Precise Lazy Initialization for Programs with Complex Heap Inputs

Juan Manuel Copia, Facundo Molina, Nazareno Aguirre, Marcelo F. Frias,
Alessandra Gorla and Pablo Ponzio

4th International KLEE Workshop on Symbolic Execution, 15–16 April 2024

SYMBOLIC EXECUTION

```
func(int x, int y) {
  if (x < 10) {
    if (y > x)
      x = x + y;
    else
      ERROR!
  }
  return x;
}
```

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: True

SYMBOLIC EXECUTION

```
func(int x, int y) {
  if (x < 10) {
    if (y > x)
      x = x + y;
    else
      ERROR!
  }
  return x;
}
```

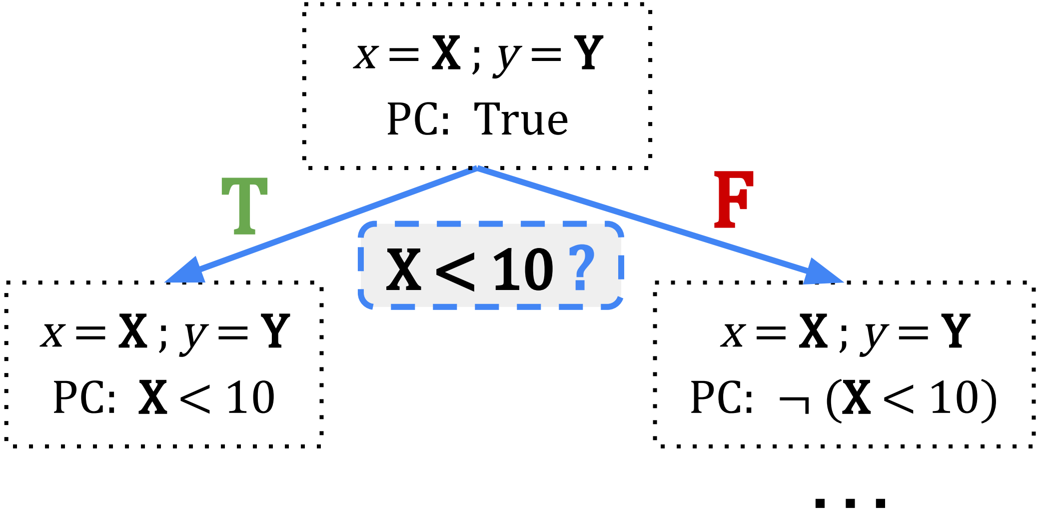
$x = \mathbf{X}\,;\ y = \mathbf{Y}$

PC: True

$\mathbf{X} < 10$?

SYMBOLIC EXECUTION

```
func(int x, int y) {
  if (x < 10) {
    if (y > x)
      x = x + y;
    else
      ERROR!
  }
  return x;
}
```



$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: True

$\mathbf{X} < 10$?

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\mathbf{X} < 10$

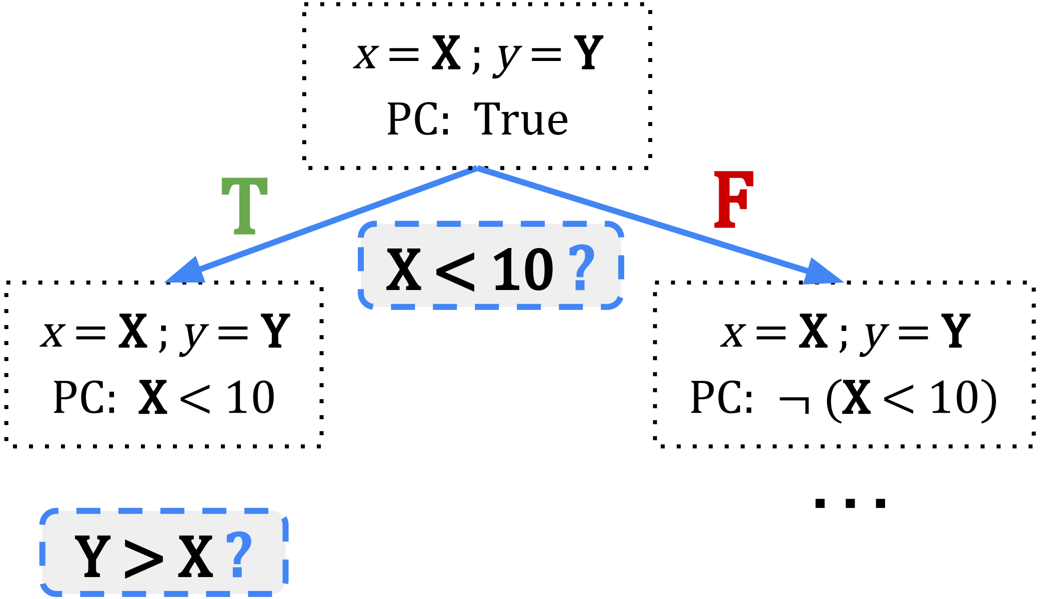
$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\neg\,(\mathbf{X} < 10)$

...

SYMBOLIC EXECUTION

```
func(int x, int y) {
  if (x < 10) {
    if (y > x)
      x = x + y;
    else
      ERROR!
  }
  return x;
}
```



$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: True

T

F

$\mathbf{X} < \mathbf{10}$?

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\mathbf{X} < 10$

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

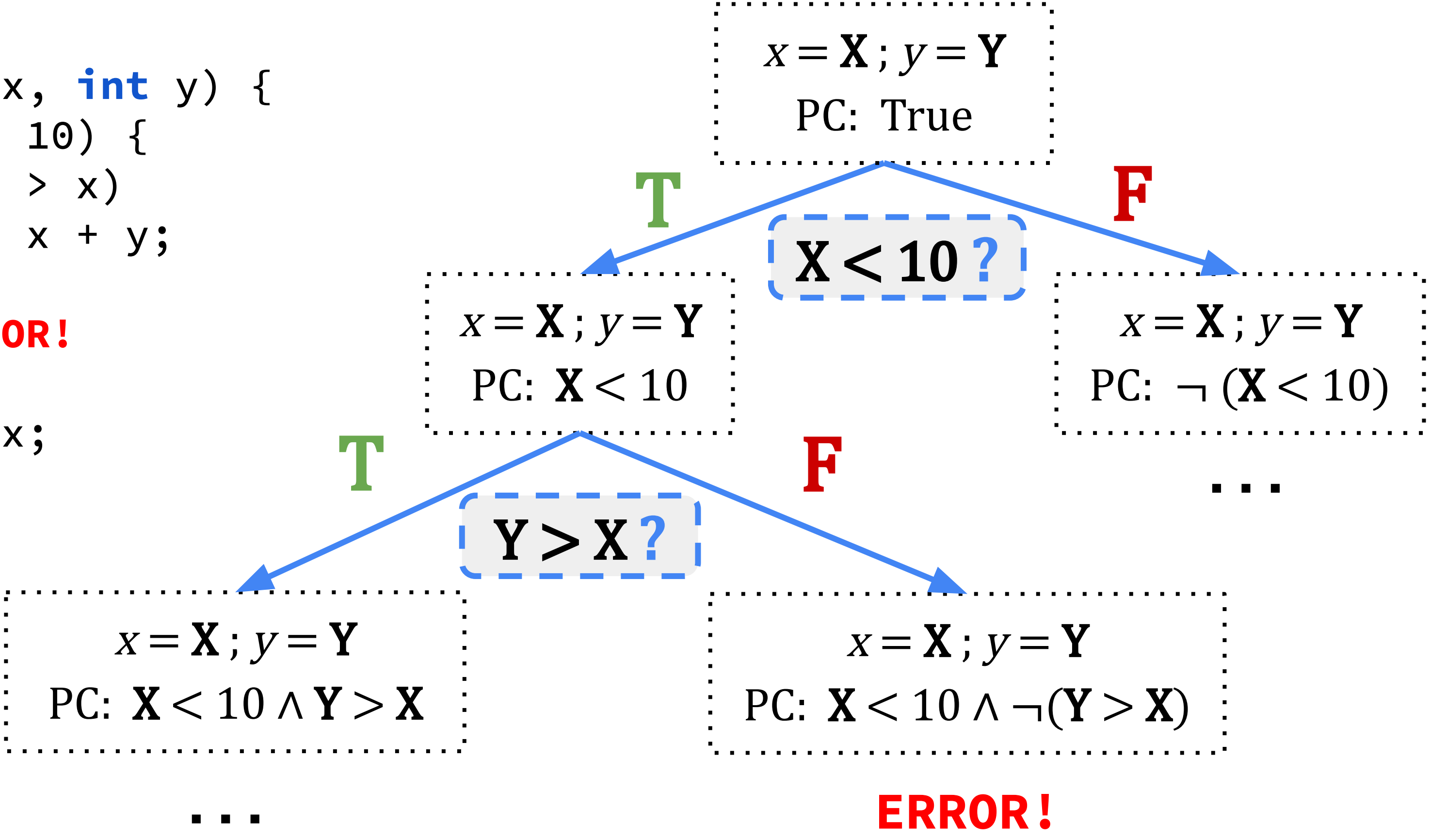
PC: $\neg\,(\mathbf{X} < 10)$

...

$\mathbf{Y} > \mathbf{X}$?

SYMBOLIC EXECUTION

```
func(int x, int y) {
  if (x < 10) {
    if (y > x)
      x = x + y;
    else
      ERROR!
  }
  return x;
}
```



$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: True

T

$\mathbf{X} < 10$?

F

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\mathbf{X} < 10$

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\neg\,(\mathbf{X} < 10)$

...

T

$\mathbf{Y} > \mathbf{X}$?

F

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\mathbf{X} < 10 \wedge \mathbf{Y} > \mathbf{X}$

$x = \mathbf{X}\,;\, y = \mathbf{Y}$

PC: $\mathbf{X} < 10 \wedge \neg(\mathbf{Y} > \mathbf{X})$

...

ERROR!

PROGRAMS WITH HEAP ALLOCATED INPUTS

PROGRAMS WITH HEAP ALLOCATED INPUTS

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

PROGRAMS WITH HEAP ALLOCATED INPUTS

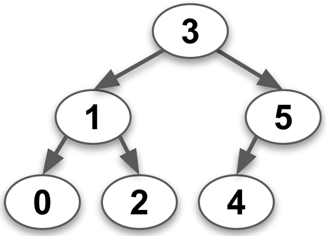
```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

- ***Precondition***:

PROGRAMS WITH HEAP ALLOCATED INPUTS

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

- ***Precondition*:**
  - **Sorted** keys

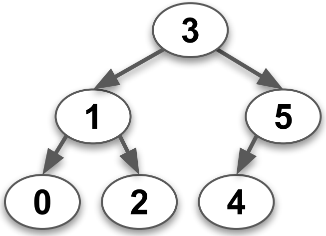


✓

PROGRAMS WITH HEAP ALLOCATED INPUTS

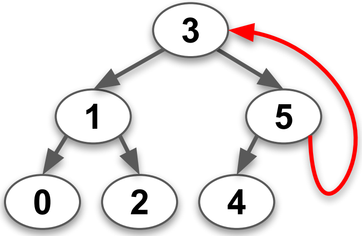
```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

- ***Precondition*****:**
  - **Sorted** keys



✓

  - No **cycles**

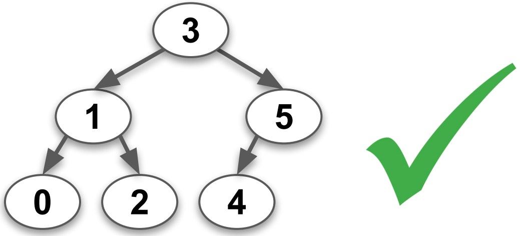


✗

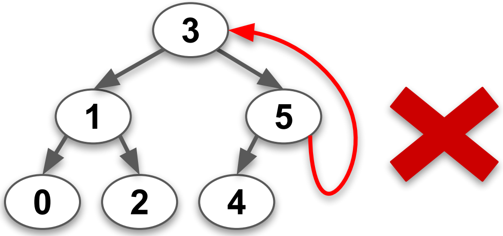
PROGRAMS WITH HEAP ALLOCATED INPUTS

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

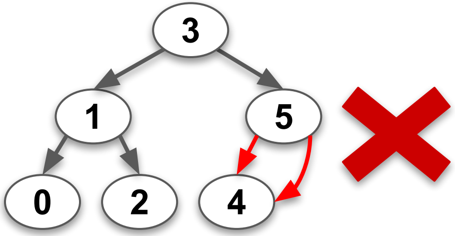
- ***Precondition***:
  - **Sorted** keys



  - No **cycles**



  - No **Node Sharing**



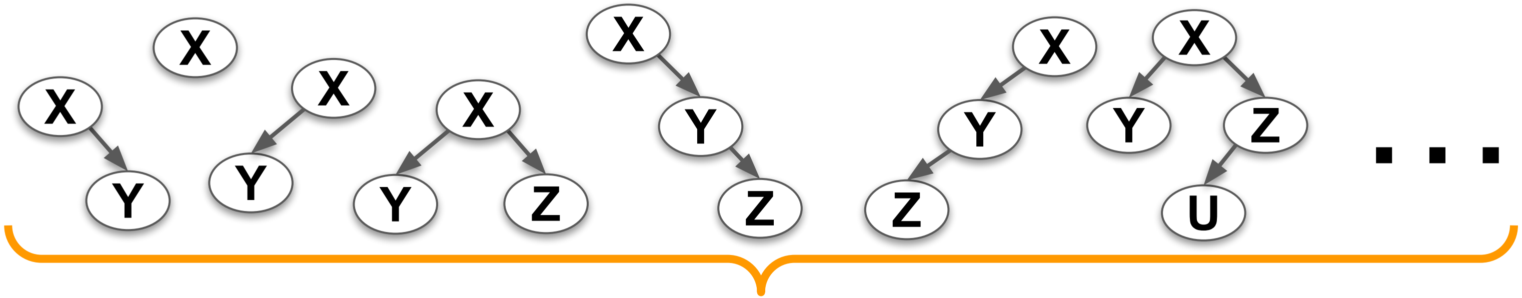
EAGER APPROACH

EAGER APPROACH

- Treats the **heap** in a **fully concrete** way and primitive types in a **symbolic way**.

EAGER APPROACH

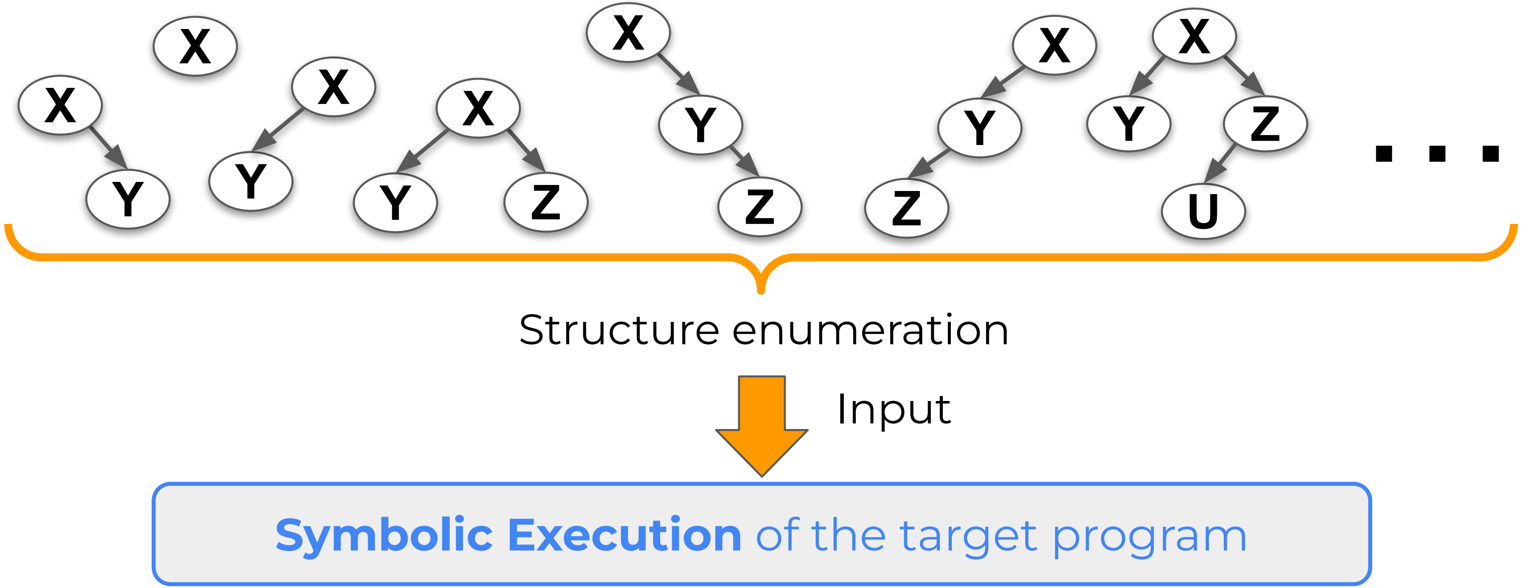
- Treats the **heap** in a **fully concrete** way and primitive types in a **symbolic way**.



Structure enumeration

EAGER APPROACH

- Treats the **heap** in a **fully concrete** way and primitive types in a **symbolic way**.



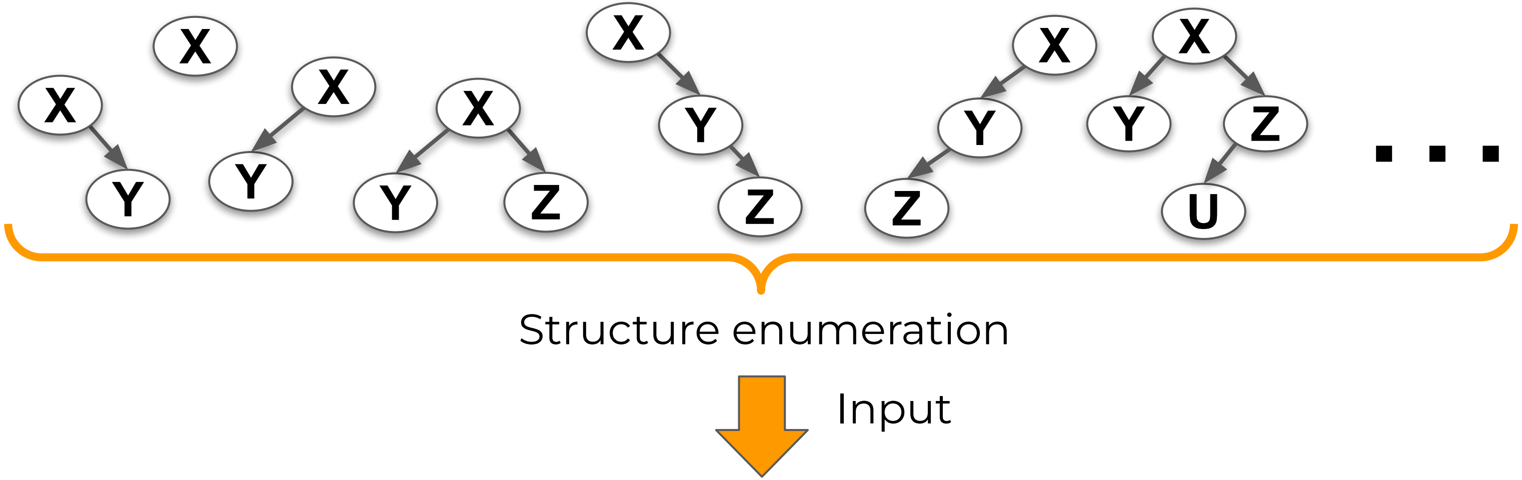
Structure enumeration

Input

Symbolic Execution of the target program

EAGER APPROACH

- Treats the **heap** in a **fully concrete** way and primitive types in a **symbolic way**.



