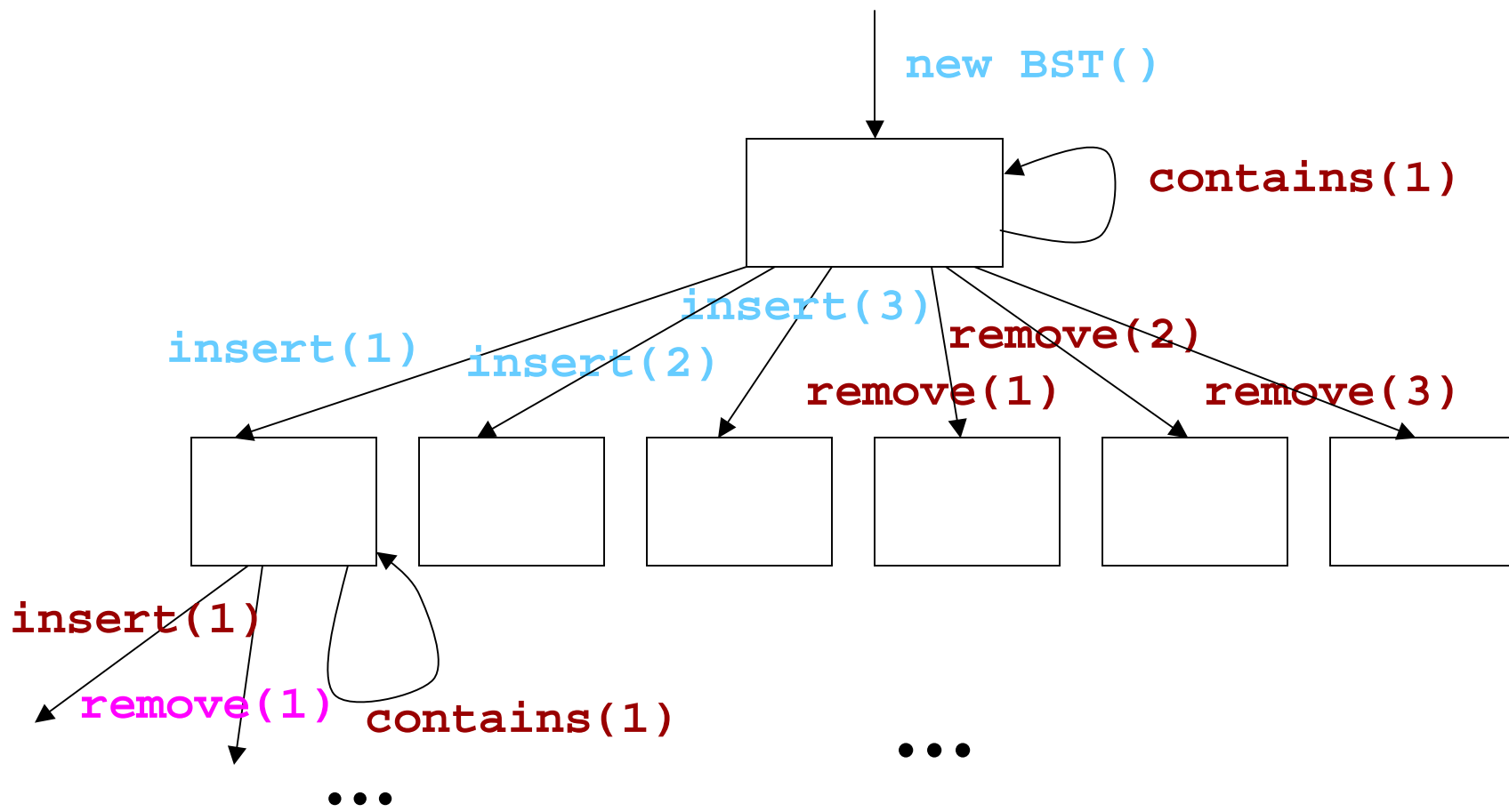

Iteration 2

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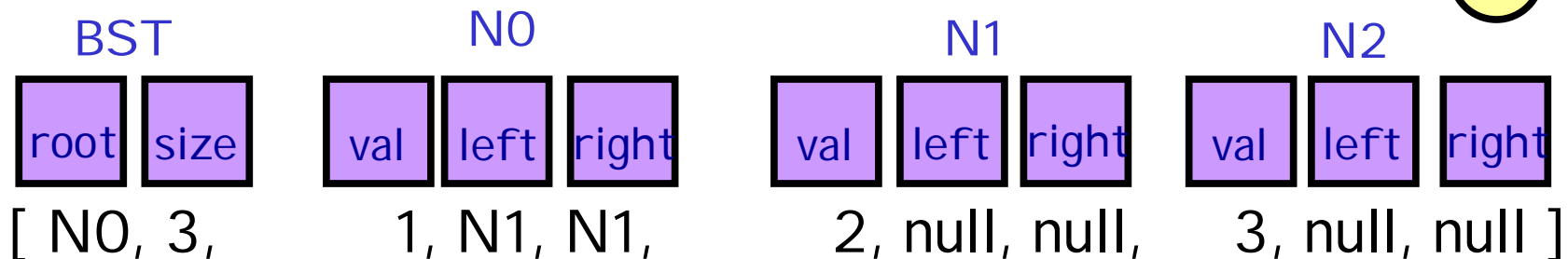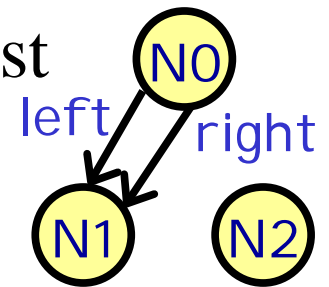
- Given predicate p and finitization f, generate all inputs for which p returns "*true*"
  - uses finitization to define input space
    - e.g., defines #nodes and what values can be on a BST node.
  - systematically explores valid input space
    - prunes input space using field accesses

3 nodes
Node vals: {1, 2, 3}
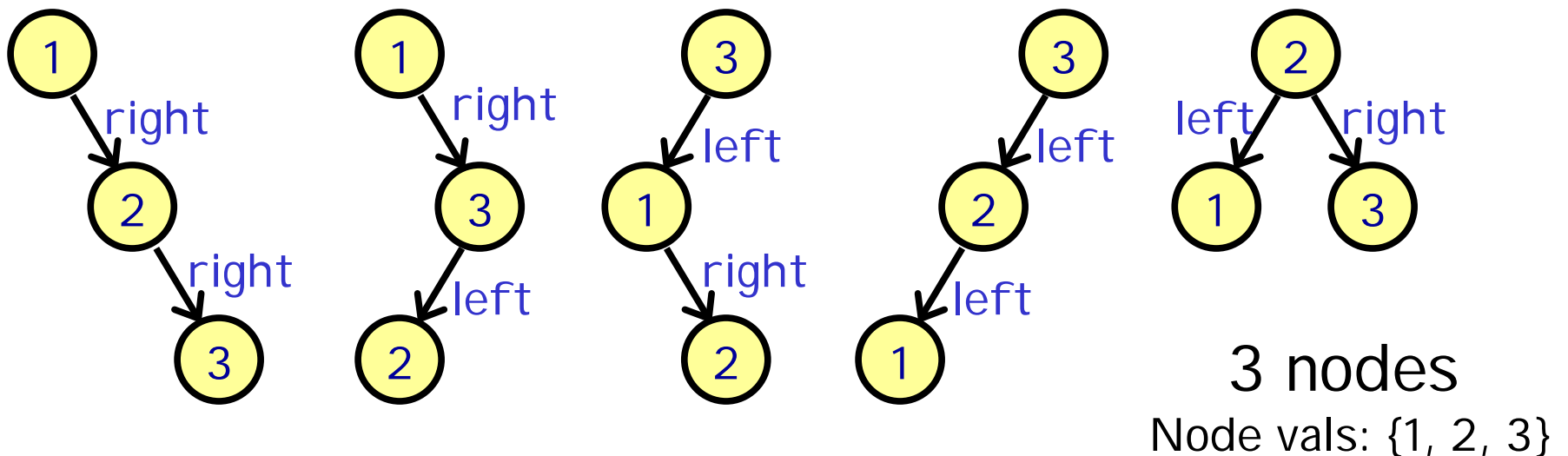
Korat's Smart Search

- candidate is a vector of field domain indices
- for each candidate vector, Korat
  - creates corresponding structure
  - invokes **repOk** and monitors the execution
  - builds field ordering, i.e., list of fields ordered by time of first access
  - if **repOk** returns "**true**", outputs structure(s)
  - if **repOk** returns "**false**", backtracks on the last field accessed using field ordering

BST
| root | size |

N0
| val | left | right |

N1
| val | left | right |

N2
| val | left | right |

[N0, 3,   1, N1, N1,   2, null, null,   3, null, null]

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)
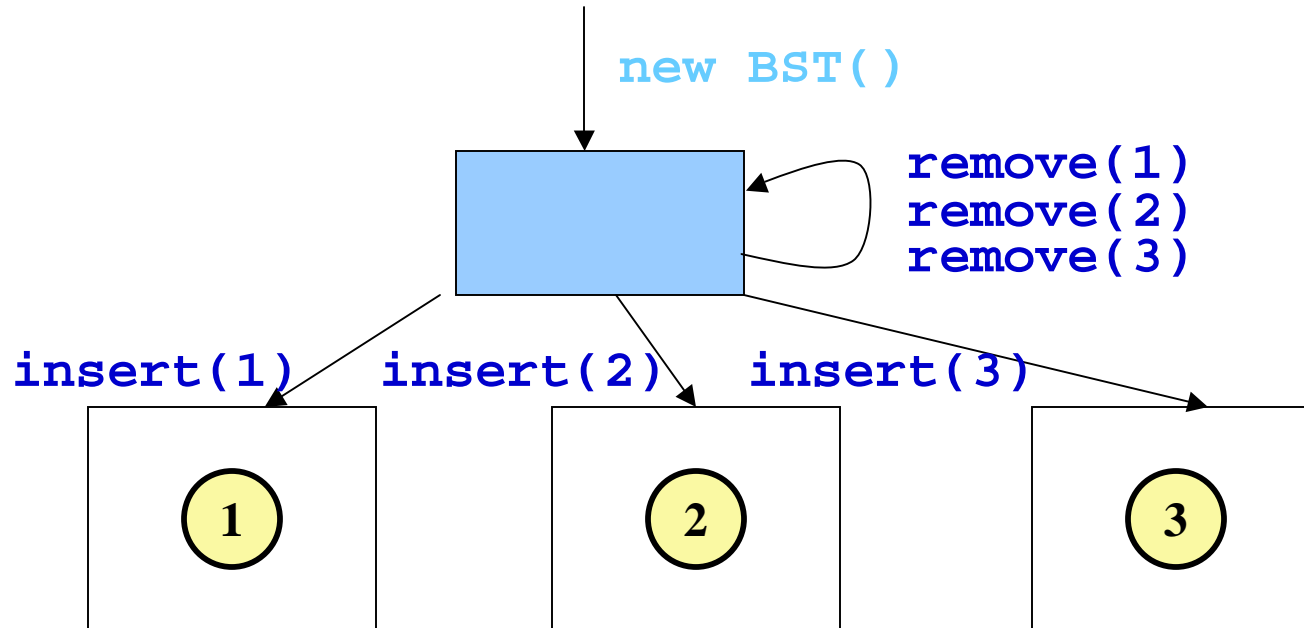
Exploring Concrete States