Structure enumeration

Input

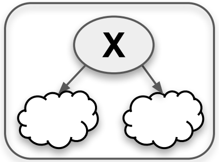
Symbolic Execution of the target program

Structure explosion problem

LAZY APPROACH

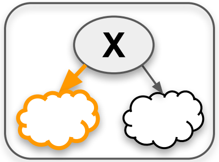
```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

LAZY APPROACH



```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

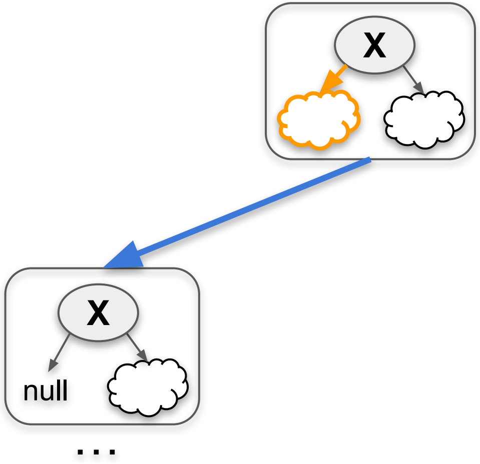
LAZY APPROACH



```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```

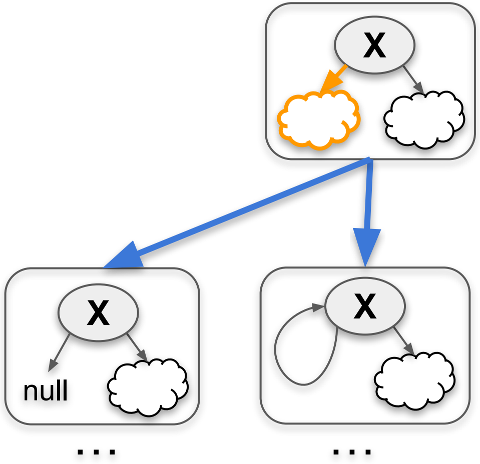
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



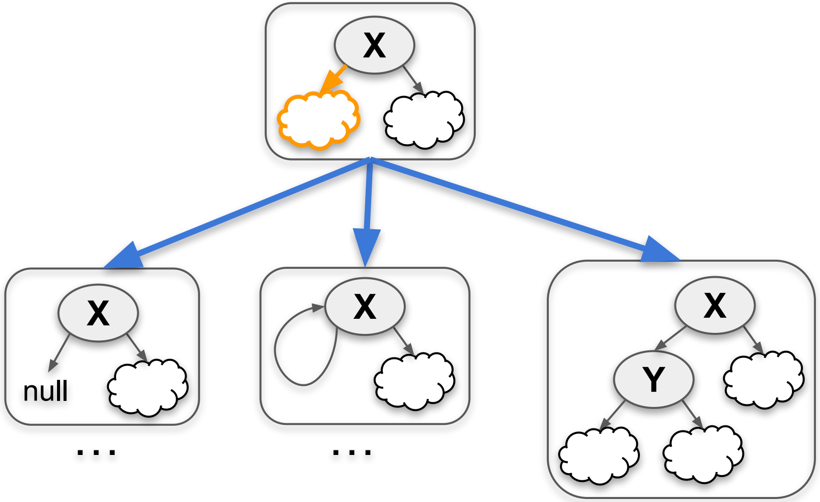
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



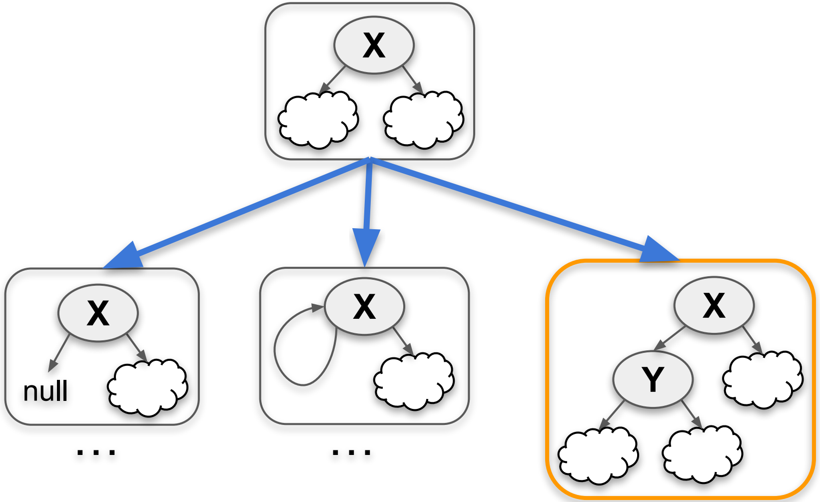
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



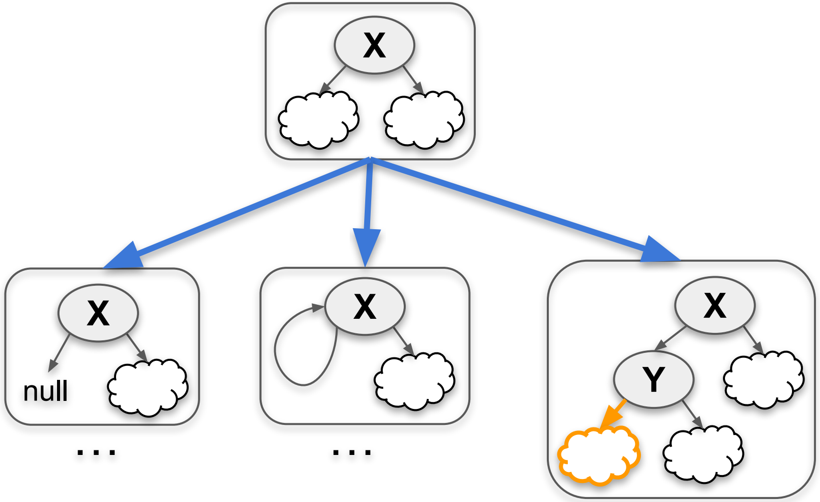
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



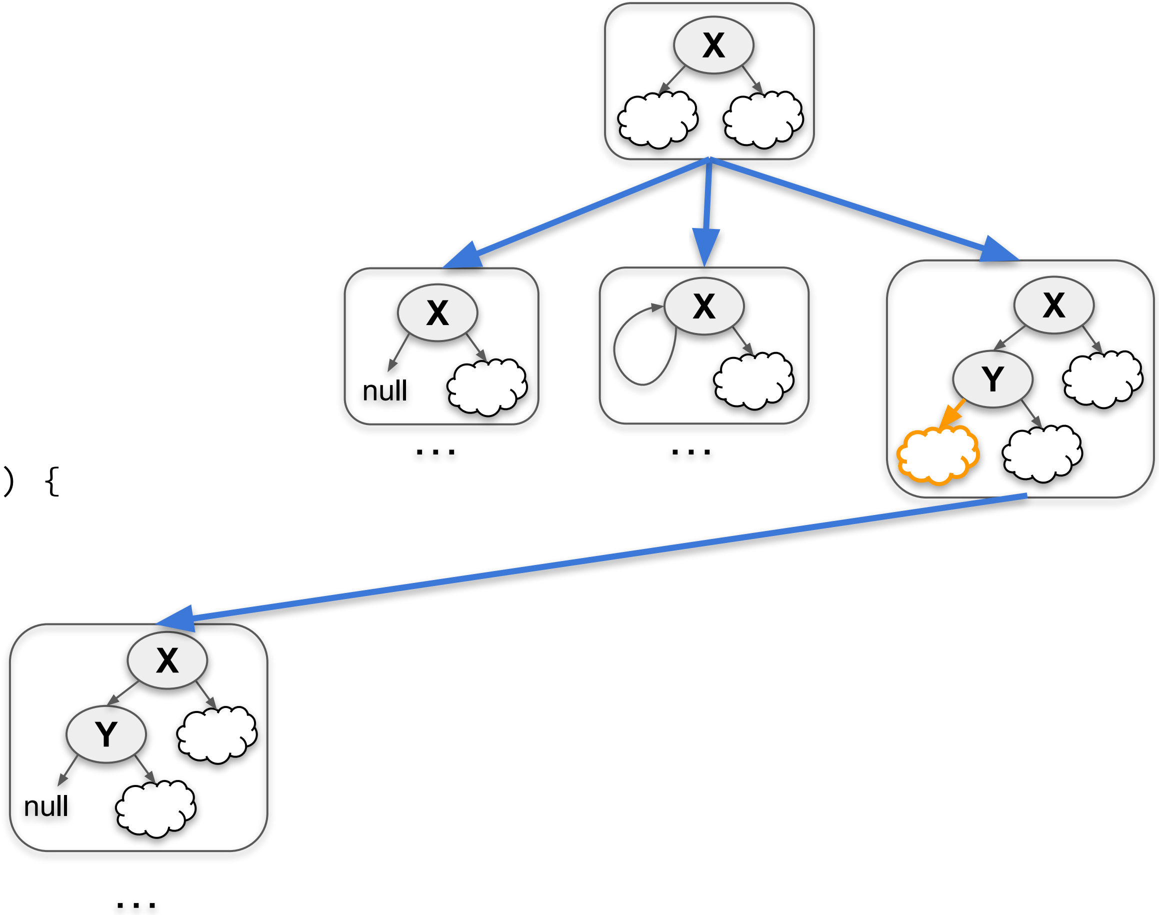
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



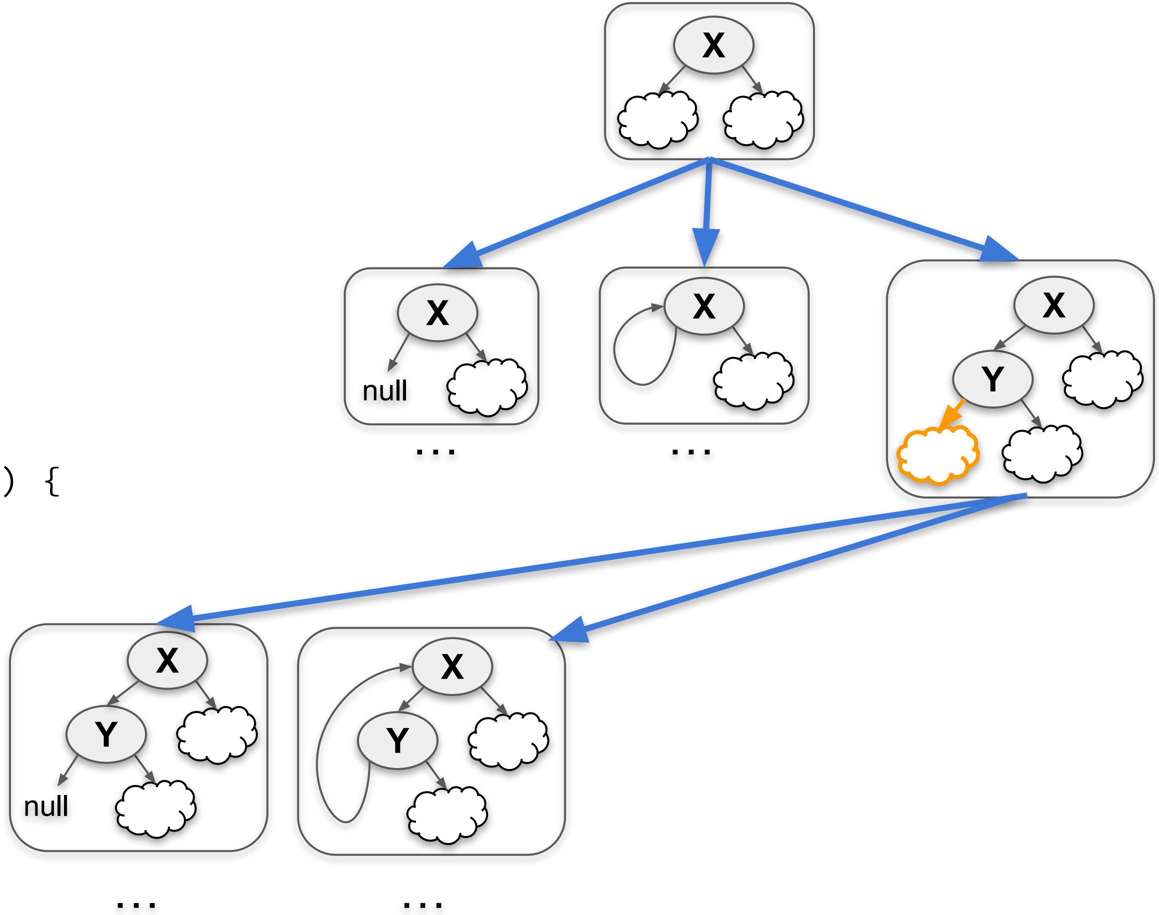
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



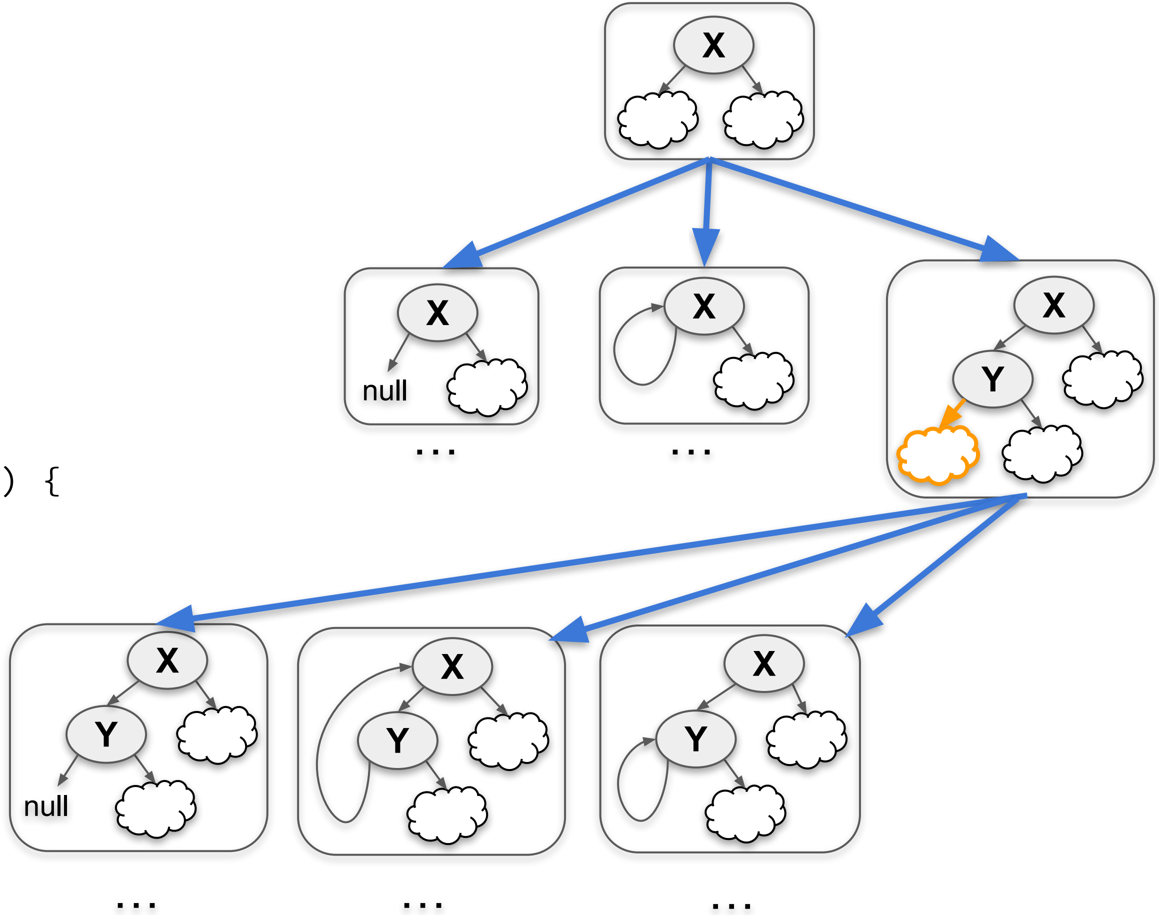
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



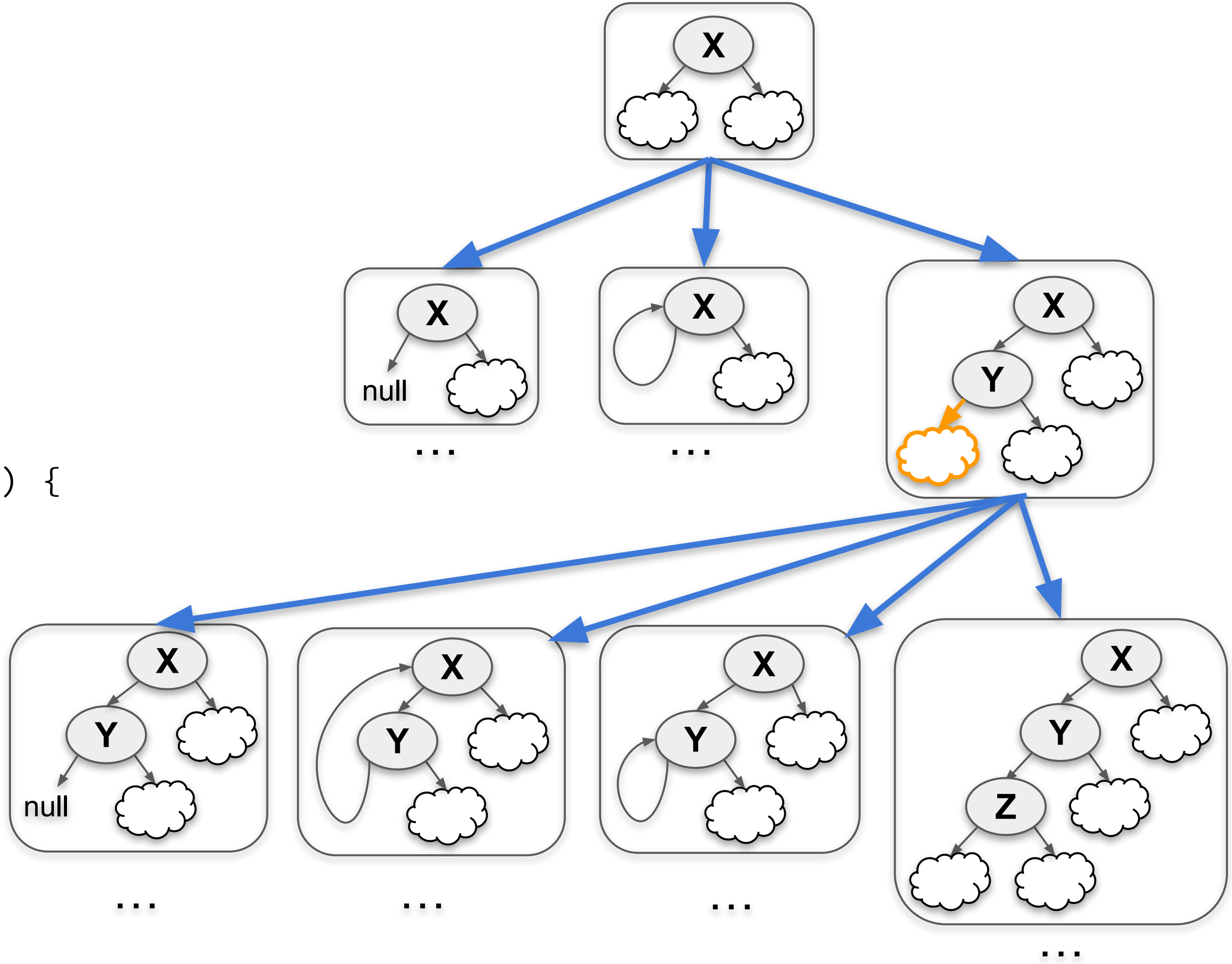
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



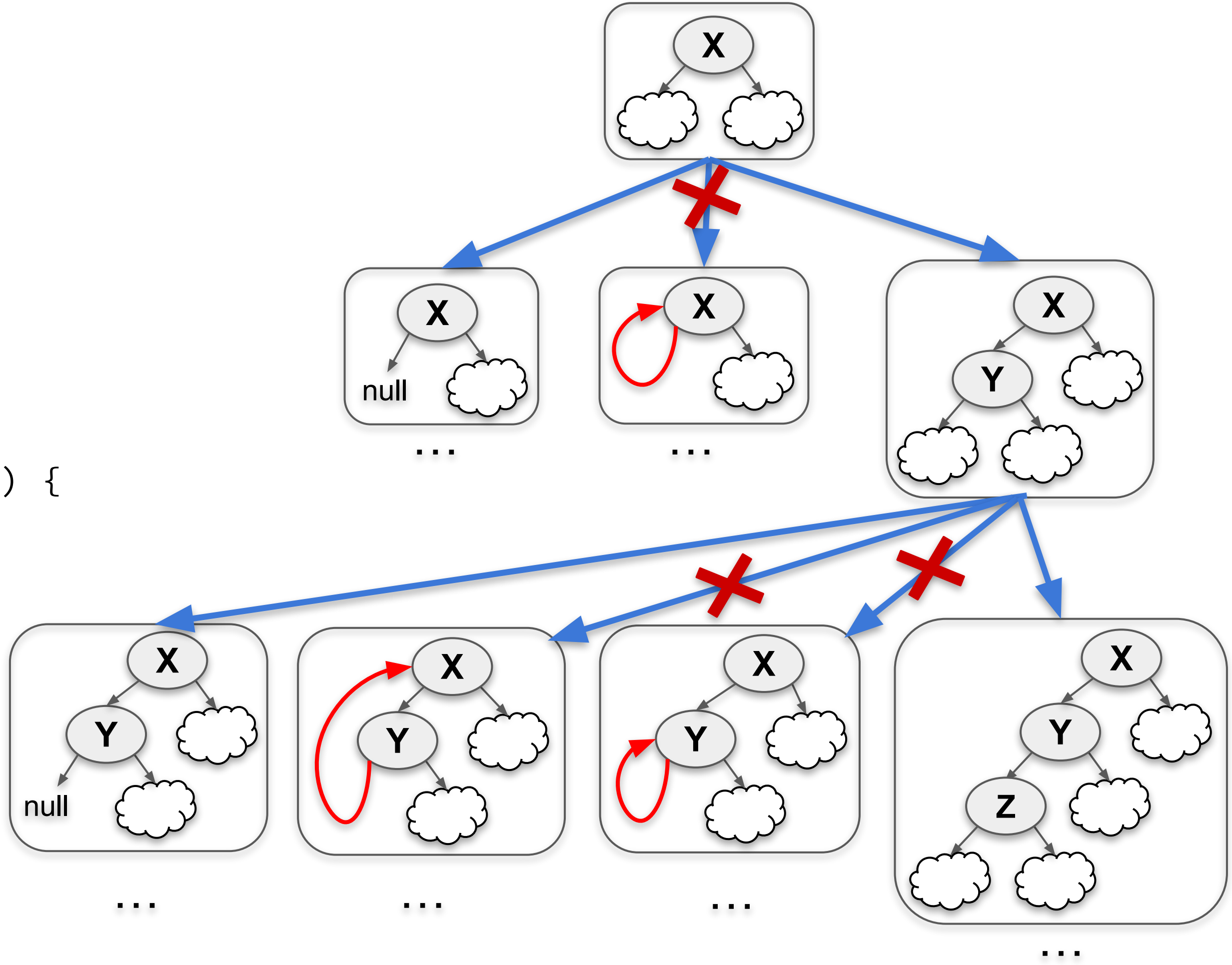
LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
      curr = curr.left;
      minKey = curr.key;
    }
    return minKey;
  }
}
```