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

`new BST()`

Exploring Concrete States

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

`new BST()`

`remove(1)`
`remove(2)`
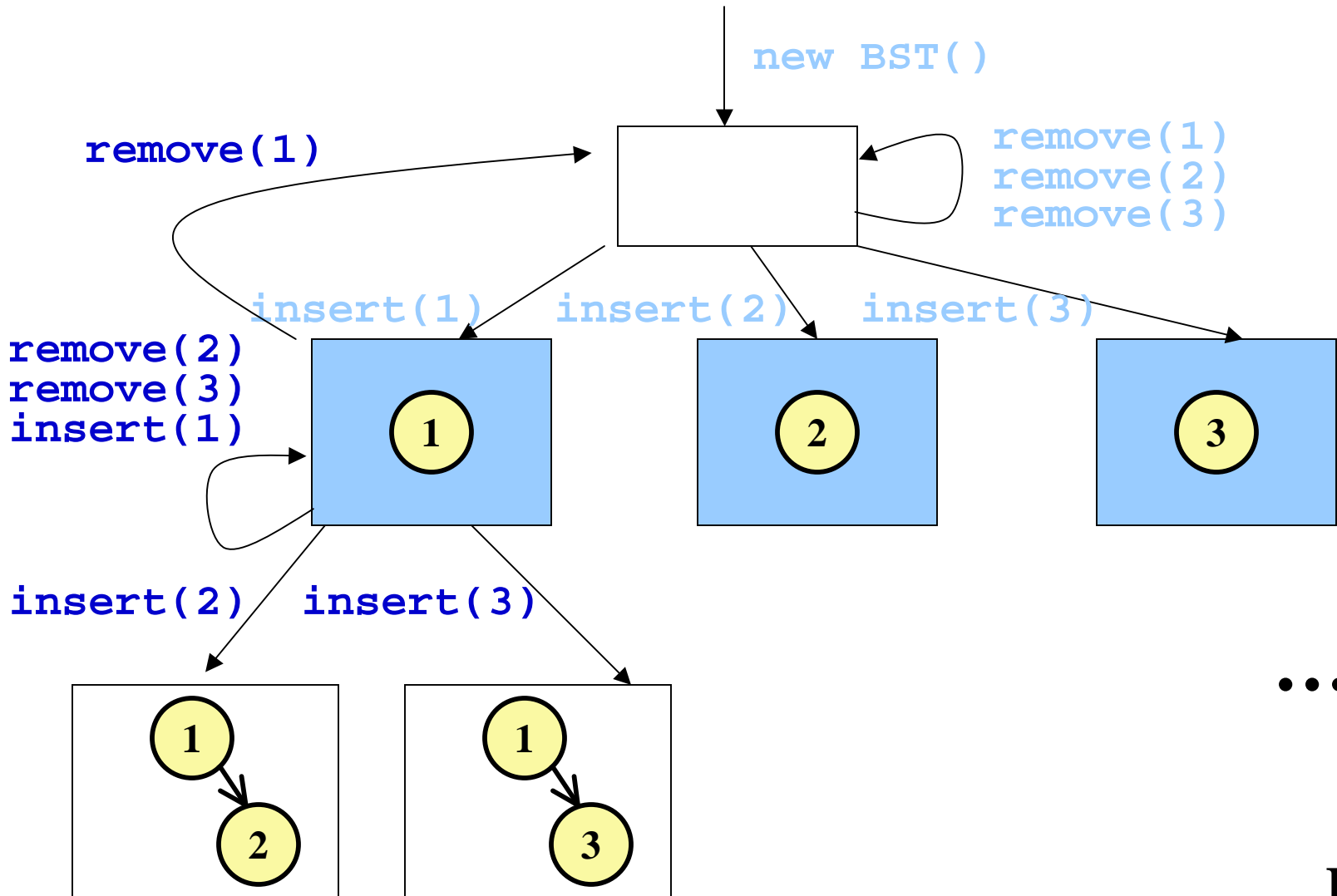`remove(3)`

`insert(1)` `insert(2)` `insert(3)`

1   2   3
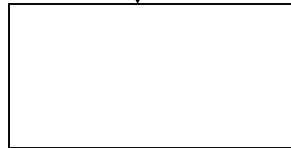
Iteration 1

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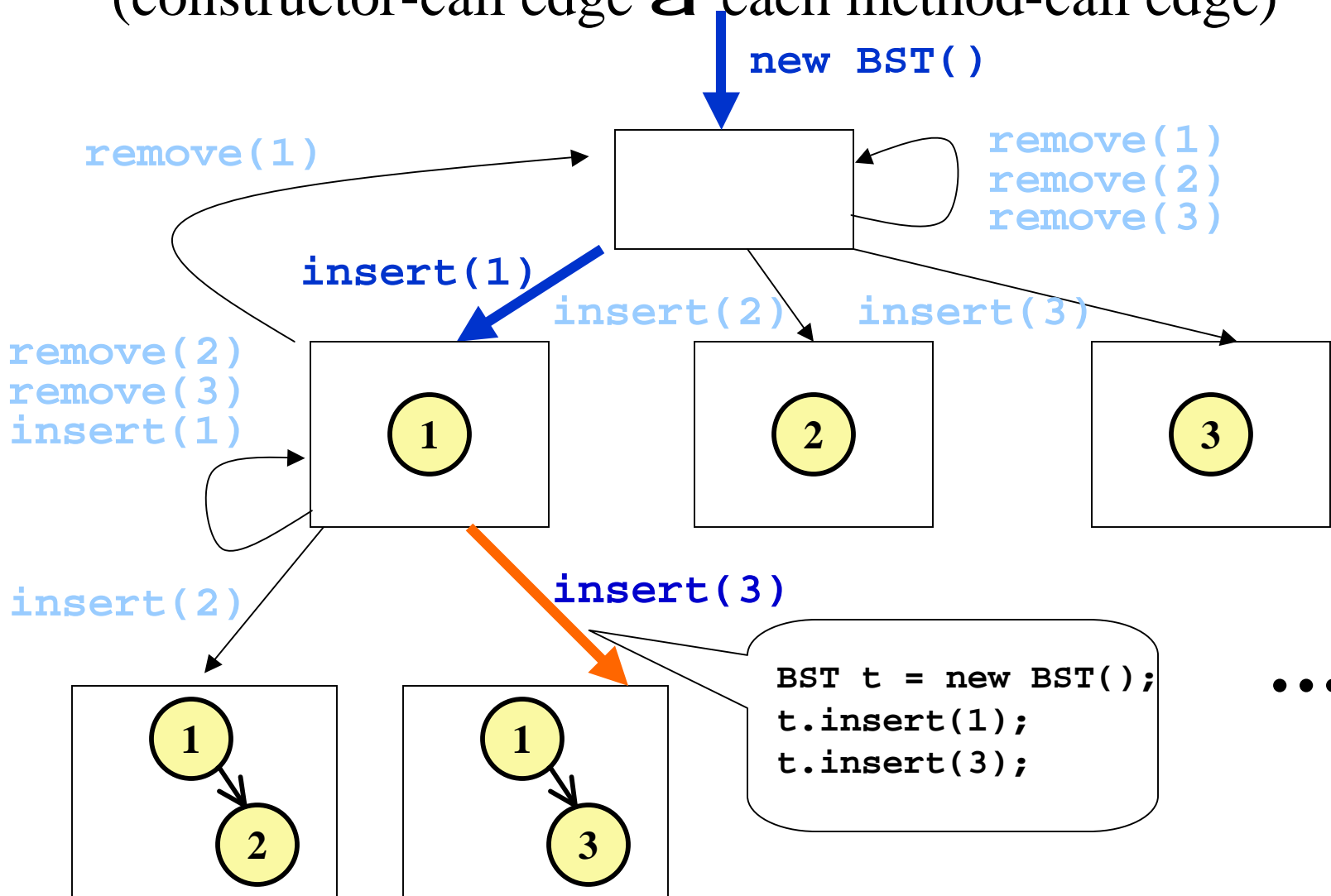
- Method arguments: `insert(1), insert(2), insert(3), remove(1), remove(2), remove(3)`

Iteration 2

Generating Tests from Exploration

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

new BST()

remove(1)

remove(1)
remove(2)
remove(3)

insert(1)

insert(2)   insert(3)

remove(2)
remove(3)
insert(1)

1

2

3

insert(2)

insert(3)

1
2

1
3

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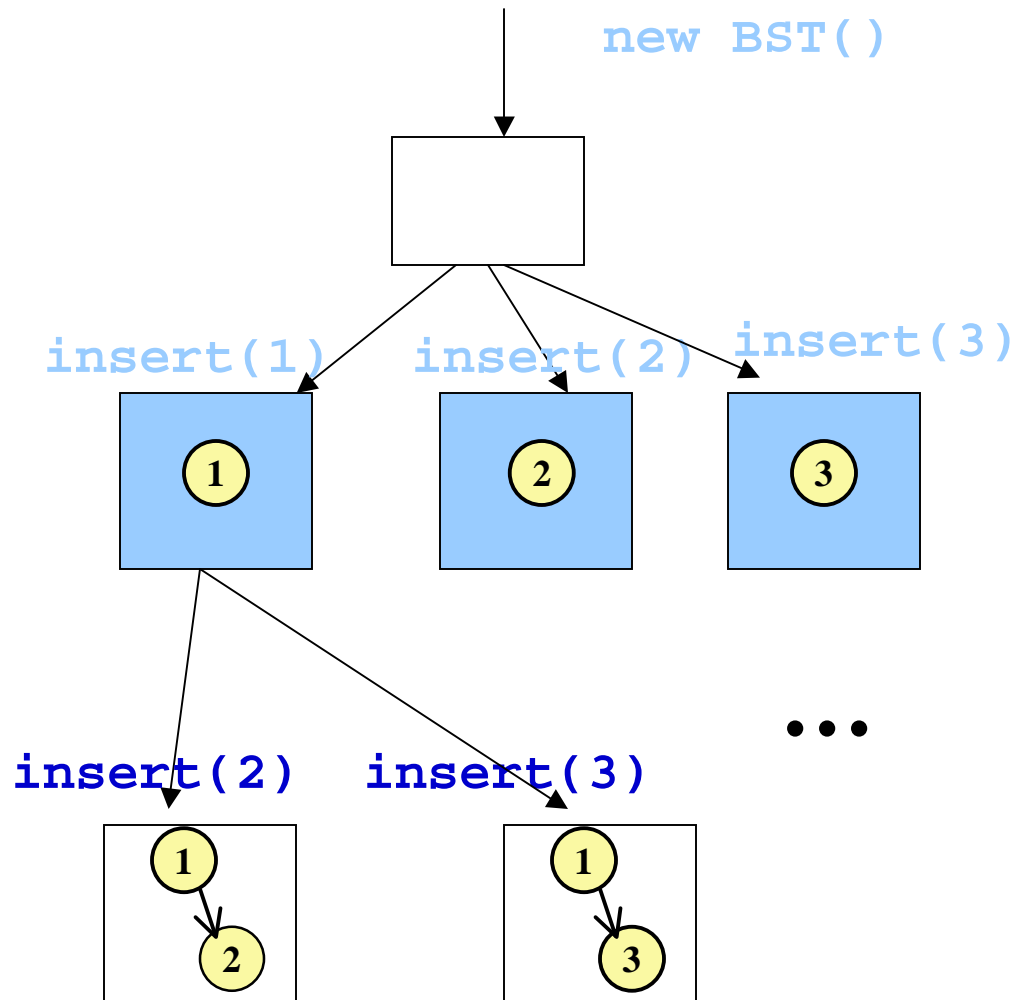
•••