LAZY APPROACH

```
public class BST {
  BST left;
  BST right;
  int key;

  public int getMin() {
    BST curr = this;
    int minKey = this.key
    while (curr.left != null) {
        curr = curr.left;
        minKey = curr.key;
    }
    return minKey;
  }
}
```



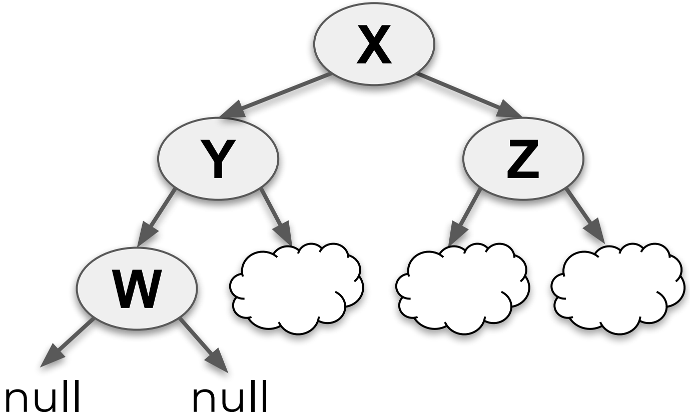
SATISFIABILITY OF SYMBOLIC STRUCTURES

SATISFIABILITY OF SYMBOLIC STRUCTURES

- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?

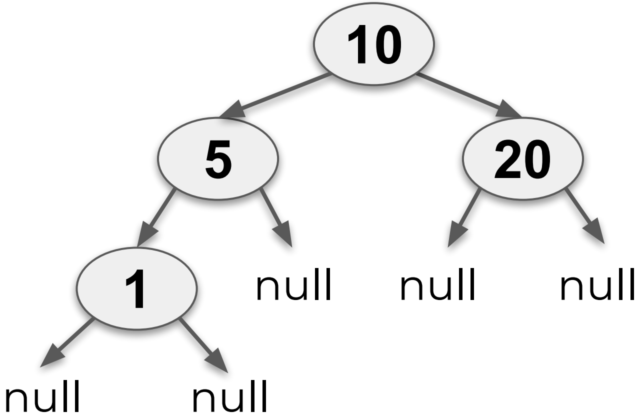
SATISFIABILITY OF SYMBOLIC STRUCTURES

- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?



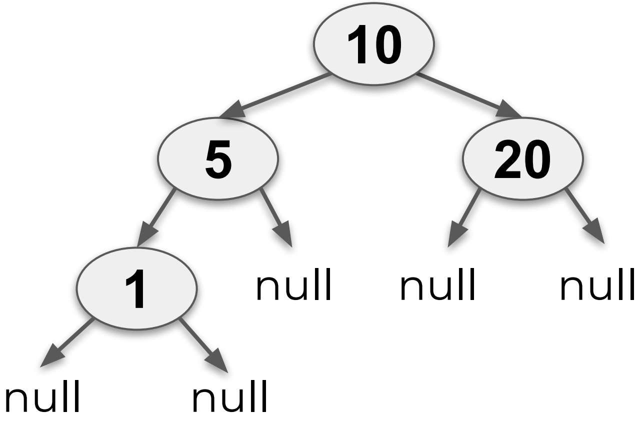
SATISFIABILITY OF SYMBOLIC STRUCTURES

- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?



SATISFIABILITY OF SYMBOLIC STRUCTURES

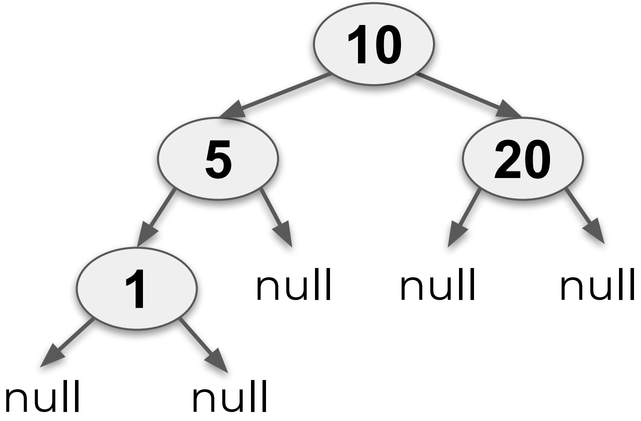
- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?



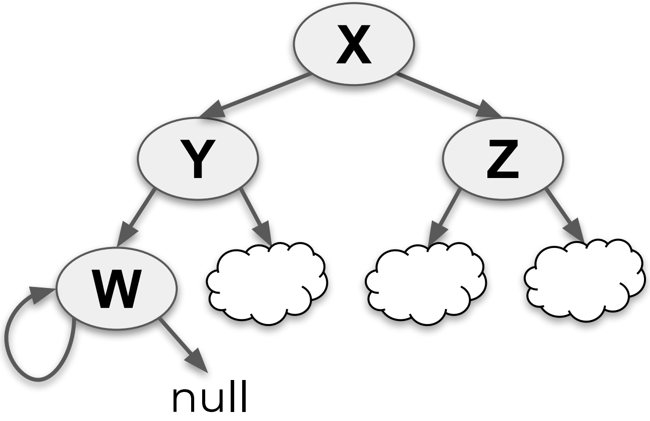
SAT

SATISFIABILITY OF SYMBOLIC STRUCTURES

- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?

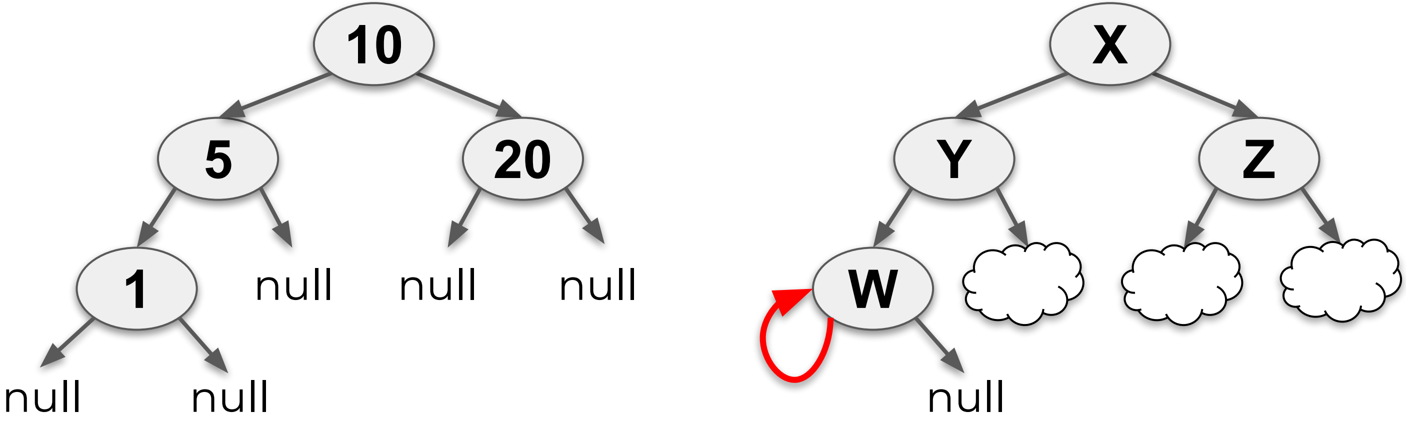


SAT



SATISFIABILITY OF SYMBOLIC STRUCTURES

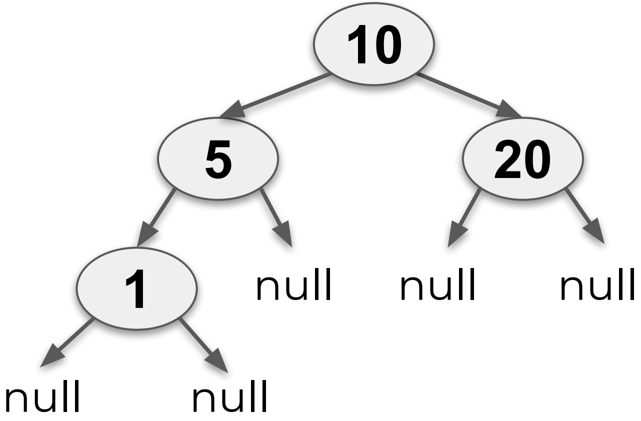
- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?



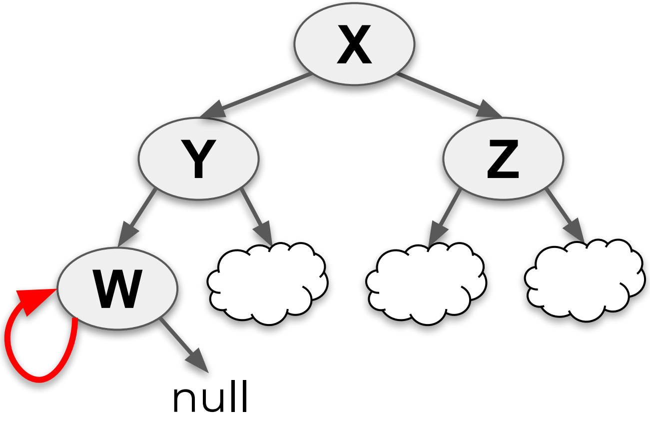
SAT

SATISFIABILITY OF SYMBOLIC STRUCTURES

- Can the symbolic structure be **extended** to a **fully concrete** structure satisfying the **specification** within the specified bounds?



SAT



UNSAT

SPECIALIZED PRECONDITIONS

SPECIALIZED PRECONDITIONS

- Prior to this work, **state-of-the-art approaches** required **specialized specifications** to identify infeasible symbolic states.

SPECIALIZED PRECONDITIONS

- Prior to this work, **state-of-the-art approaches** required **specialized specifications** to identify infeasible symbolic states.

Symbolic-aware operational specifications

```
public boolean isBinTree() {
  if (!IS_SYMBOLIC(root))
       return true;
  Set<BST> visited = new HashSet<>();
  LinkedList<BST> worklist = new LinkedList<>();
  visited.add(root);
  worklist.add(root);
  while (!worklist.isEmpty()) {
    BST node = worklist.removeFirst();
    if (!IS_SYMBOLIC(node.left)) {
      if (node.left != null && !visited.add(node.left))
        return false;
      worklist.add(node.left);
    }
    if (!IS_SYMBOLIC(node.right)) {
      if (node.right != null && !visited.add(node.right))
        return false;
      worklist.add(node.right);
    }
  }
  return true;
}
```

SPECIALIZED PRECONDITIONS

- Prior to this work, **state-of-the-art approaches** required **specialized specifications** to identify infeasible symbolic states.

Symbolic-aware operational specifications

```
public boolean isBinTree() {
  if (!IS_SYMBOLIC(root))
        return true;
  Set<BST> visited = new HashSet<>();
  LinkedList<BST> worklist = new LinkedList<>();
  visited.add(root);
  worklist.add(root);
  while (!worklist.isEmpty()) {
    BST node = worklist.removeFirst();
    if (!IS_SYMBOLIC(node.left)) {
      if (node.left != null && !visited.add(node.left))
        return false;
      worklist.add(node.left);
    }
    if (!IS_SYMBOLIC(node.right)) {
      if (node.right != null && !visited.add(node.right))
        return false;
      worklist.add(node.right);
    }
  }
  return true;
}
```

SPECIALIZED PRECONDITIONS

- Prior to this work, **state-of-the-art approaches** required **specialized specifications** to identify infeasible symbolic states.

Symbolic-aware operational specifications

```java
public boolean isBinTree() {
 if (!IS_SYMBOLIC(root))
    return true;
 Set<BST> visited = new HashSet<>();
 LinkedList<BST> worklist = new LinkedList<>();
 visited.add(root);
 worklist.add(root);
 while (!worklist.isEmpty()) {
  BST node = worklist.removeFirst();
  if (!IS_SYMBOLIC(node.left)) {
   if (node.left != null && !visited.add(node.left))
    return false;
   worklist.add(node.left);
  }
  if (!IS_SYMBOLIC(node.right)) {
   if (node.right != null && !visited.add(node.right))
    return false;
   worklist.add(node.right);
  }
 }
 return true;
}
```

Declarative Specifications

```
instanceof avl_tree/AvlTree_Any expands to instanceof avl_tree/AvlTree_HEX &&

------------------------------------------------
{R_ANY}/root(/left|/right)* instanceof avl_tree/AvlNode_HEX aliases nothing &&
{R_ANY}/root instanceof avl_tree/AvlNode_HEX expands to instanceof
avl_tree/AvlNode_HEX triggers
avl_tree/AvlNode_HEX:(Lavl_tree/AvlNode_HEX;)V:_got_AvlNode_onRoot:{$REF} &&
{R_ANY}/root(/left|/right)*/left instanceof avl_tree/AvlNode_HEX expands to
instanceof avl_tree/AvlNode_HEX triggers
avl_tree/AvlNode_HEX:(Lavl_tree/AvlNode_HEX;)V:_got_AvlNode_onTheLeft:{$REF} &&
{R_ANY}/root(/left|/right)*/right instanceof avl_tree/AvlNode_HEX expands to
instanceof avl_tree/AvlNode_HEX triggers
avl_tree/AvlNode_HEX:(Lavl_tree/AvlNode_HEX;)V:_got_AvlNode_onTheRight:{$REF} &&
{R_ANY}/root(/left|/right)*/left instanceof avl_tree/AvlNode_HEX null triggers
avl_tree/AvlNode_HEX:(Lavl_tree/AvlNode_HEX;)V:_got_null_onTheLeft:{$REF}/{UP} &&
{R_ANY}/root(/left|/right)*/right instanceof avl_tree/AvlNode_HEX null triggers
avl_tree/AvlNode_HEX:(Lavl_tree/AvlNode_HEX;)V:_got_null_onTheRight:{$REF}/{UP}
&&

{R_ANY}/root/parent instanceof avl_tree/AvlNode_HEX expands to nothing &&
{R_ANY}/root/parent instanceof avl_tree/AvlNode_HEX aliases nothing &&
{R_ANY}/root(/left|/right)+/parent instanceof avl_tree/AvlNode_HEX not null &&
{R_ANY}/root(/left|/right)+/parent instanceof avl_tree/AvlNode_HEX expands to
nothing &&
{R_ANY}/root(/left|/right)+/parent instanceof avl_tree/AvlNode_HEX aliases
{$REF}/{UP}/{UP}
```

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros		
Cons		

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don’t** necessarily require a **specification**.	
Cons		

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don’t** necessarily require a **specification**.	
Cons	- Structure Explosion.	