Improvement over
Method-Sequence Exploration

- Industry standard 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

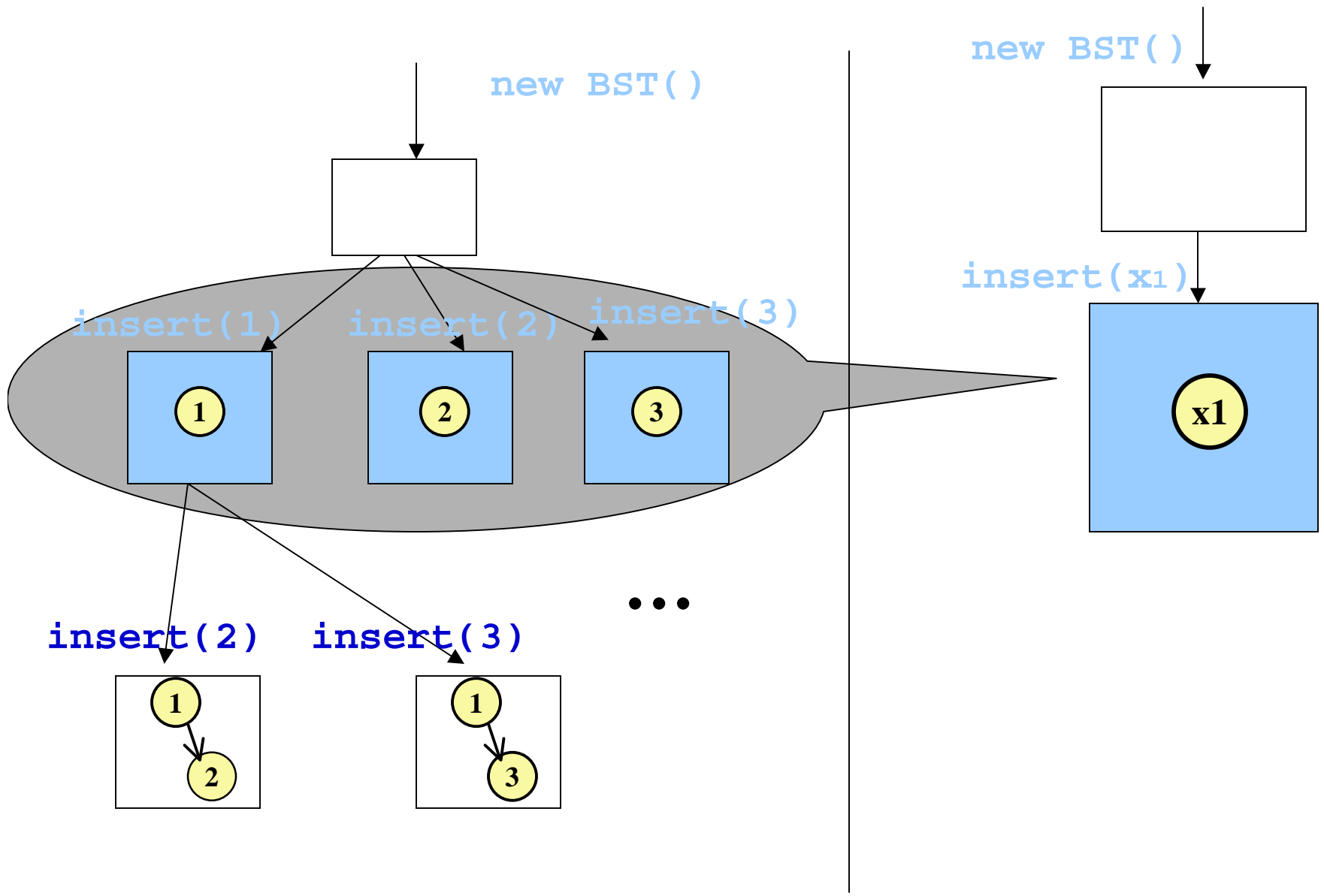# Issues of Concrete-State Exploration

- 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

new BST()

insert(1)   insert(2)   insert(3)

insert(2)   insert(3)

. . .

Iteration 2

State Abstraction: Symbolic States

new BST()

new BST()

insert(1)   insert(2)   insert(3)

insert(x₁)

1   2   3

x1

· · ·

insert(2)   insert(3)

1   1

2   3

Iteration 2

State Abstraction: Symbolic States

new BST()

insert(1)   insert(2)   insert(3)

1   2   3

...

insert(2)   insert(3)

1 → 2   1 → 3

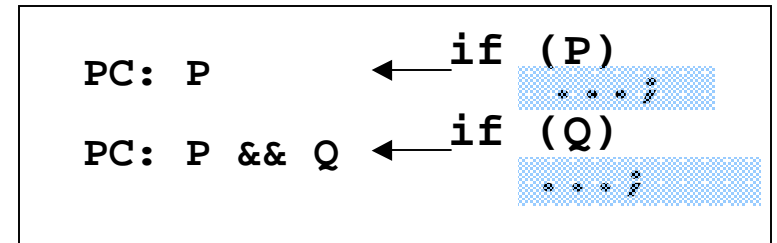new BST()

insert(x1)

x1

insert(x2)

x1 < x2

x1 → x2
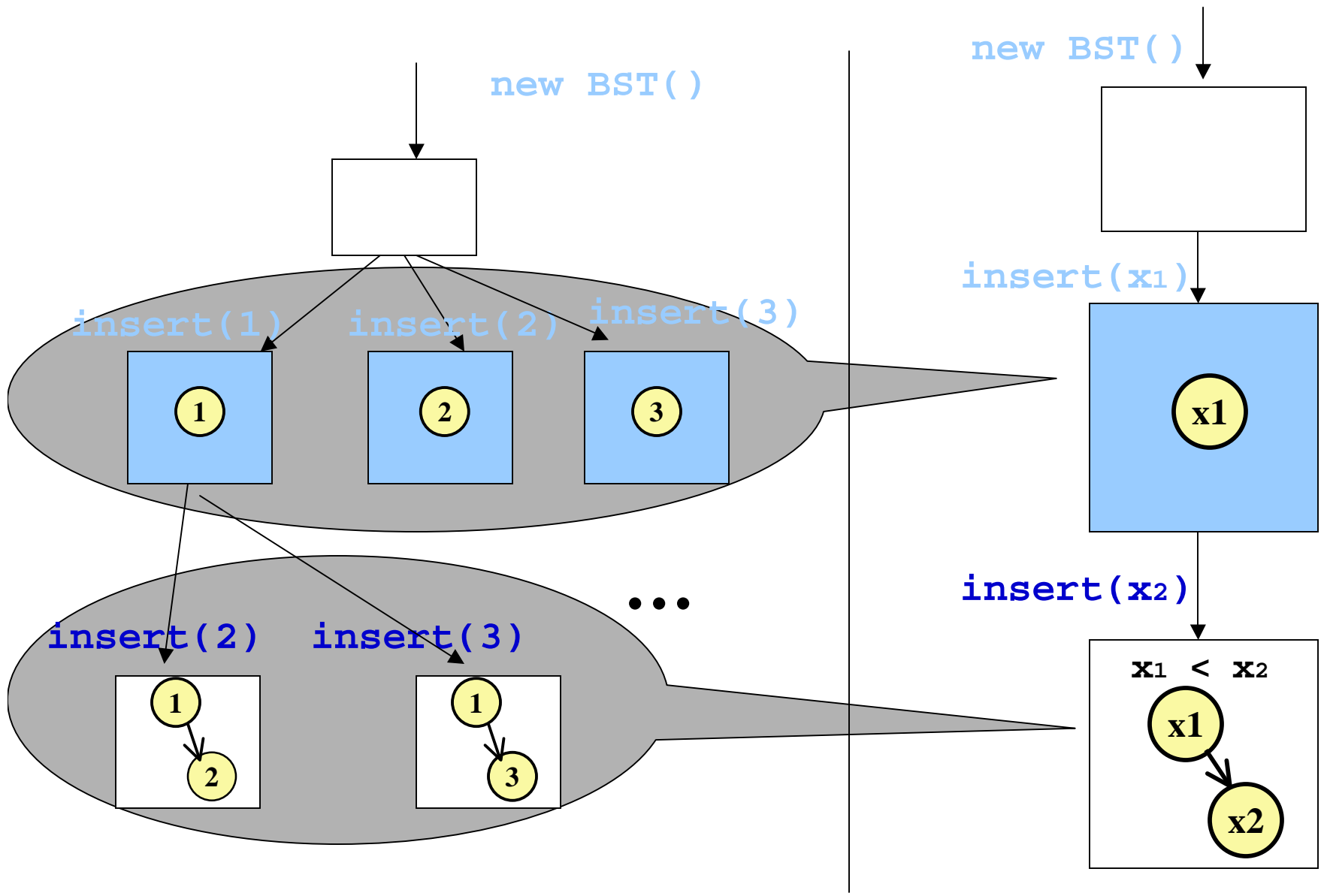
Iteration 2

Symbolic Execution

- Execute a method on symbolic input values

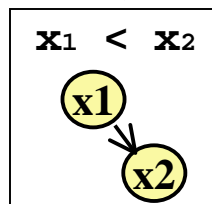 - inputs: `insert(SymbolicInt x)`

- 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.,
```
x₁ < x₂
  x1
    ↓
    x2
```

Symbolic Execution Example

```java
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++;
}
```
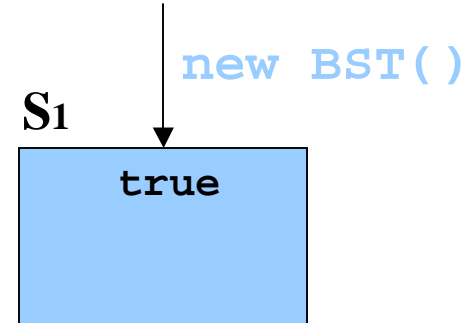
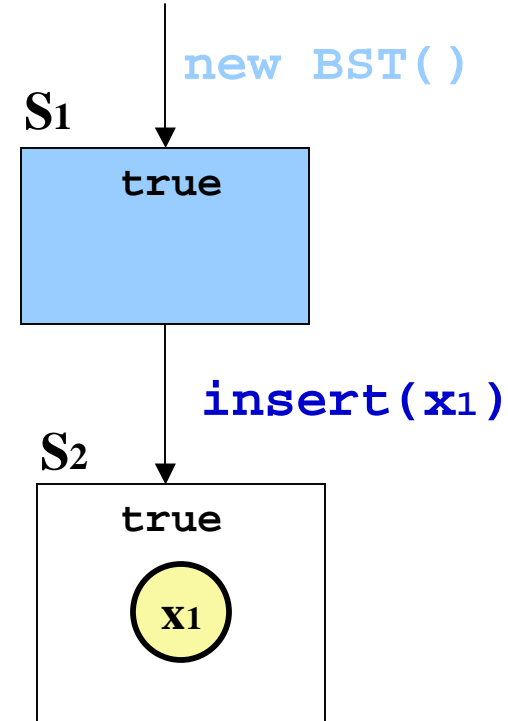S1