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don't** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	• **Don't** necessarily require a **specification**.	• Avoid structure explosion in many cases.
Cons	• Structure Explosion.	• Require **specialized** specifications to prune **infeasible paths**.

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don't** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	- Require **specialized** specifications to prune **infeasible paths**. - Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don't** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	- Require **specialized** specifications to prune **infeasible paths**. - Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

LISSA

LISSA

- LISSA [1] employed a **bounded exhaustive solver** (SymSolve) to decide satisfiability of the **heap constraints.**

[1] Copia et al. **LISSA: Lazy Initialization with Specialized Solver Aid.** ASE 2022

LISSA

- LISSA [1] employed a **bounded exhaustive solver** (SymSolve) to decide satisfiability of the **heap constraints.**
- Supports traditional **concrete operational** specifications (repOKs).

---

[1] Copia et al. **LISSA: Lazy Initialization with Specialized Solver Aid.** ASE 2022

LISSA

- LISSA [1] employed a **bounded exhaustive solver** (SymSolve) to decide satisfiability of the **heap constraints.**
- Supports traditional **concrete operational** specifications (repOKs).

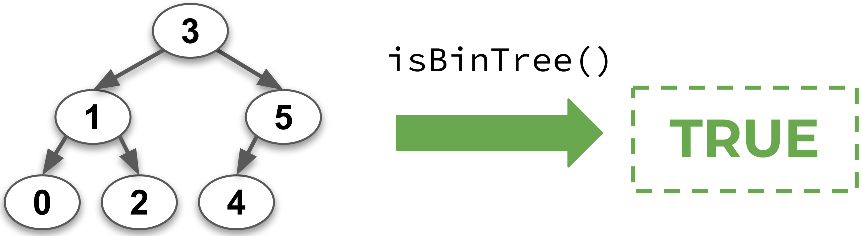
```java
public boolean isBinTree() {
  Set<BST> visited = new HashSet<>();
  LinkedList<BST> worklist = new LinkedList<>();
  visited.add(root);
  worklist.add(root);
  while (!worklist.isEmpty()) {
    BST node = worklist.removeFirst();
    if (node.left != null) {
      if (!visited.add(node.left))
        return false;
      worklist.add(node.left);
    }
    if (node.right != null) {
      if (!visited.add(node.right))
        return false;
      worklist.add(node.right);
    }
  }
  return true;
}
```

[1] Copia et al. **LISSA: Lazy Initialization with Specialized Solver Aid.** ASE 2022

LISSA

- LISSA [1] employed a **bounded exhaustive solver** (SymSolve) to decide satisfiability of the **heap constraints.**
- Supports traditional **concrete operational** specifications (repOKs).

```java
public boolean isBinTree() {
  Set<BST> visited = new HashSet<>();
  LinkedList<BST> worklist = new LinkedList<>();
  visited.add(root);
  worklist.add(root);
  while (!worklist.isEmpty()) {
    BST node = worklist.removeFirst();
    if (node.left != null) {
      if (!visited.add(node.left))
        return false;
      worklist.add(node.left);
    }
    if (node.right != null) {
      if (!visited.add(node.right))
        return false;
      worklist.add(node.right);
    }
  }
  return true;
}
```

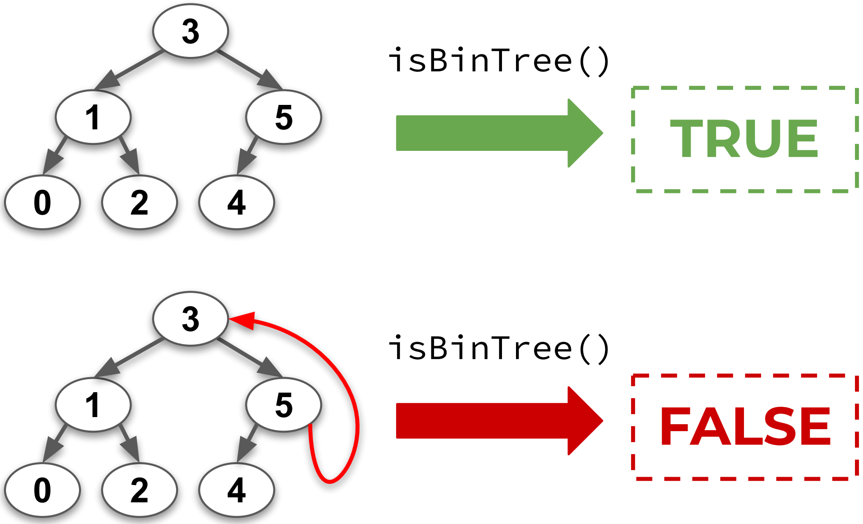


[1] Copia et al. **LISSA: Lazy Initialization with Specialized Solver Aid.** ASE 2022

LISSA

- LISSA [1] employed a **bounded exhaustive solver** (SymSolve) to decide satisfiability of the **heap constraints.**
- Supports traditional **concrete operational** specifications (repOKs).

```
public boolean isBinTree() {
  Set<BST> visited = new HashSet<>();
  LinkedList<BST> worklist = new LinkedList<>();
  visited.add(root);
  worklist.add(root);
  while (!worklist.isEmpty()) {
    BST node = worklist.removeFirst();
    if (node.left != null) {
      if (!visited.add(node.left))
        return false;
      worklist.add(node.left);
    }
    if (node.right != null) {
      if (!visited.add(node.right))
        return false;
      worklist.add(node.right);
    }
  }
  return true;
}
```



`isBinTree()`

TRUE

`isBinTree()`

FALSE

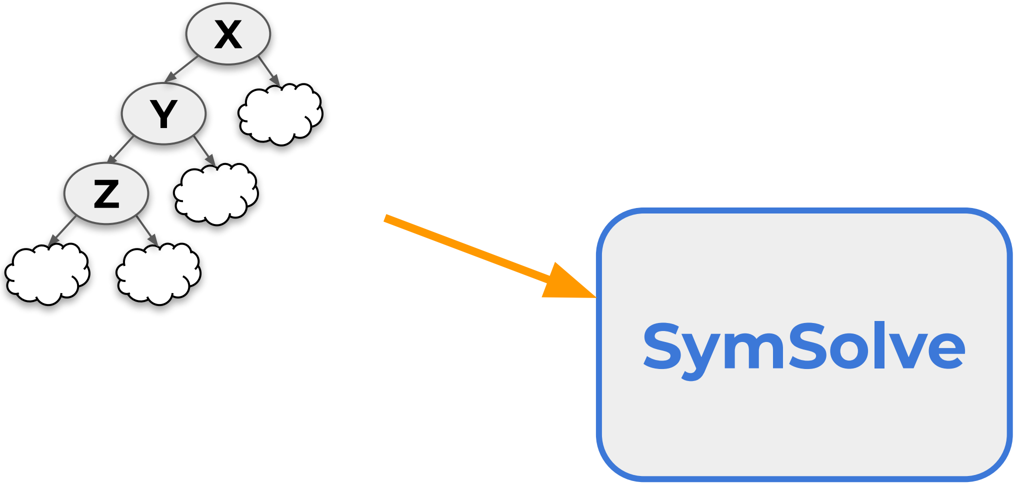
[1] Copia et al. **LISSA: Lazy Initialization with Specialized Solver Aid.** ASE 2022

SYMSOLVE

SYMSOLVE

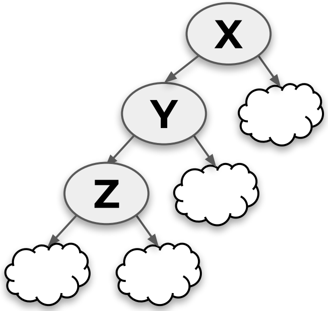
SymSolve

SYMSOLVE



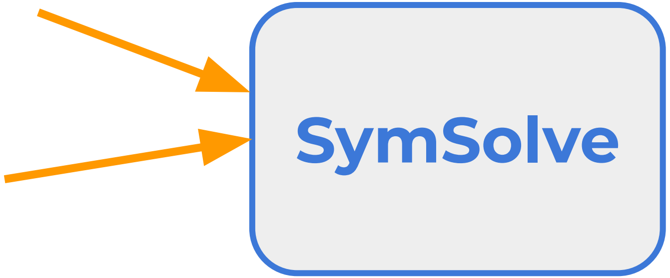
SymSolve

SYMSOLVE



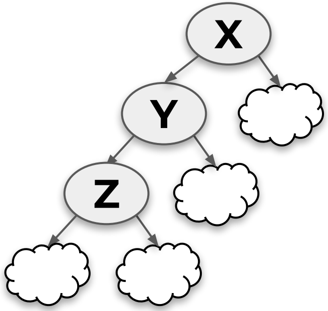
Scopes:

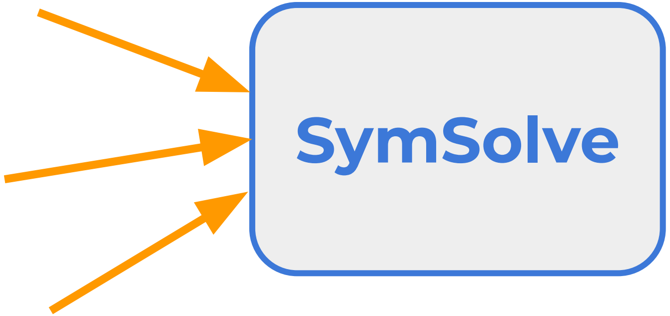
e.g. 5 Nodes.



SymSolve

SYMSOLVE





Scopes:

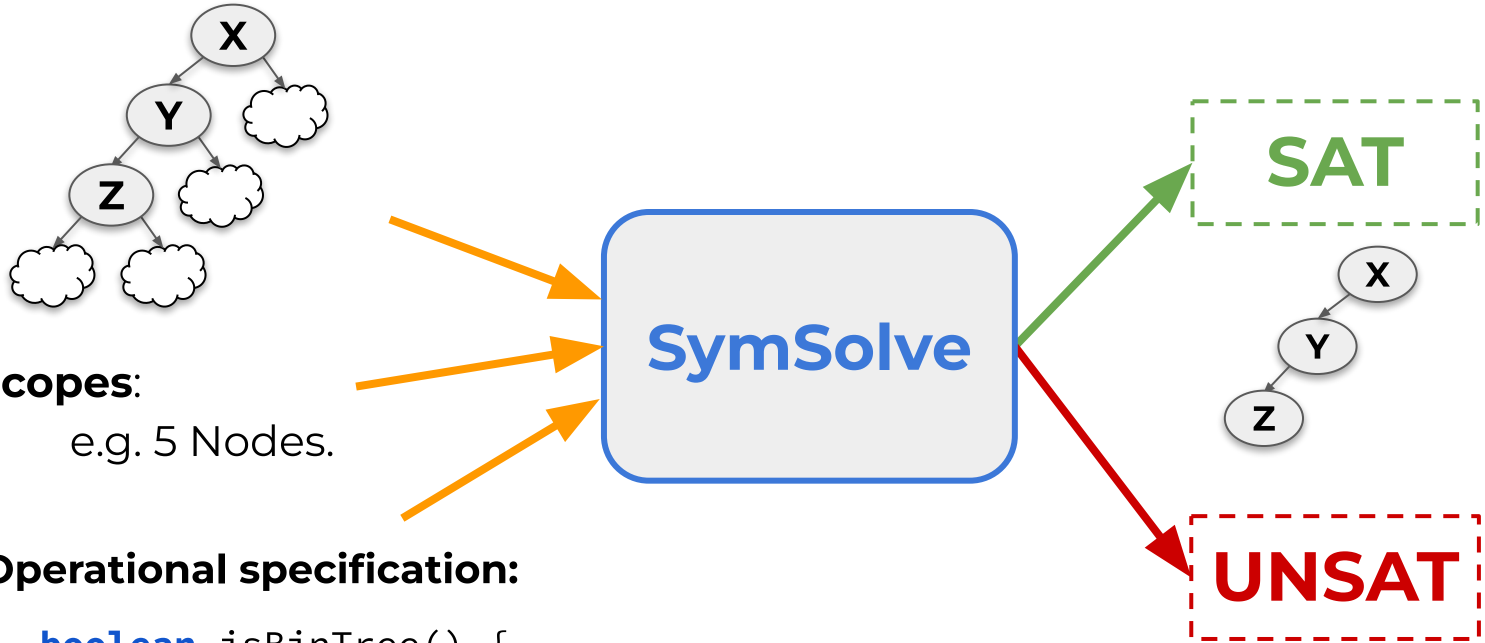
e.g. 5 Nodes.

Operational specification:

```
boolean isBinTree() {
  ...
}
```

SymSolve

SYMSOLVE



Scopes:

e.g. 5 Nodes.

Operational specification:

```
boolean isBinTree() {
  ...
}
```

SymSolve

SAT

UNSAT

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	• **Don't** necessarily require a **specification**.	• Avoid structure explosion in many cases.
Cons	• Structure Explosion.	• Require **specialized** specifications to prune **infeasible paths**. • Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don’t** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	- Require **traditional** specifications to prune **infeasible paths**. - Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don't** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	- Require **traditional** specifications to prune **infeasible paths**. - Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

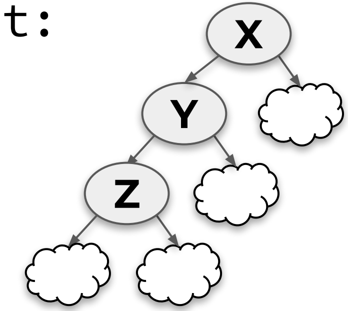
APPROACHES OVERVIEW

	Eager	**Lazy**
Pros	- **Don't** necessarily require a **specification**.	- Avoid structure explosion in many cases.
Cons	- Structure Explosion.	- Require **traditional** specifications to prune **infeasible paths**. - Existing approaches do not reason about the **path condition** and thus, can have **false alarms**.

PATH CONDITION AND HEAP SEPARATION PROBLEM

PATH CONDITION AND HEAP SEPARATION PROBLEM

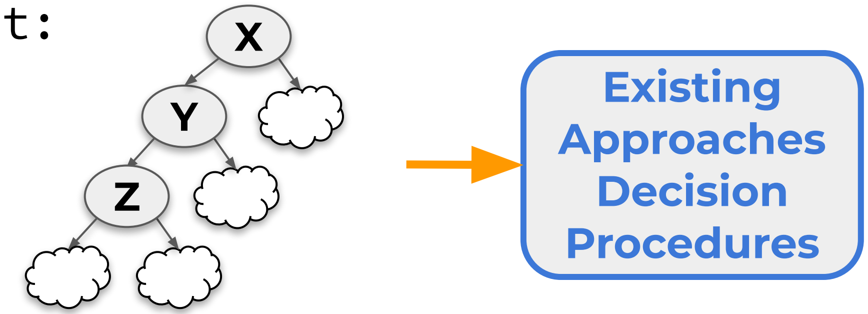
`t:`



PC: `[t.size == 2]`

PATH CONDITION AND HEAP SEPARATION PROBLEM

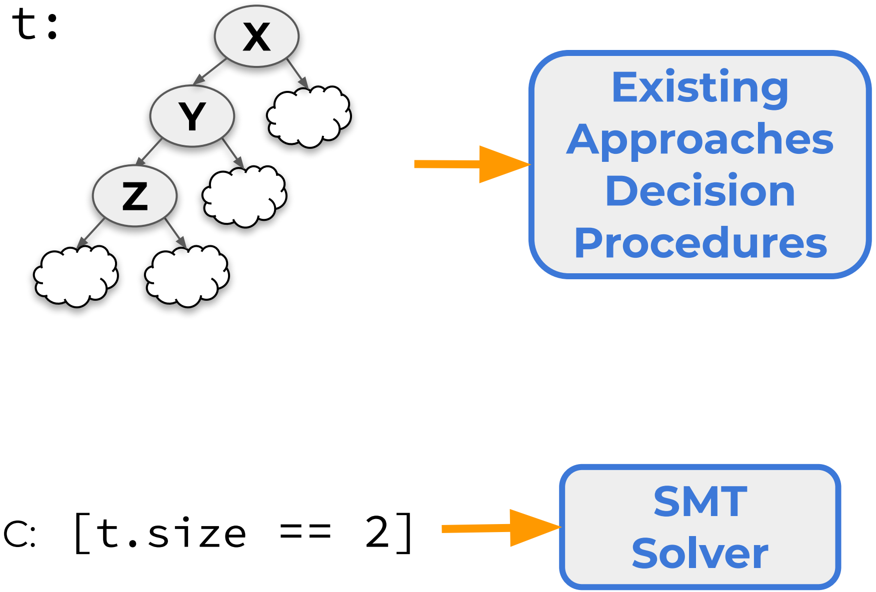
`t:`



Existing Approaches Decision Procedures

PC: `[t.size == 2]`

PATH CONDITION AND HEAP SEPARATION PROBLEM



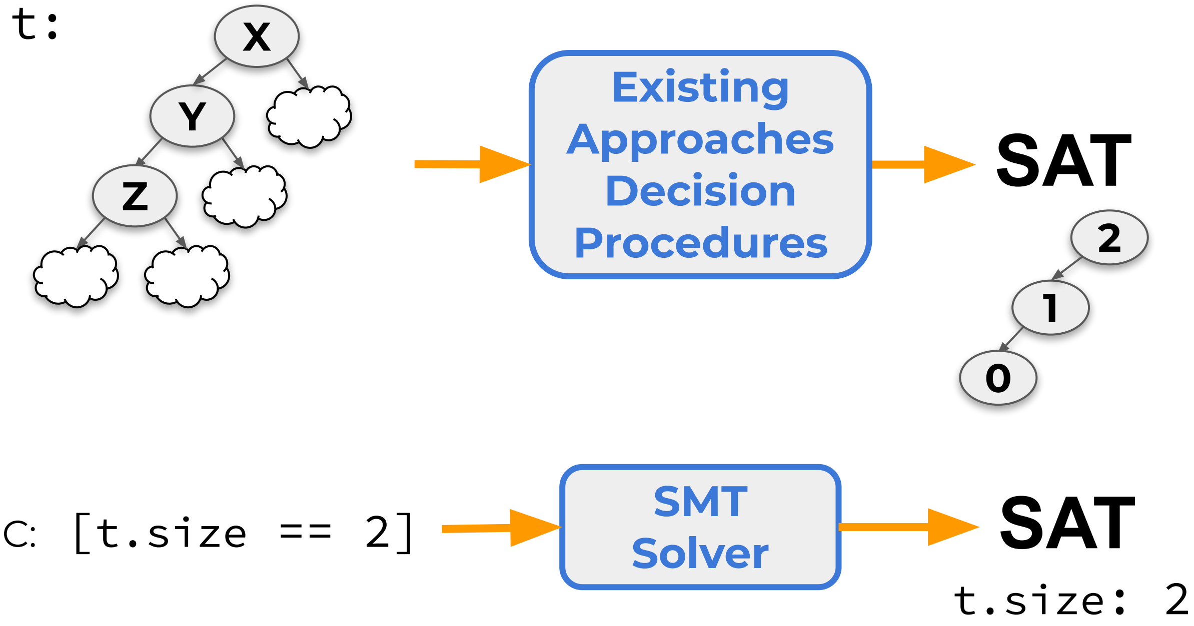
`t:`

Existing Approaches Decision Procedures

PC: `[t.size == 2]`

SMT Solver

PATH CONDITION AND HEAP SEPARATION PROBLEM



`t:`

Existing Approaches Decision Procedures

SAT

PC: `[t.size == 2]`

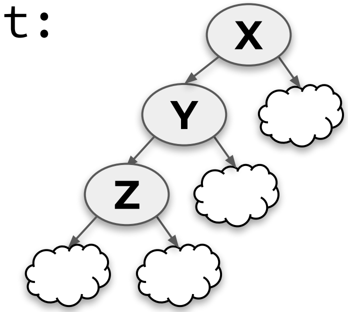
SMT Solver

SAT

`t.size: 2`

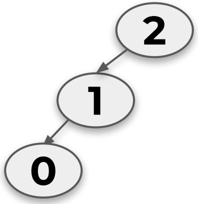
PATH CONDITION AND HEAP SEPARATION PROBLEM