true

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++;
}
```

new BST()

S₁

true

insert(x₁)

S₂

true

x₁

Iteration 1

Exploring Symbolic States
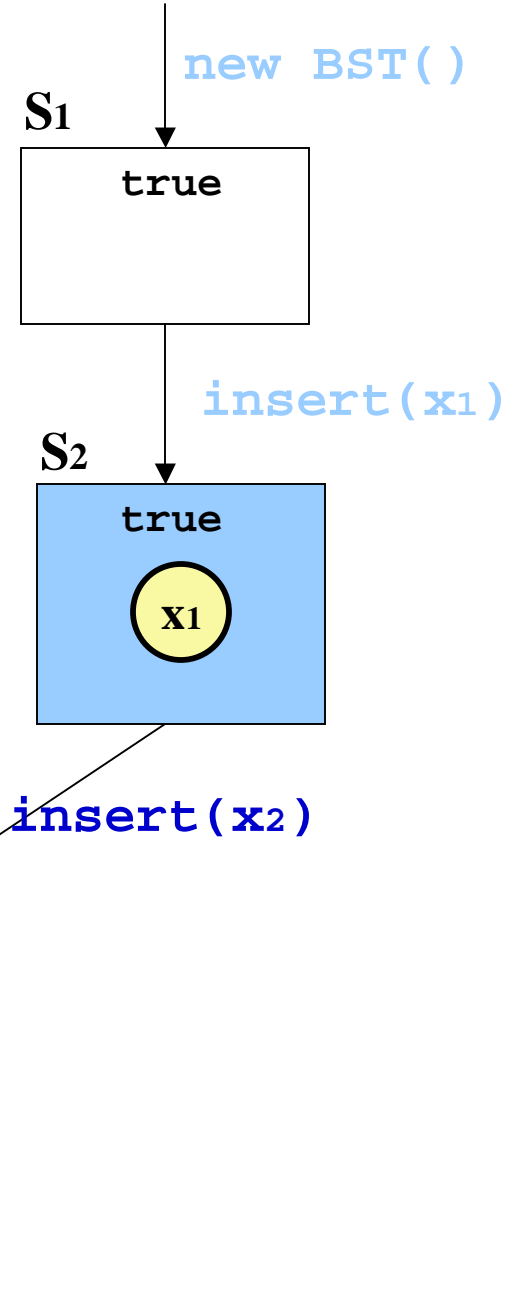
```
public void insert(SymbolicInt x) {
  if (root == null) {
    root = new Node(x);
  } else {
    Node t = root;
    while (true) {
      if (t.value < x) {
        //explore the right subtree
        ...
      } else if (t.value > x) {
        //explore the left subtree
        ...
      } else return;
    }
  }
  size++;
}
```

new BST()

S₁

true

insert(x₁)

S₂

true

X₁

insert(x₂)

S₃

x₁ < x₂

X₁

X₂

Iteration 2

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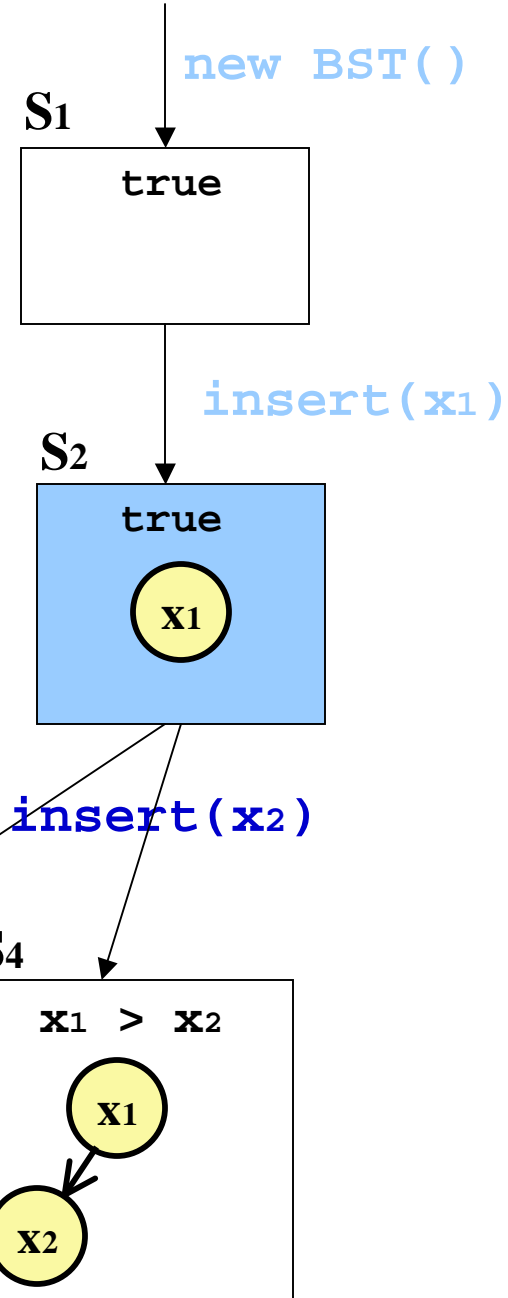
```