`t:`



Existing Approaches Decision Procedures

SAT



SAT

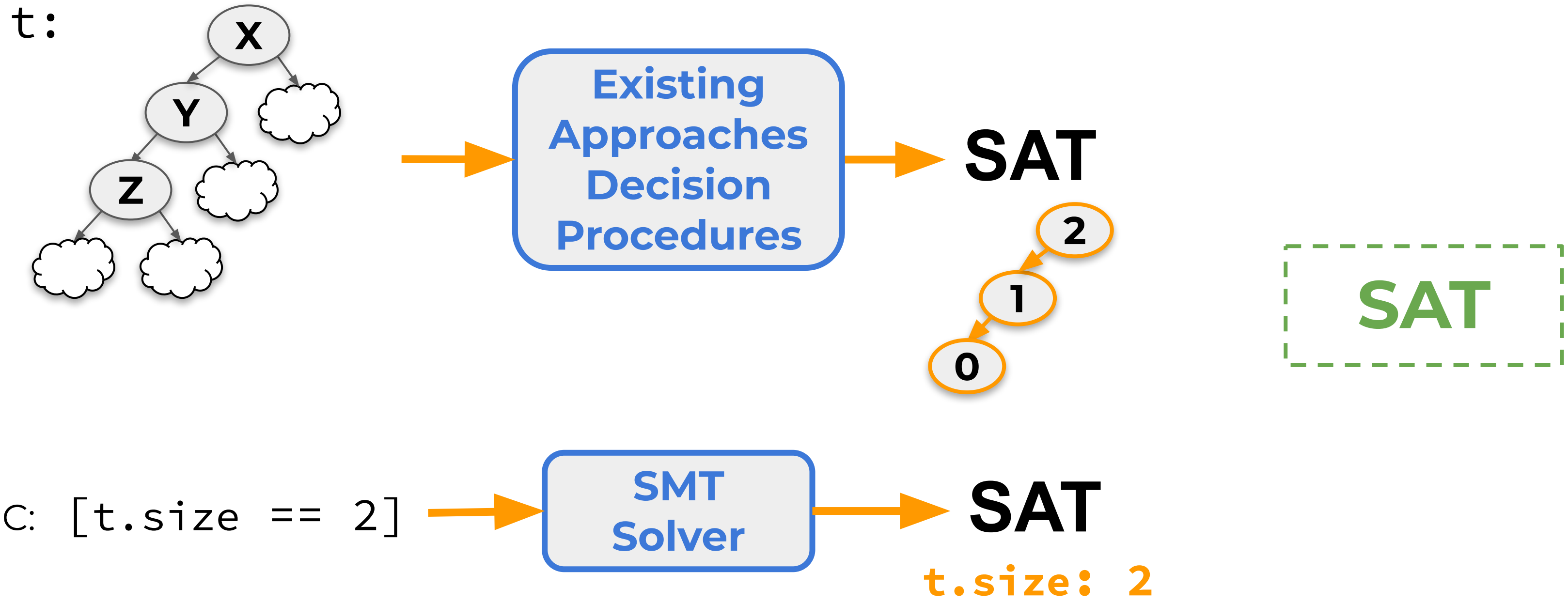
PC: `[t.size == 2]`

SMT Solver

SAT

`t.size: 2`

PATH CONDITION AND HEAP SEPARATION PROBLEM



`t:`

Existing Approaches Decision Procedures

SAT

SAT

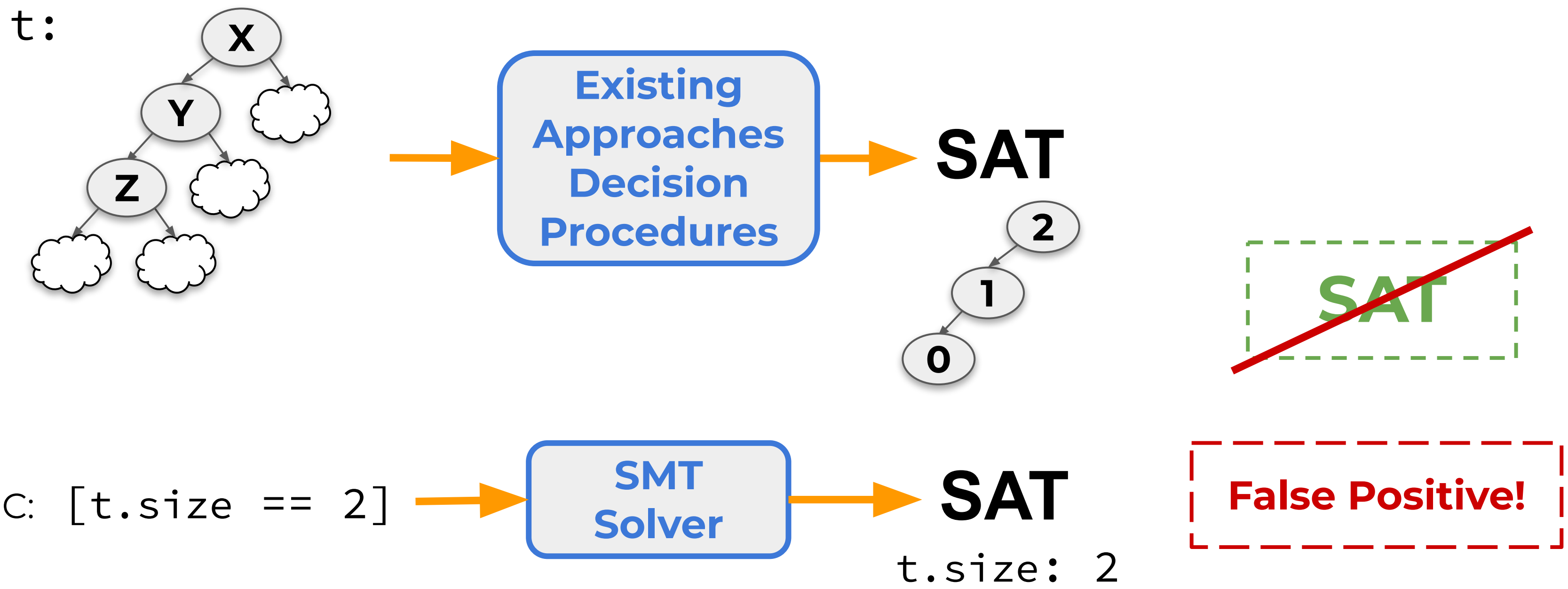
PC: `[t.size == 2]`

SMT Solver

SAT

`t.size: 2`

PATH CONDITION AND HEAP SEPARATION PROBLEM



`t:`

Existing Approaches Decision Procedures

SAT

~~SAT~~

PC: `[t.size == 2]`

SMT Solver

SAT

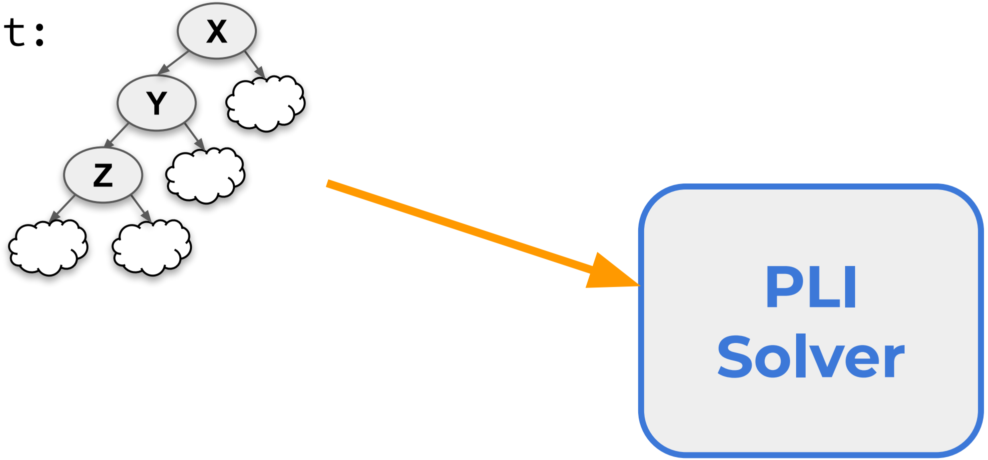
`t.size: 2`

False Positive!

PLI: PRECISE LAZY INITIALIZATION

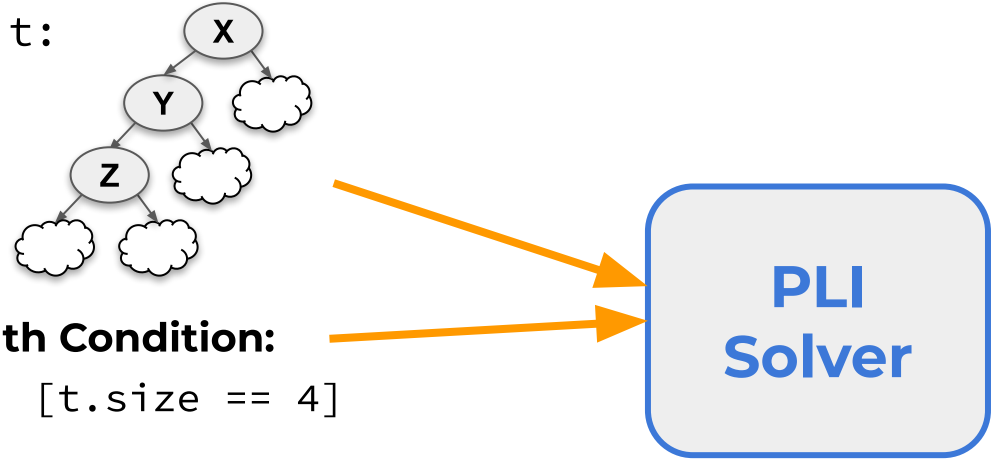
PLI Solver

PLI: PRECISE LAZY INITIALIZATION



PLI Solver

PLI: PRECISE LAZY INITIALIZATION



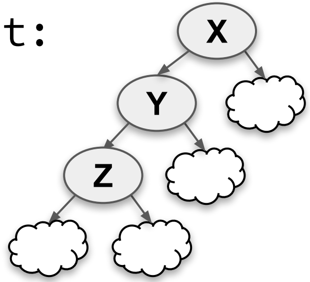
Path Condition:

`[t.size == 4]`

PLI Solver

PLI: PRECISE LAZY INITIALIZATION

`t:`

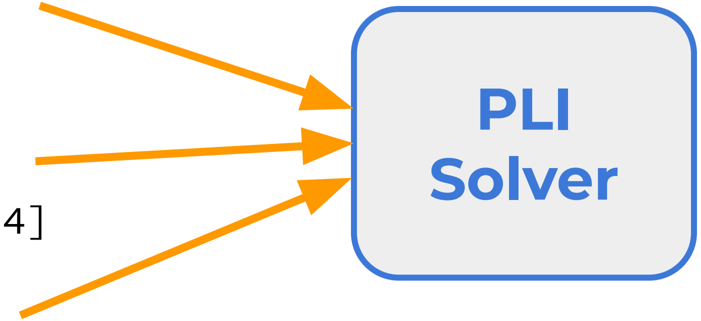


Path Condition:

`[t.size == 4]`

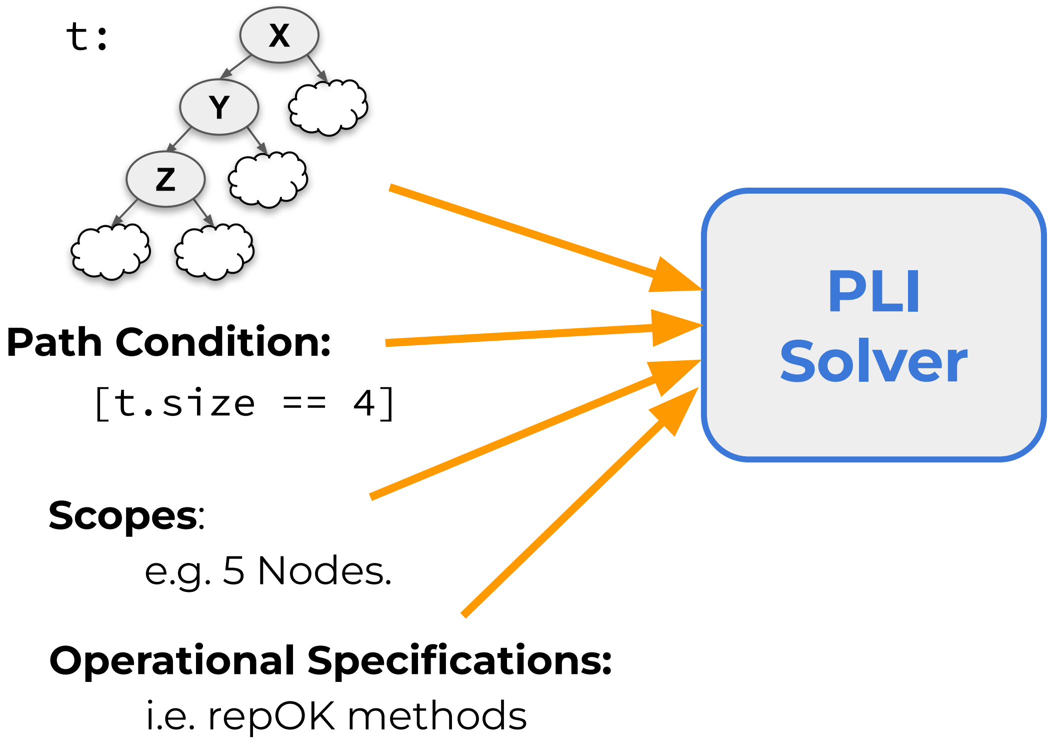
Scopes:

e.g. 5 Nodes.



PLI Solver

PLI: PRECISE LAZY INITIALIZATION



Path Condition:

`[t.size == 4]`

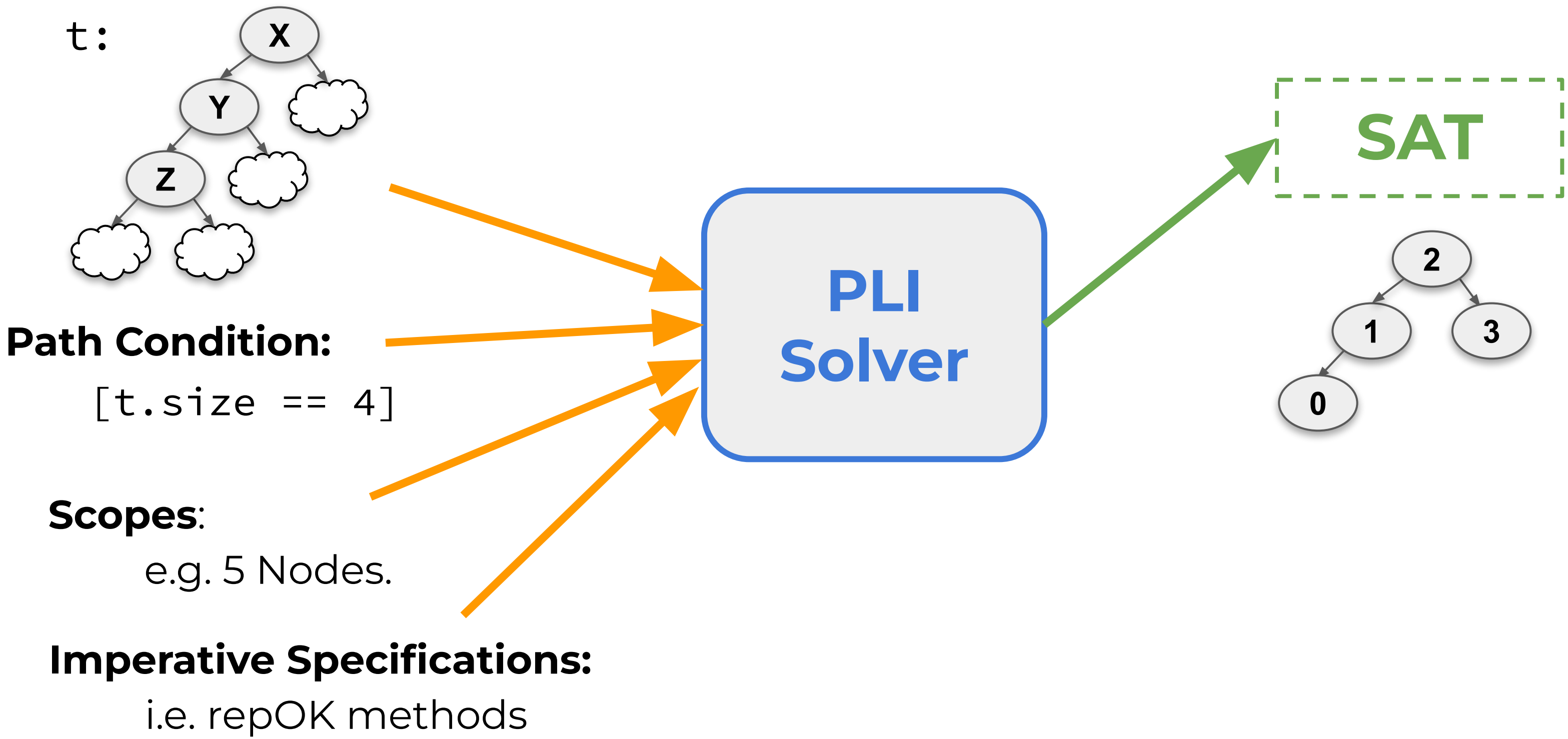
Scopes:

e.g. 5 Nodes.