public void insert(SymbolicInt x) {
  if (root == null) {
    root = new Node(x);
  } else {
    Node t = root;
    while (true) {
      if (t.value < x) {
        //explore the right subtree
        ...

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

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

new BST()

S₁

true

insert(x₁)

S₂

true

x₁

insert(x₂)

S₃

x₁ < x₂

x₁

x₂

S₄

x₁ > x₂

x₁

x₂

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++;
}
```

new BST()

S₁

true

insert(x₁)

S₂

true

X₁

insert(x₂)

S₃

X₁ < X₂

X₁

X₂

S₄

X₁ > X₂

X₁

X₂

S₅

X₁ = X₂

X₁

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++;
}
```
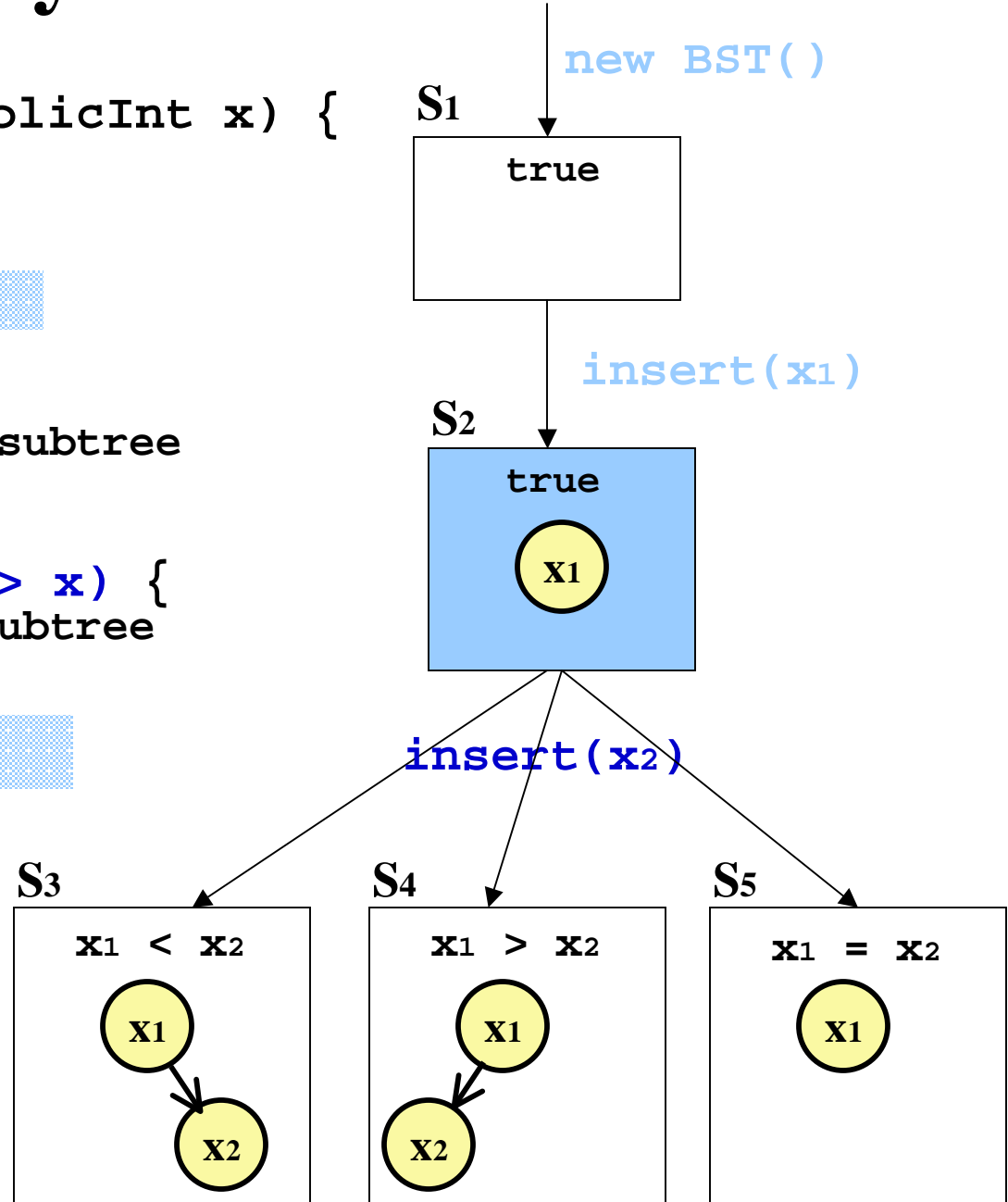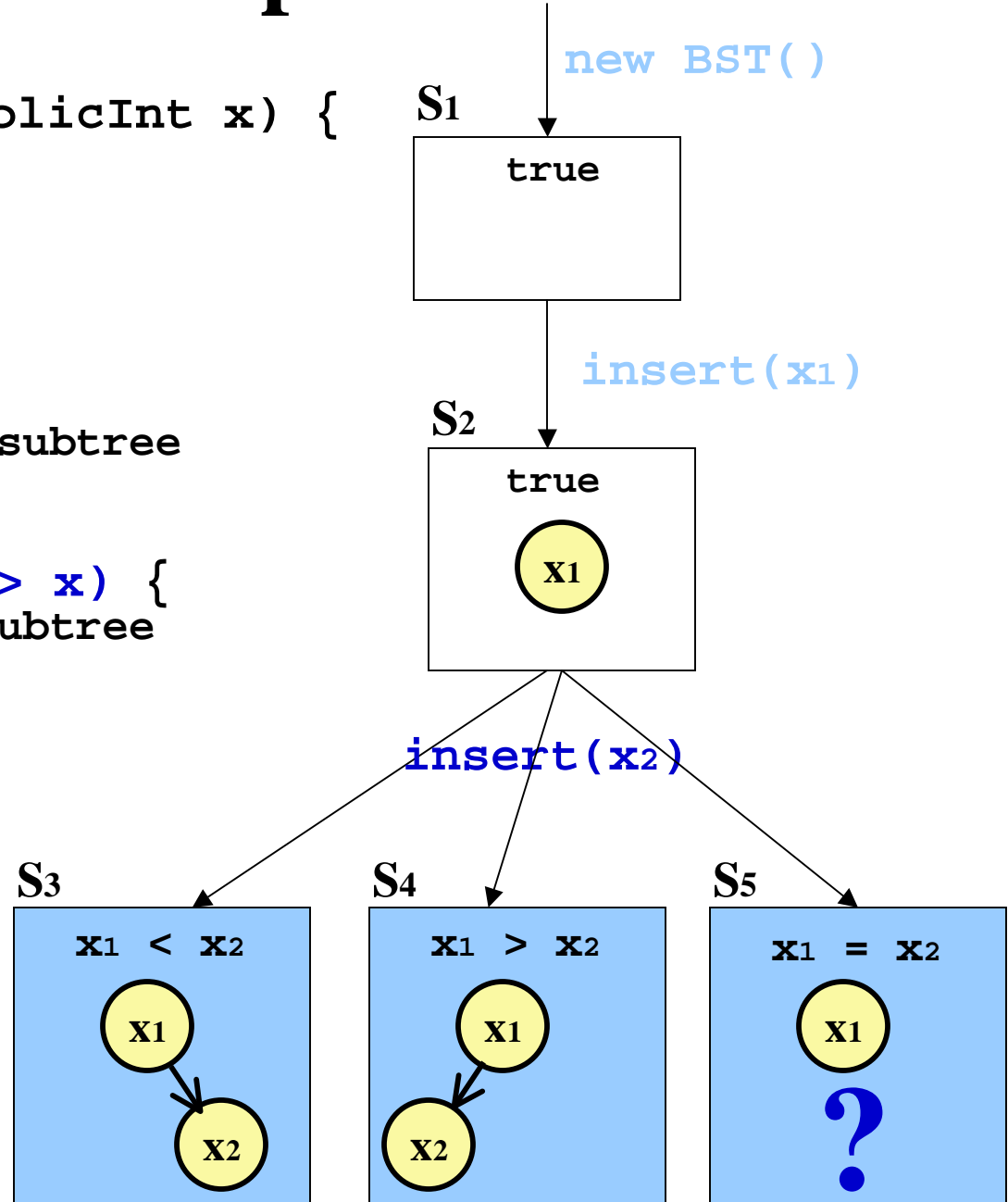
Iteration 3