Operational Specifications:

i.e. repOK methods

PLI: PRECISE LAZY INITIALIZATION



`t:`

Path Condition:

`[t.size == 4]`

Scopes:

e.g. 5 Nodes.

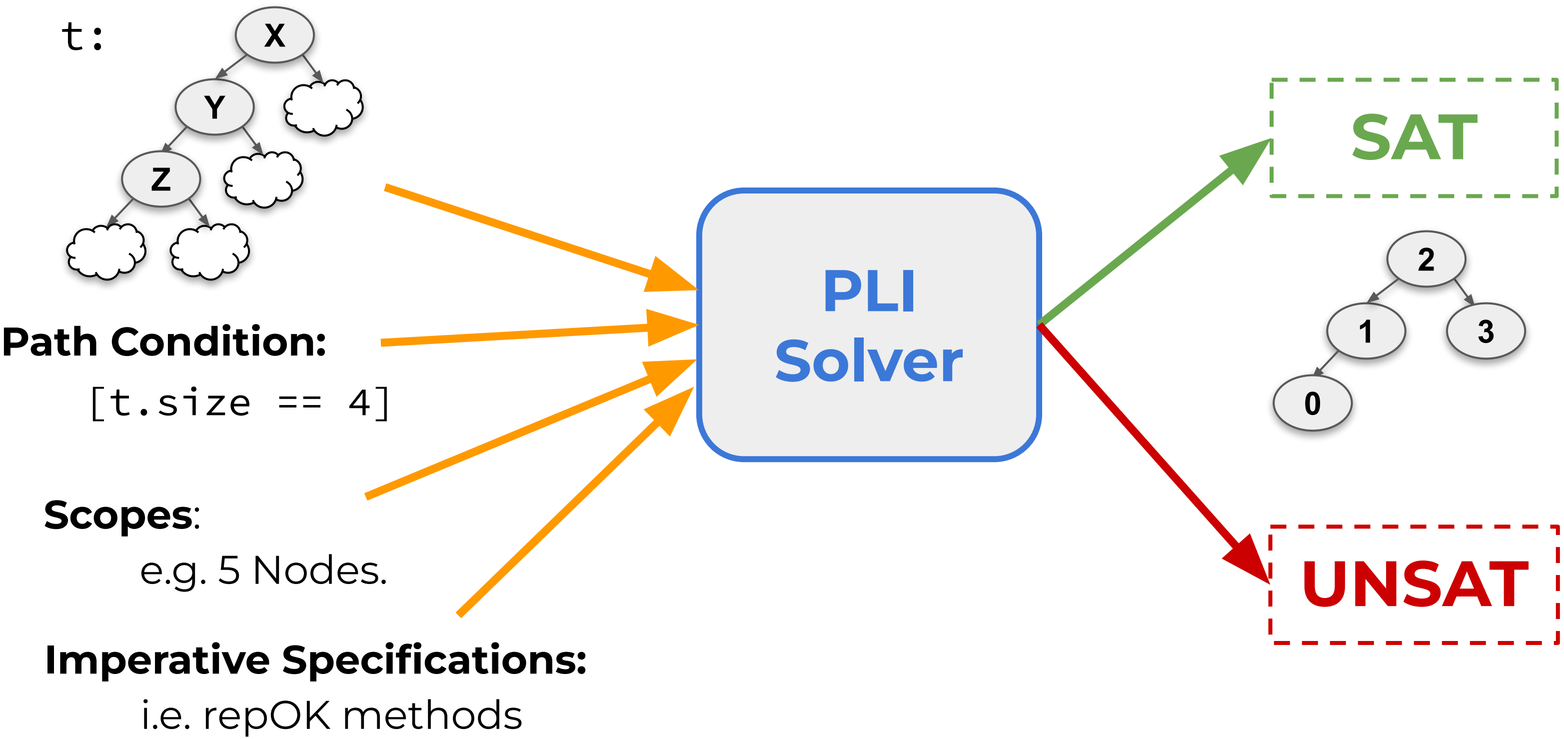
Imperative Specifications:

i.e. repOK methods

PLI Solver

SAT

PLI: PRECISE LAZY INITIALIZATION



Path Condition:

`[t.size == 4]`

Scopes:

e.g. 5 Nodes.

Imperative Specifications:

i.e. repOK methods

PLI Solver

SAT

UNSAT

PLI: PRECISE LAZY INITIALIZATION

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

```
preH() && preP()
```

PLI: PRECISE LAZY INITIALIZATION

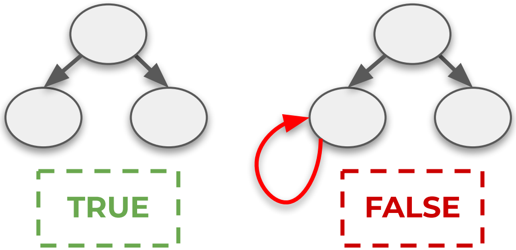
- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

```
preH() && preP()
```

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

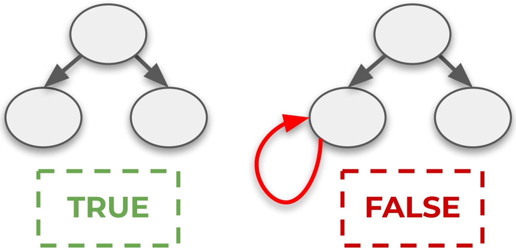
```
preH() && preP()
```



PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

`preH() && preP()`



PLI: PRECISE LAZY INITIALIZATION

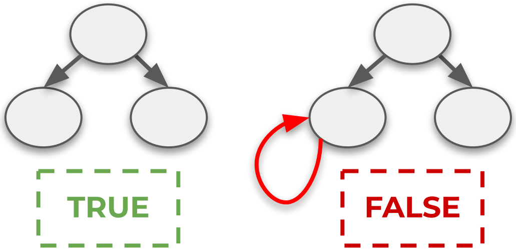
- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

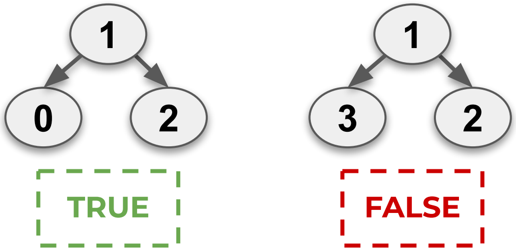
`preH() && preP()`

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

```
preH() && preP()
```





- The PLI Solver combines two solving mechanism:

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

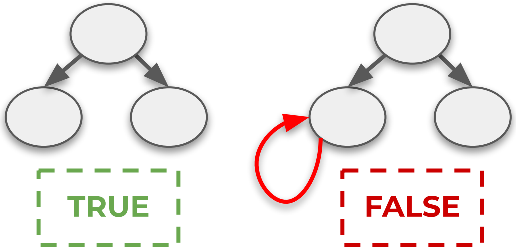
```
preH() && preP()
```

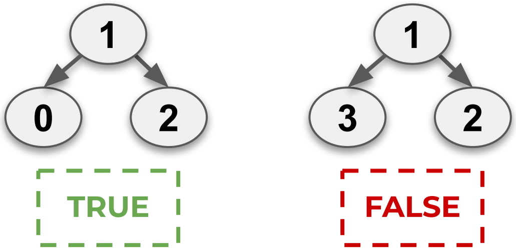
- The PLI Solver combines two solving mechanism:
    - A **heap constraint solver** (SymSolve) to solve **preH** constraints.

PLI: PRECISE LAZY INITIALIZATION

- The specification of the **program precondition** must be provided as a **conjunction** of two specifications:

```
preH() && preP()
```



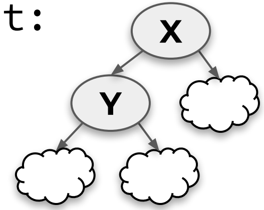


- The PLI Solver combines two solving mechanism:
  - A **heap constraint solver** (SymSolve) to solve **preH** constraints.
  - **Symbolic execution** (SMT solving) to solve **preP** constraints.

THE PLI SOLVER

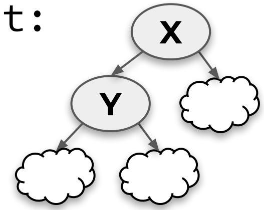
THE PLI SOLVER

`t:`



THE PLI SOLVER

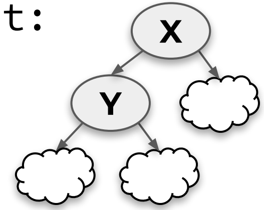
`t:`



Scope: 5 Nodes

THE PLI SOLVER

`t:`

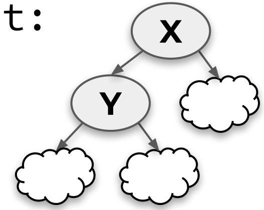


Scope: 5 Nodes

PC: `[t.size == 3]`

THE PLI SOLVER

`t:`

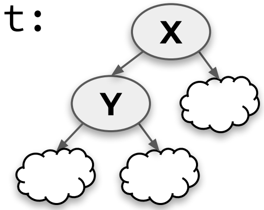


Scope: 5 Nodes

PC: `[t.size == 3]`

preH: `isBinTree()`

THE PLI SOLVER



Scope: 5 Nodes

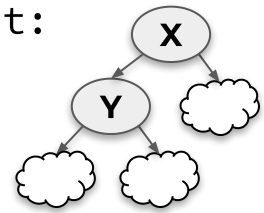
PC: `[t.size == 3]`

preH: `isBinTree()`

preP: `isSorted() && sizeOK()`

THE PLI SOLVER

`t:`



Scope: 5 Nodes

preH: `isBinTree()`

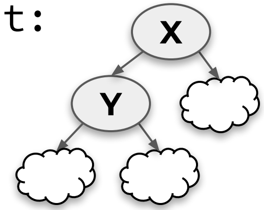
PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

PLI Solver

SymSolve

THE PLI SOLVER

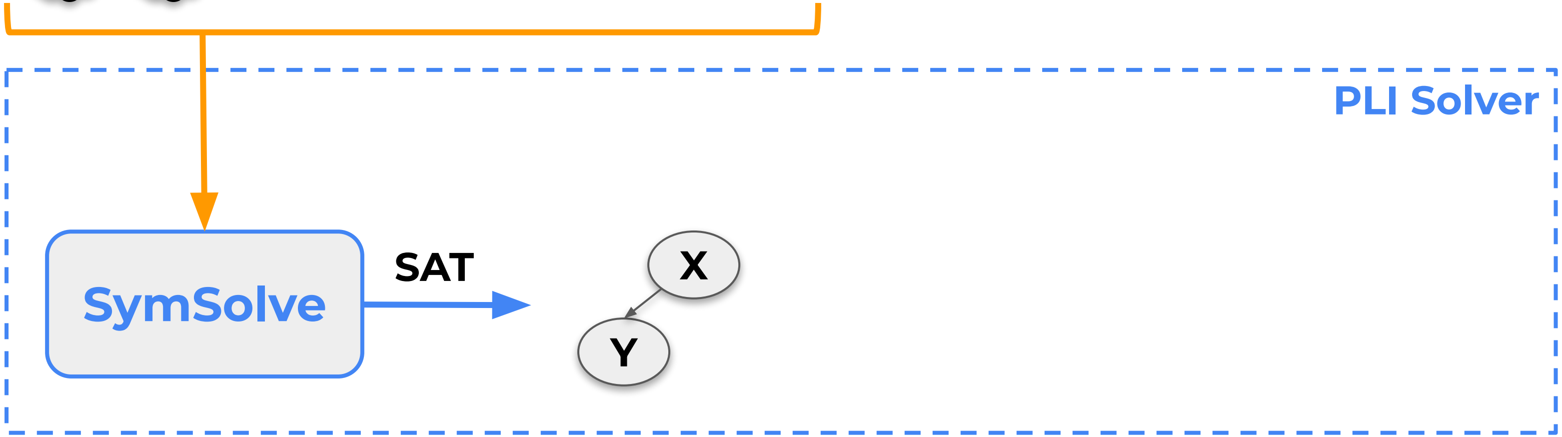


Scope: 5 Nodes

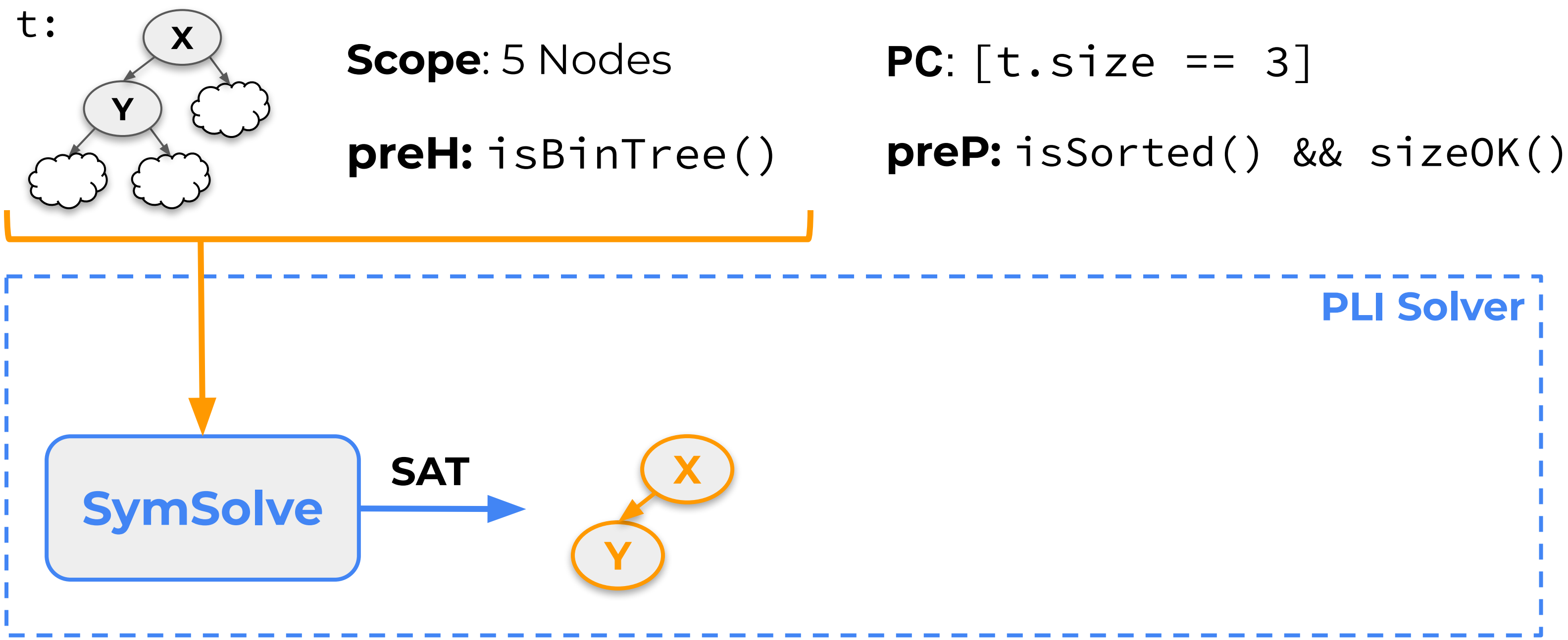
preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`



THE PLI SOLVER



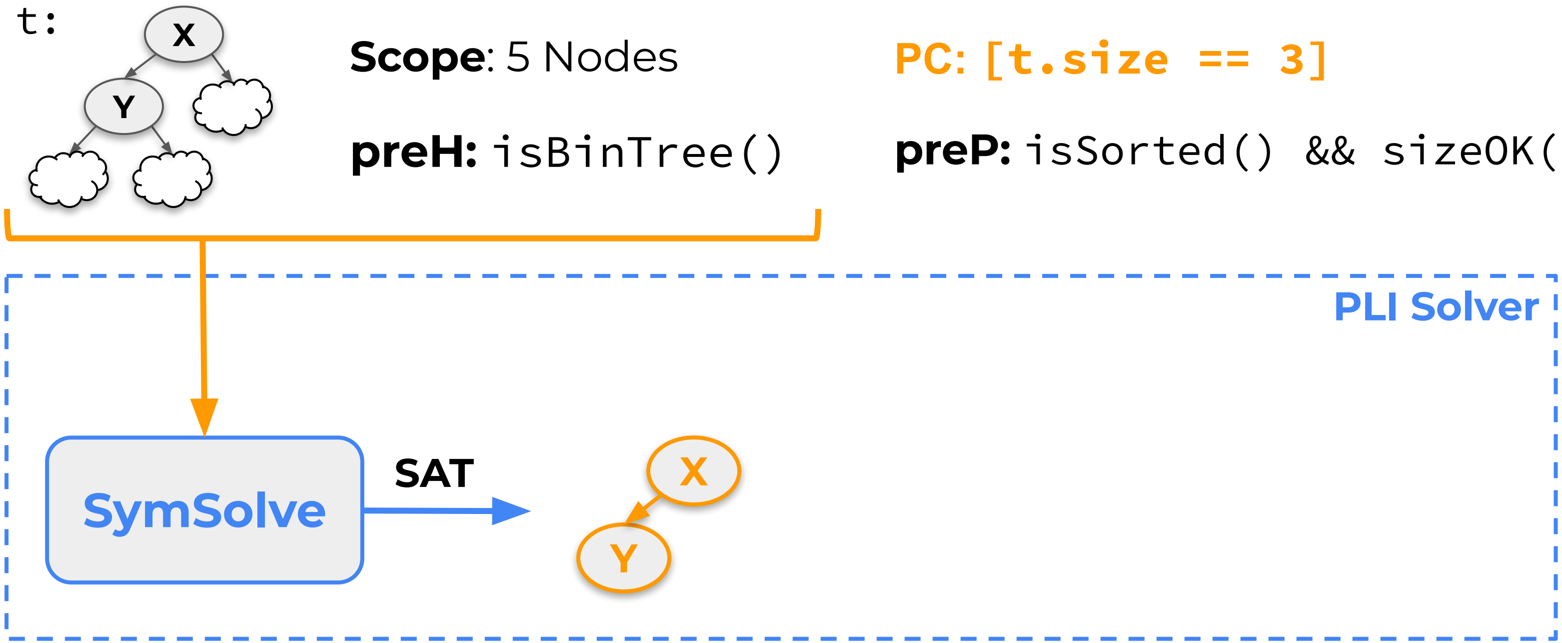
Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