Symbolic State Subsumption

- Symbolic state `S2:<H2,c2>` subsumes `S5:<H5,c5>`
  - Heaps `H2` and `H5` are isomorphic
  - Path condition `c5` → `c2` [checked using CVC Lite, Omega]
- If `S2` has been explored, `S5` is pruned.
  - Still guarantee path coverage within a method

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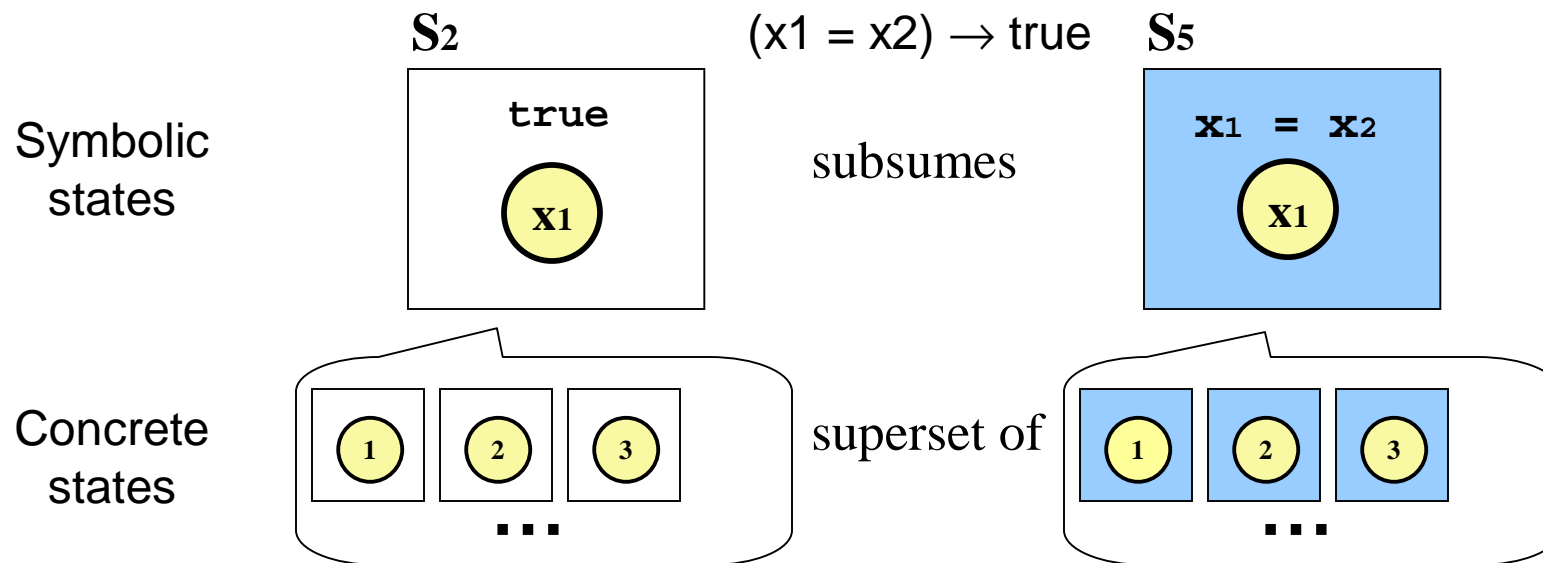
```
public void insert(SymbolicInt x) {
  if (root == null) {
    root = new Node(x);
  } else {
    Node t = root;
    while (true) {
      if (t.value < x) {
        //explore the right subtree
        ...

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

      } else return;
    }
  }
  size++;
}
```
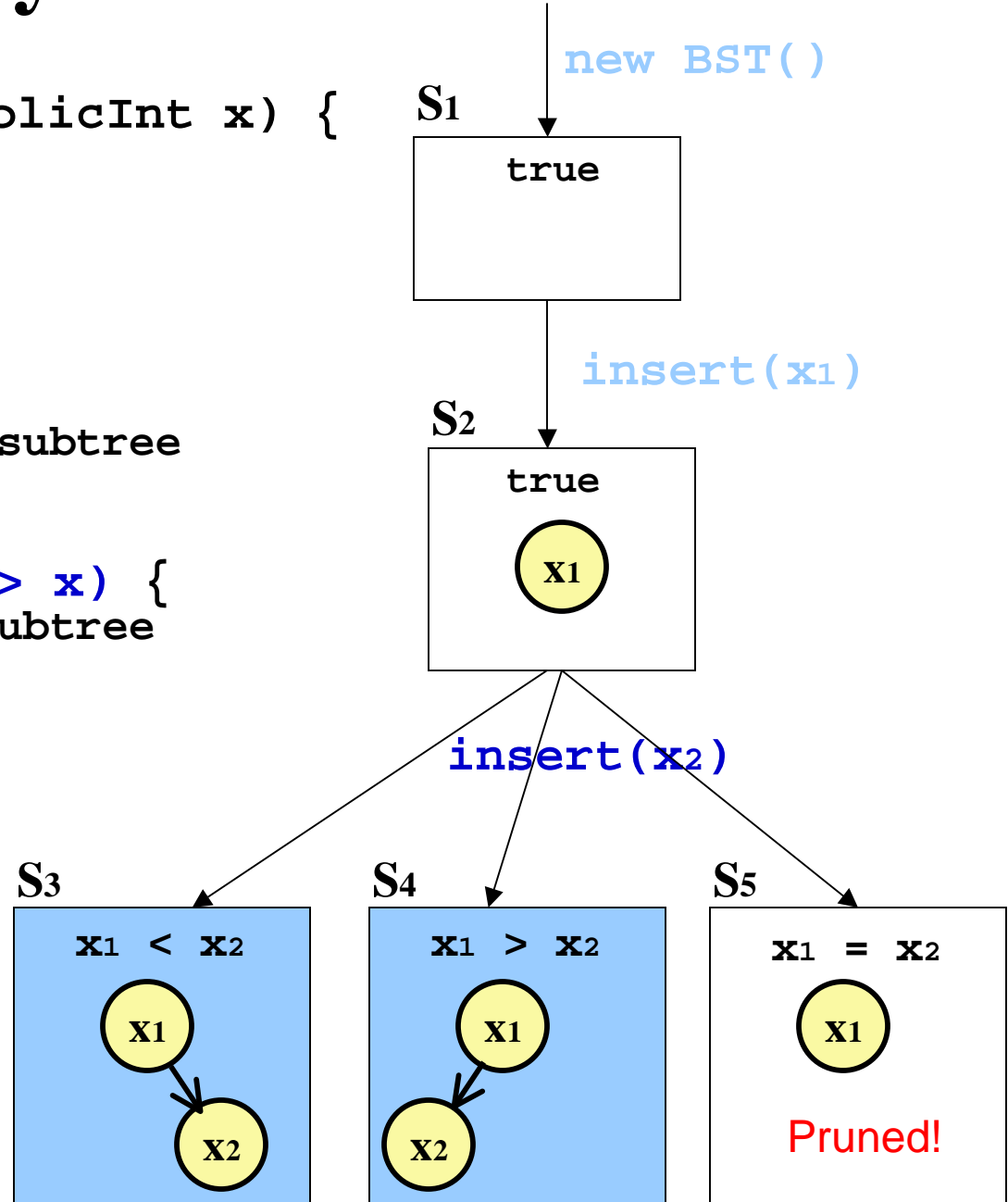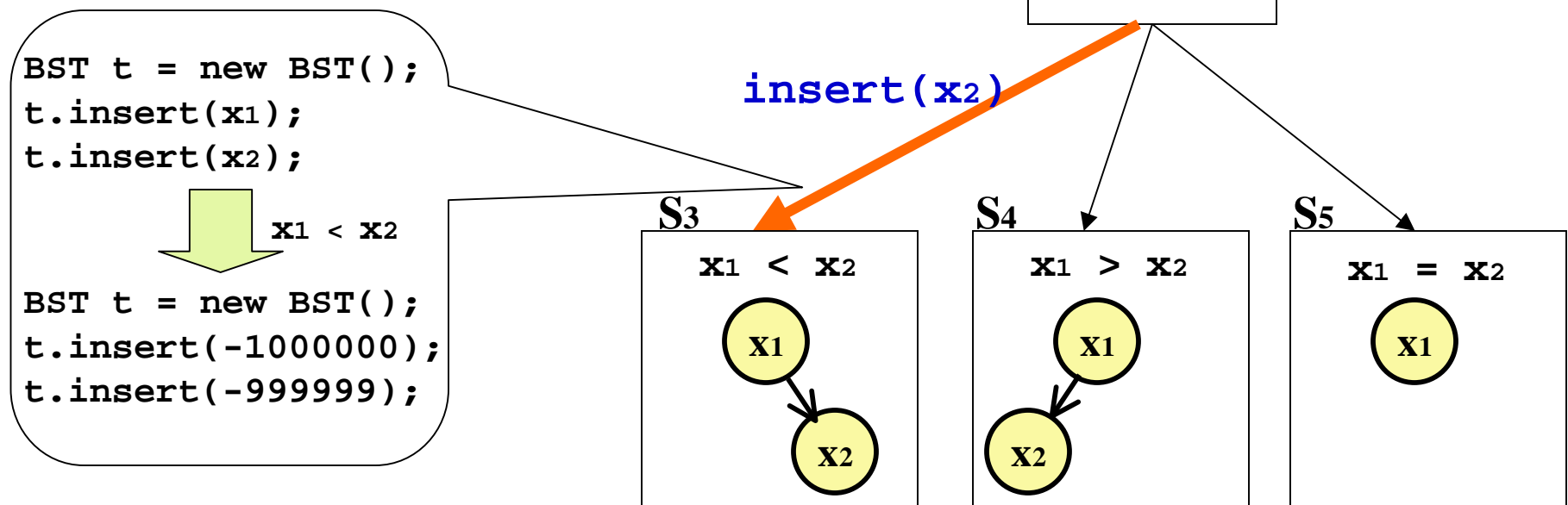
new BST()

S1

true

insert(x1)

S2

true

x1

insert(x2)

S3

x1 < x2

x1

x2

S4

x1 > x2

x1

x2

S5

x1 = x2

x1

Pruned!

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]

new BST()

S_1

```
true
```

insert(x_1)

S_2

```
true
```
x_1

insert(x_2)

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

$x_1 < x_2$

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

S_3

$x_1 < x_2$

x_1
x_2

S_4

$x_1 > x_2$

x_1
x_2

S_5

$x_1 = x_2$

x_1

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

Summary

- ## Method-sequence exploration
  - Parasoft Jtest 4.5 and others

- ## Specification-based state exploration
  - TestEra [Marinov&Khurshid ASE 01], Korat [Boyapati et al. ISSTA 02]

- ## Concrete-state exploration
  - Rostra [Xie et al. ASE 04], NASA JPF [Visser et al. ISSTA 04]

- ## Symbolic-state exploration
  - Symstra [Xie et al. TACAS 05]

- ## More recently: concolic-state exploration
  - DART [Godefroid et al. PLDI 05], EGT [Cadar&Engler SPIN 05], CUTE [Sen et al. FSE 05]

Questions?