THE PLI SOLVER



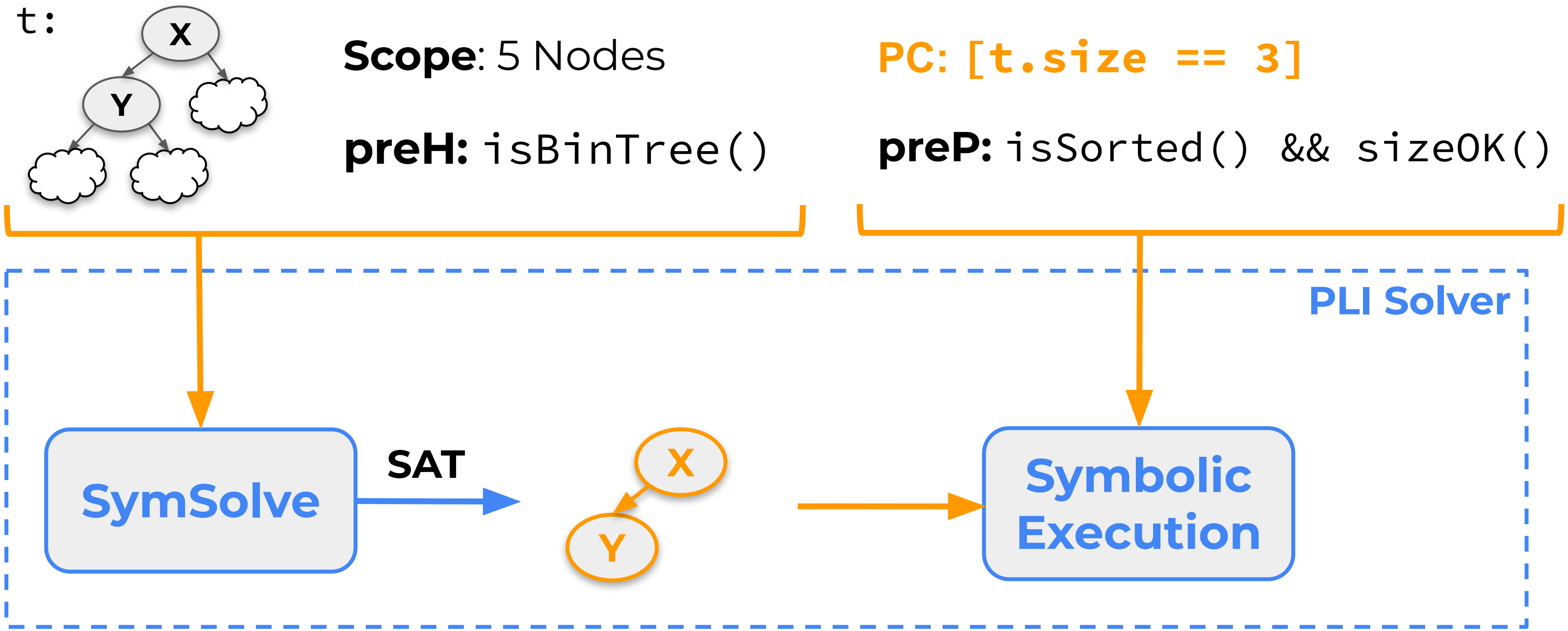
Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

THE PLI SOLVER



Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

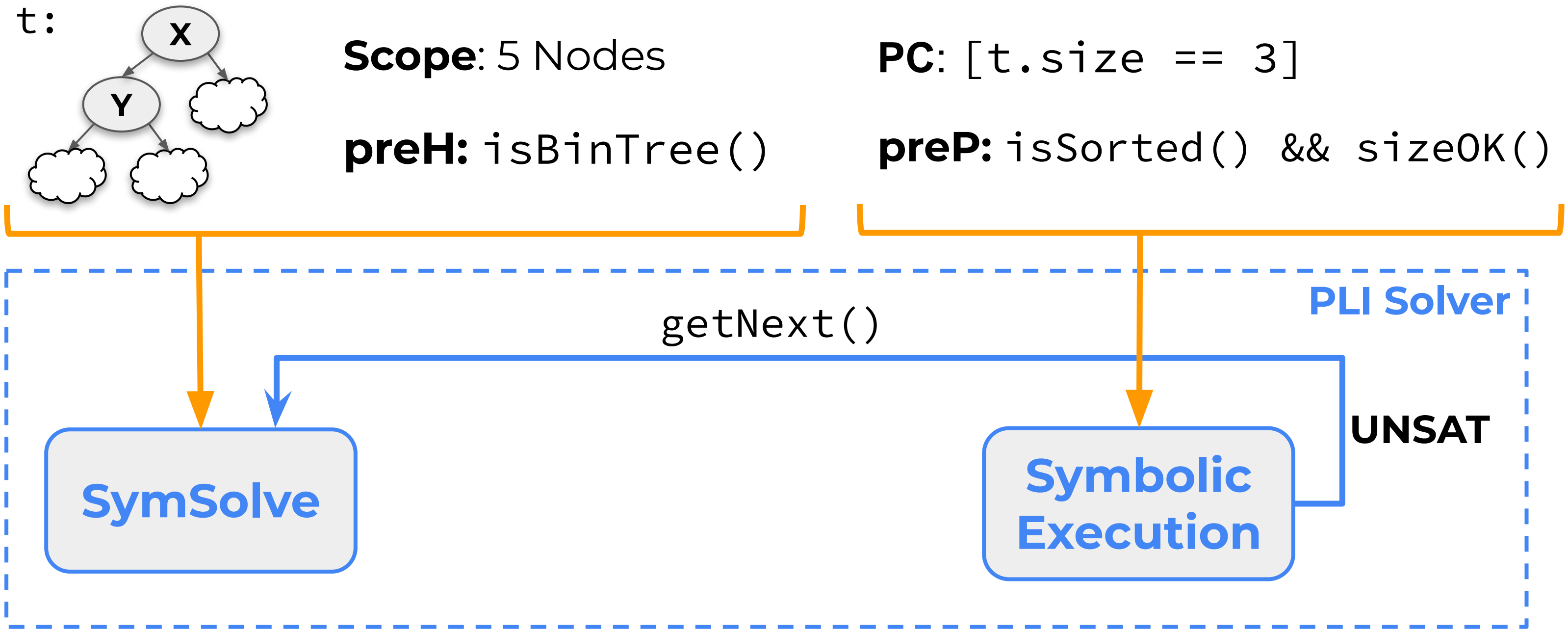
PLI Solver

SymSolve

SAT

Symbolic Execution

THE PLI SOLVER



Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

PLI Solver

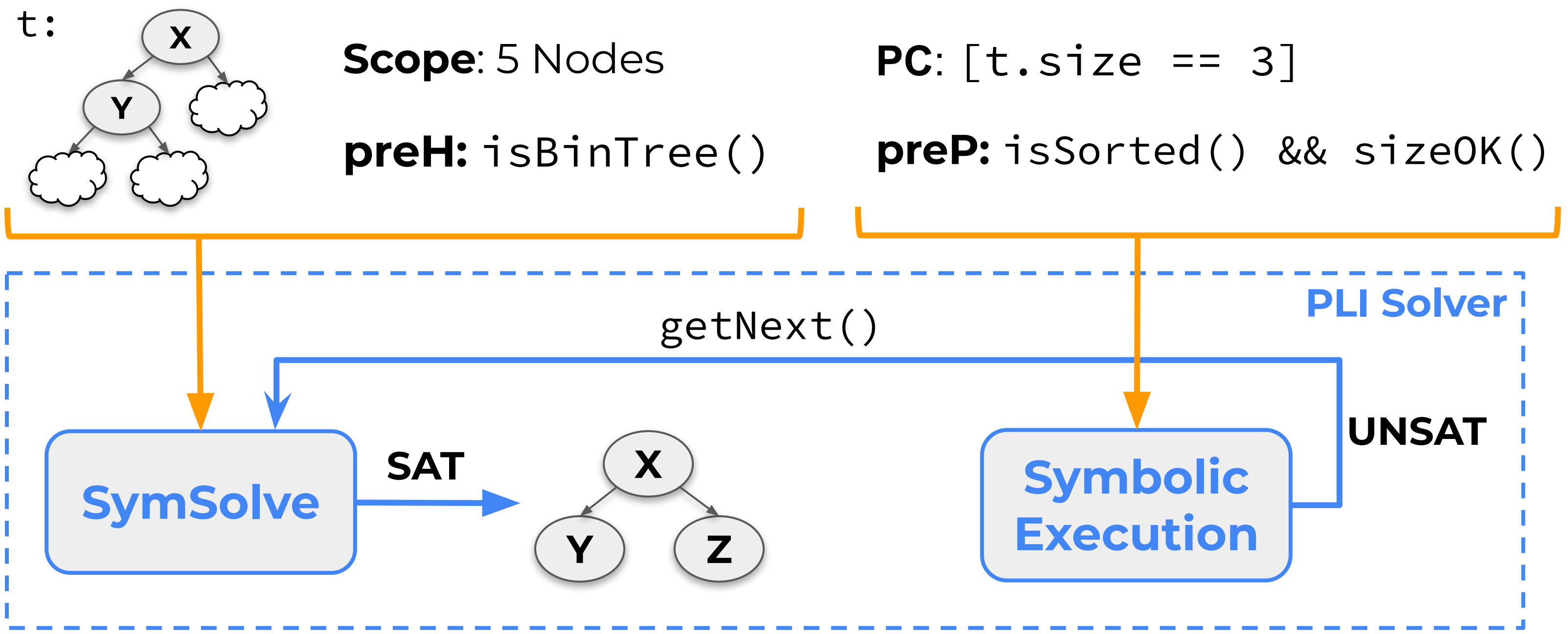
`getNext()`

UNSAT

SymSolve

Symbolic Execution

THE PLI SOLVER



`t:`

Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

PLI Solver

`getNext()`

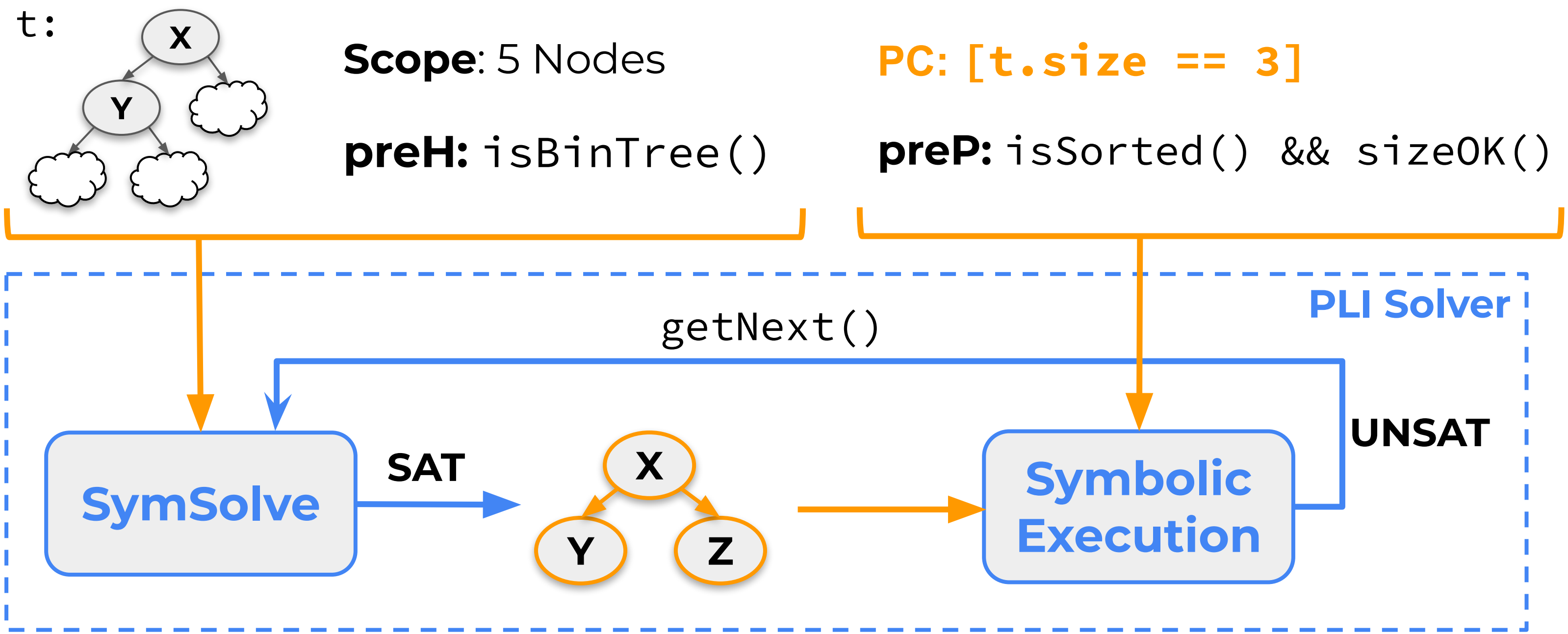
SymSolve

SAT

Symbolic Execution

UNSAT

THE PLI SOLVER



`t:`

Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

PLI Solver

`getNext()`

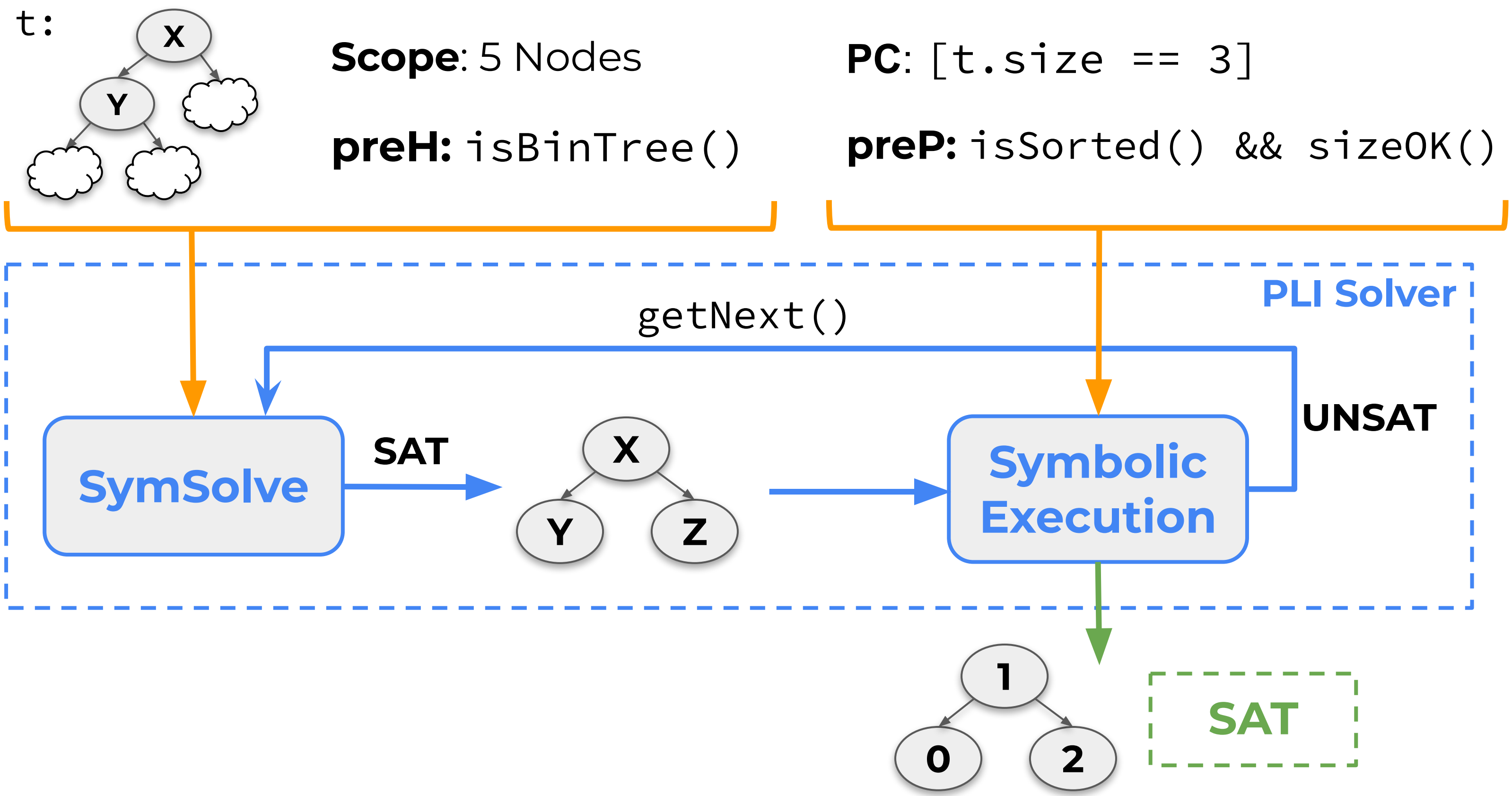
SymSolve

SAT

Symbolic Execution

UNSAT

THE PLI SOLVER



`t:`

X

Y

Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`

PLI Solver

`getNext()`

SymSolve

SAT

X

Y

Z

Symbolic Execution

UNSAT

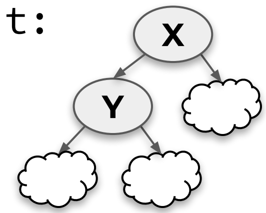
1

0

2

SAT

THE PLI SOLVER

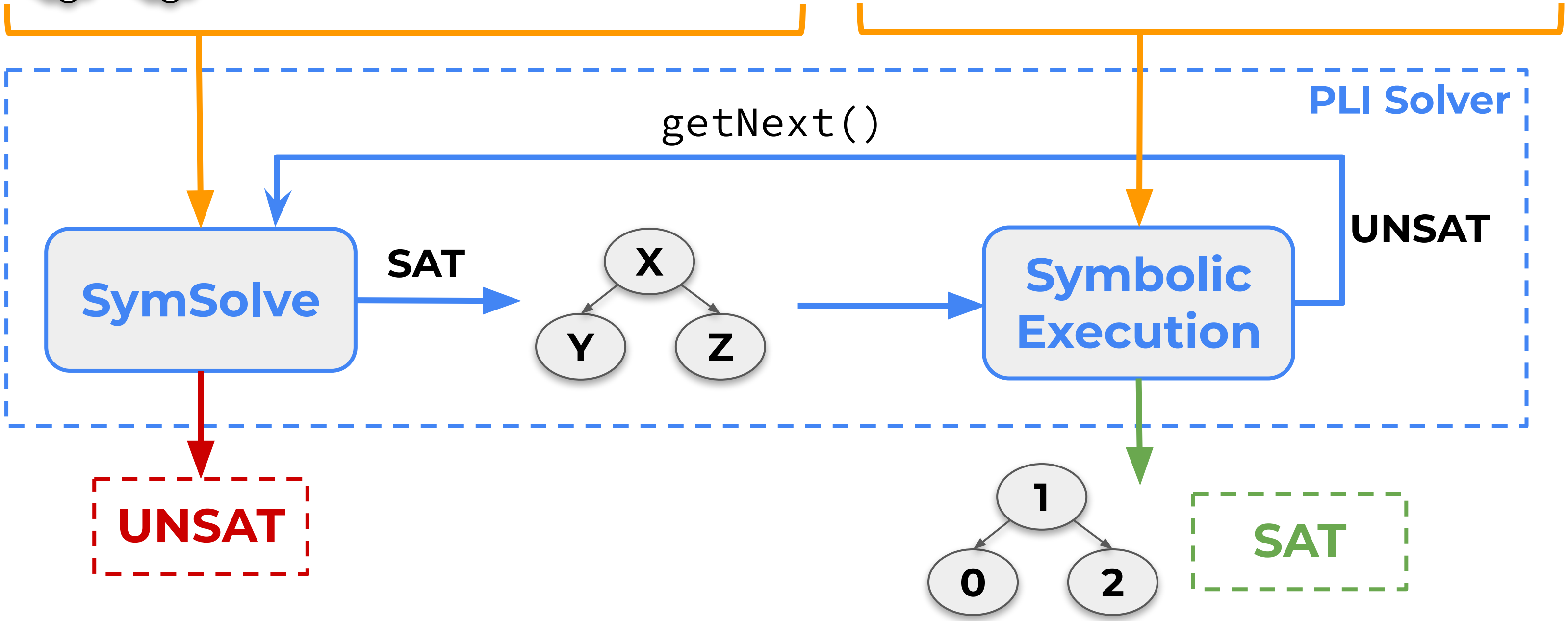


Scope: 5 Nodes

preH: `isBinTree()`

PC: `[t.size == 3]`

preP: `isSorted() && sizeOK()`



EXPERIMENTAL ASSESSMENT

EXPERIMENTAL ASSESSMENT

- We compared *PLI* against:
  - **2 Lazy approaches**: *LISSA* and *LI-HYBRID*.

EXPERIMENTAL ASSESSMENT

- We compared *PLI* against:
    - **2 Lazy approaches**: *LISSA* and *LI-HYBRID*.
    - **2 Eager approaches**: *IF-REPOK* and *DRIVER*.

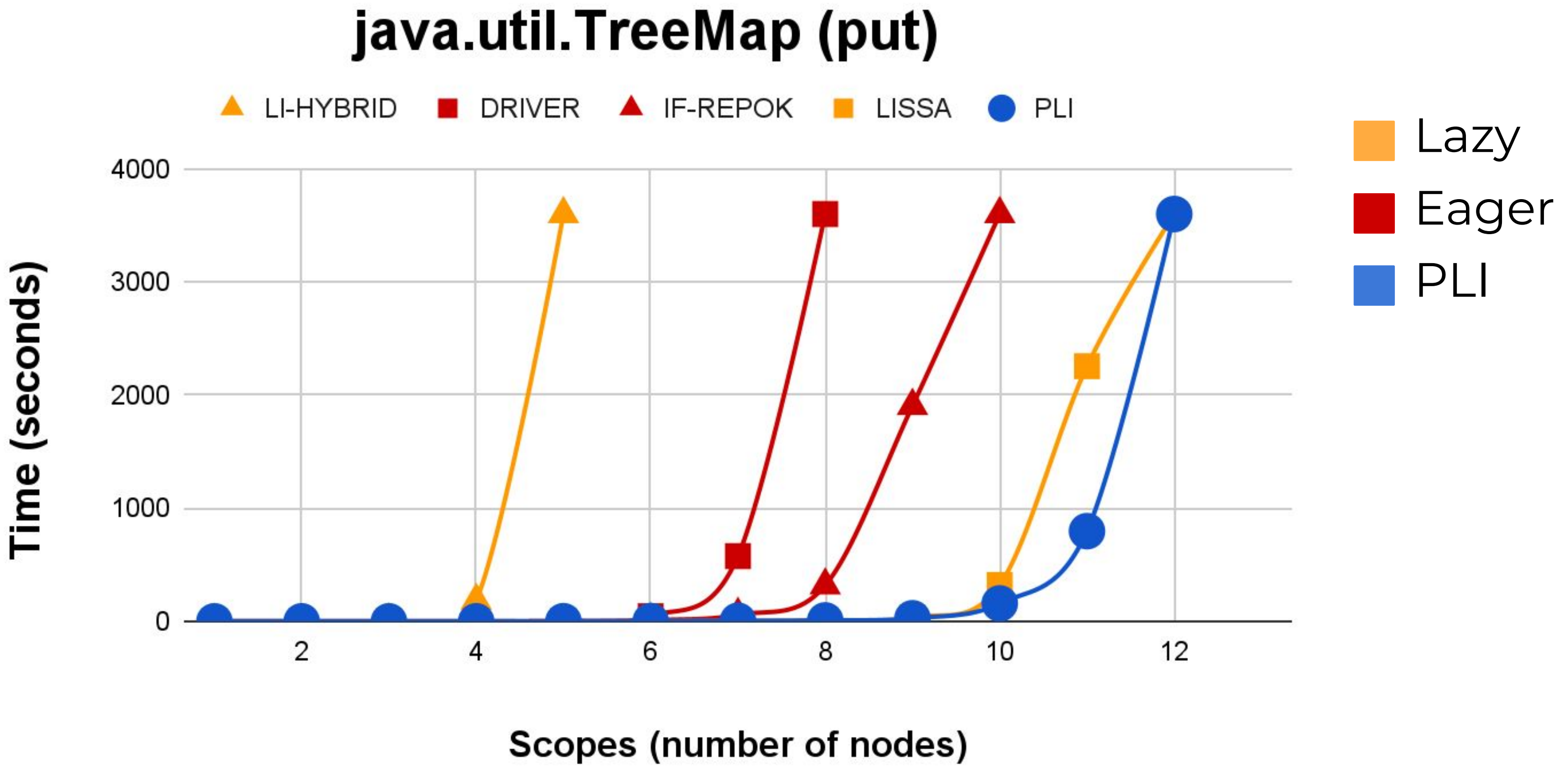
EXPERIMENTAL ASSESSMENT

- We compared *PLI* against:
    - **2 Lazy approaches**: *LISSA* and *LI-HYBRID*.
    - **2 Eager approaches**: *IF-REPOK* and *DRIVER*.

- We evaluated PLI on 12 case studies:
    - 4 Data structures from the **java.util** package: TreeMap, TreeSet, HashMap, LinkedList.
    - An AVL and BinomialHeap implementations from the literature.
    - 5 programs from **SF110**.
    - A scheduler implementation from the **SIR** repository.

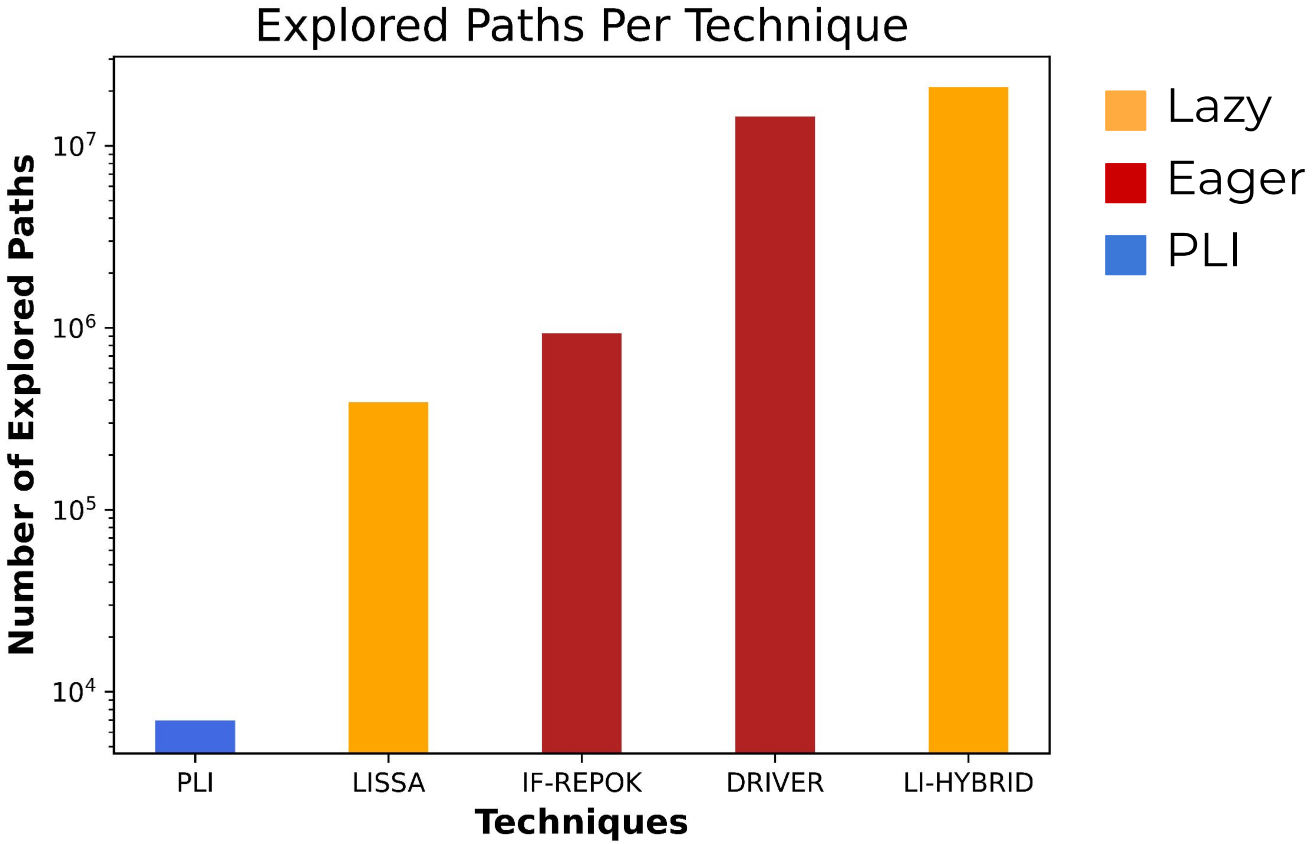
EXECUTION TIME AND SCALABILITY

EXECUTION TIME AND SCALABILITY

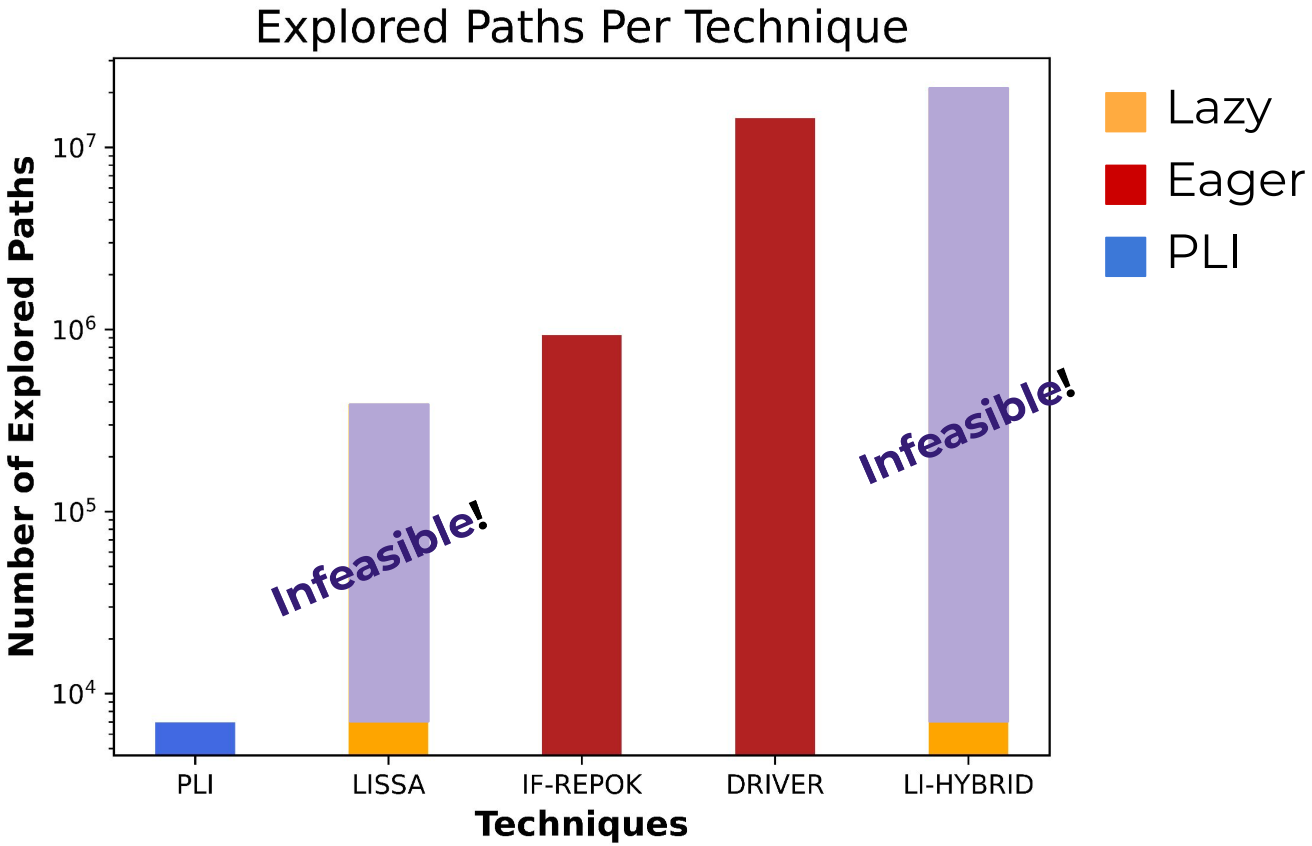


SYMBOLIC EXECUTION PATHS

SYMBOLIC EXECUTION PATHS

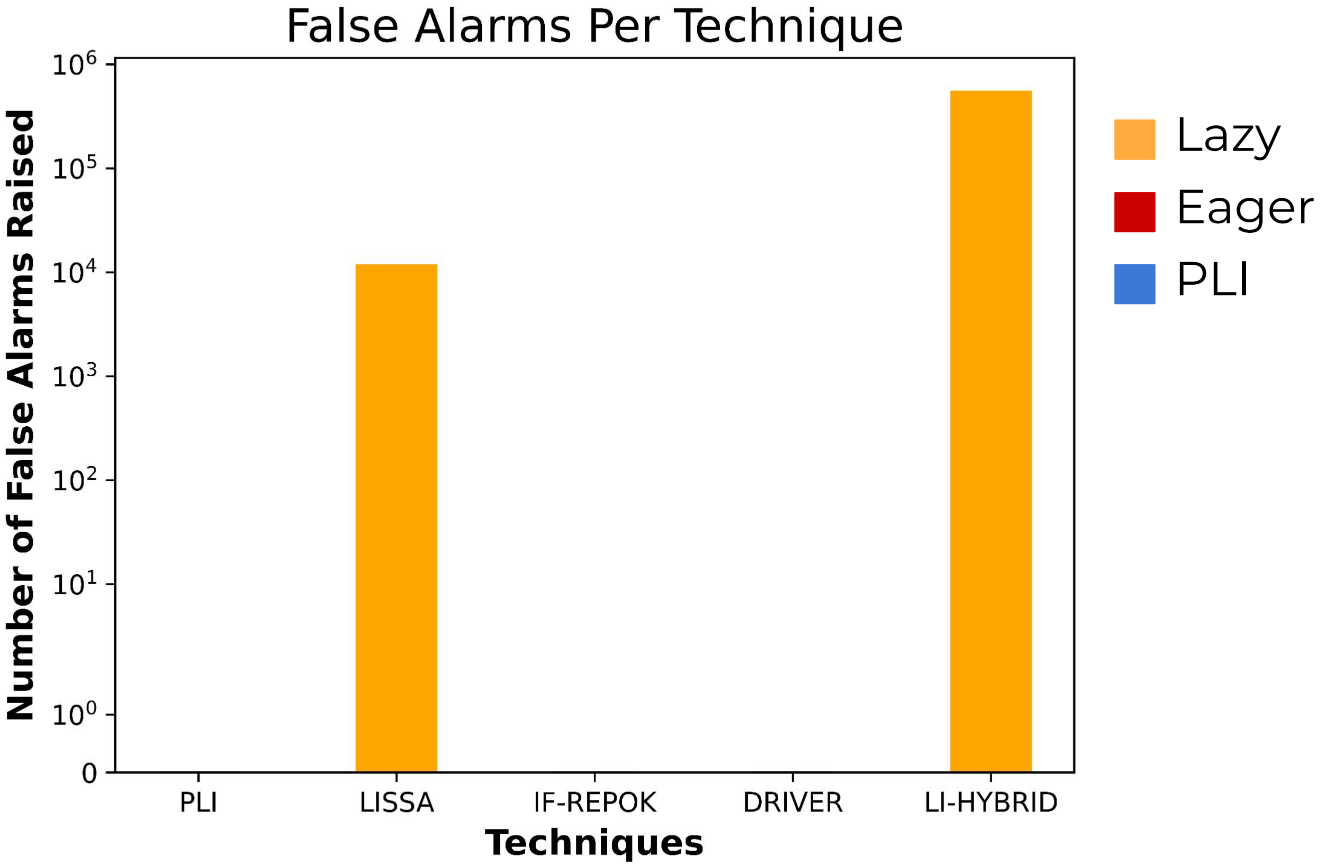


SYMBOLIC EXECUTION PATHS



FALSE ALARMS

FALSE ALARMS



CONCLUSION

CONCLUSION

- We developed PLI, a **lazy** symbolic execution technique for programs with heap-allocated inputs. PLI:

CONCLUSION

- We developed PLI, a **lazy** symbolic execution technique for programs with heap-allocated inputs. PLI:
  - Require **operational predicates** as specifications.

CONCLUSION

- We developed PLI, a **lazy** symbolic execution technique for programs with heap-allocated inputs. PLI:
    - Require **operational predicates** as specifications.
    - Solves the **path-condition / symbolic heap separation problem** of lazy approaches.

CONCLUSION

- We developed PLI, a **lazy** symbolic execution technique for programs with heap-allocated inputs. PLI:
    - Require **operational predicates** as specifications.
    - Solves the **path-condition / symbolic heap separation problem** of lazy approaches.
    - Eliminates **false positives** and **false alarms**.

CONCLUSION

- We developed PLI, a **lazy** symbolic execution technique for programs with heap-allocated inputs. PLI:
    - Require **operational predicates** as specifications.
    - Solves the **path-condition / symbolic heap separation problem** of lazy approaches.
    - Eliminates **false positives** and **false alarms**.
    - Performance is comparable to the fastest lazy approach.

THANK YOU!

The artifact received the **available**, **reviewed** and **reproducible** badges:

https://github.com/JuanmaCopia/spf-pli