KLEE Workshop 2024

Complex Test Input Generation in KLEE

Aleksei Babushkin Aleksandr Misonizhnik Yurii Kostyukov
Dmitry Mordvinov Dmitry Ivanov

Testing dynamically allocated structures

Testing dynamically allocated structures

```
 1 struct node {
 2   struct node *left;
 3   struct node *right;
 4   struct node *parent;
 5   int value;
 6 };
 7
 8 int main() {
 9   struct node *data = create_tree();
10
11   inspect(data);
12
13   return 0;
14 }
```

- Dynamically allocated recursive structures are everywhere (lists, trees, graphs, etc.)

Testing dynamically allocated structures

```
 1 struct node {
 2   struct node *left;
 3   struct node *right;
 4   struct node *parent;
 5   int value;
 6 };
 7
 8 int main() {
 9   struct node *data = create_tree();
10
11   inspect(data);
12
13   return 0;
14 }
```

- Dynamically allocated recursive structures are everywhere (lists, trees, graphs, etc.)
- Instances are operated with via a pointer to head

Testing dynamically allocated structures

```c
struct node {
  struct node *left;
  struct node *right;
  struct node *parent;
  int value;
};

int main() {
  struct node *data = create_tree();

  inspect(data);

  return 0;
}
```

- Dynamically allocated recursive structures are everywhere (lists, trees, graphs, etc.)
- Instances are operated with via a pointer to head
- The usual approach for testing is to write a test harness that generates an instance and then pass it to the function under test

Generating Procedure

Generating Procedure

```
 1 node *create_tree() {
 2   node *nodelast = NULL;
 3   node *node = NULL;
 4 
 5   while (make_symbolic_int()) {
 6     node = malloc(sizeof *node);
 7     *node = {NULL, nodelast, NULL, make_symbolic_int()};
 8 
 9     if (nodelast) {
10       nodelast->parent = node;
11     }
12 
13     nodelast = node;
14   }
15 
16   while (node != NULL) {
17     node->left = malloc(sizeof *node);
18     *node->left = {NULL, NULL, node, 42};
19 
20     node = node->right;
21   }
22 
23   return nodelast;
24 }
```

Generating Procedure

```
1  node *create_tree() {
2    node *nodelast = NULL;
3    node *node = NULL;
4
5    while (make_symbolic_int()) {
6      node = malloc(sizeof *node);
7      *node = {NULL, nodelast, NULL, make_symbolic_int()};
8
9      if (nodelast) {
10       nodelast->parent = node;
11     }
12
13     nodelast = node;
14   }
15
16   while (node != NULL) {
17     node->left = malloc(sizeof *node);
18     *node->left = {NULL, NULL, node, 42};
19
20     node = node->right;
21   }
22
23   return nodelast;
24 }
```

- Tedious to write

Generating Procedure

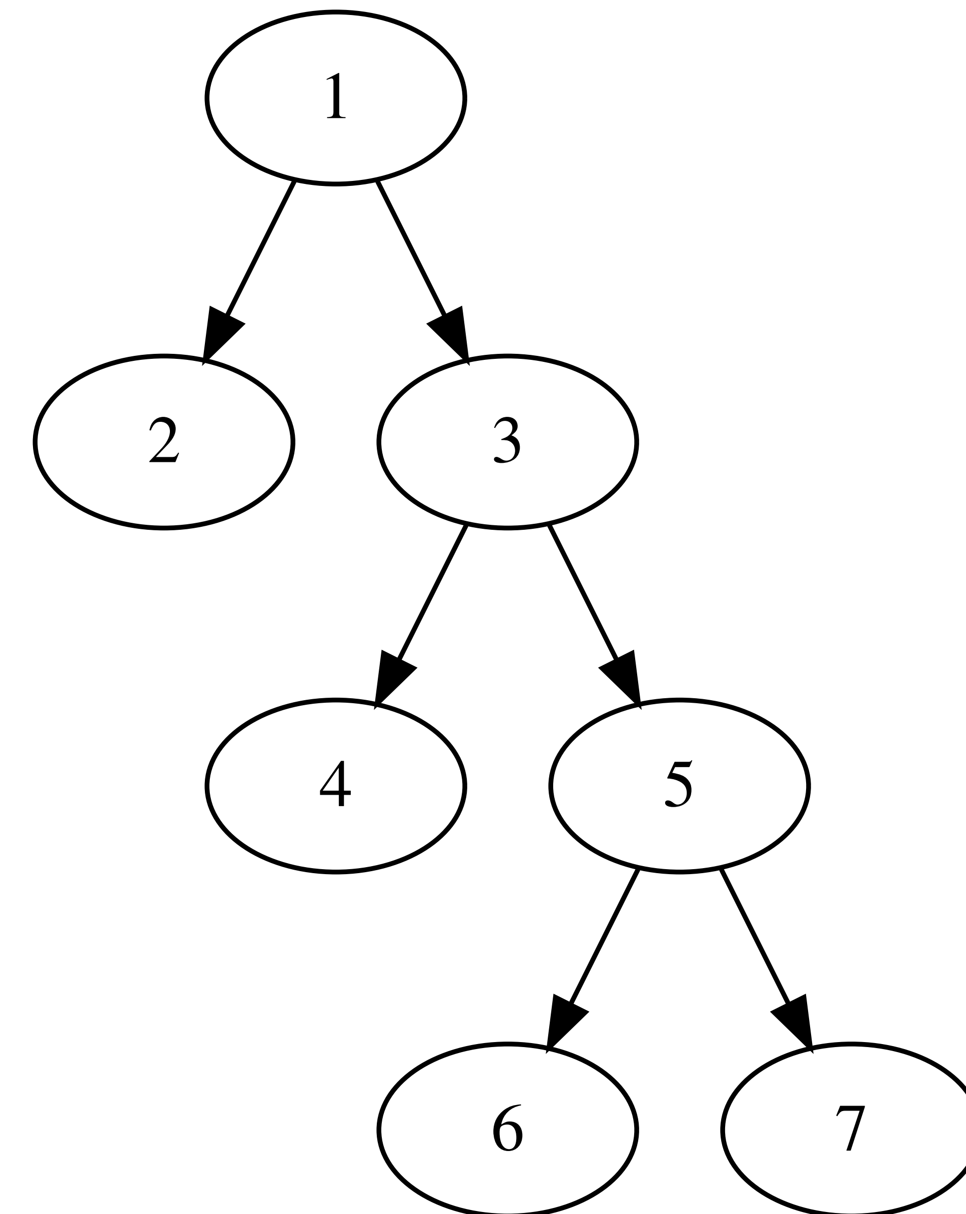
```
 1 node *create_tree() {
 2   node *nodelast = NULL;
 3   node *node = NULL;
 4
 5   while (make_symbolic_int()) {
 6     node = malloc(sizeof *node);
 7     *node = {NULL, nodelast, NULL, make_symbolic_int()};
 8
 9     if (nodelast) {
10       nodelast->parent = node;
11     }
12
13     nodelast = node;
14   }
15
16   while (node != NULL) {
17     node->left = malloc(sizeof *node);
18     *node->left = {NULL, NULL, node, 42};
19
20     node = node->right;
21   }
22
23   return nodelast;
24 }
```

- Tedious to write
- Detached from code under test

Generating Procedure

```
1  node *create_tree() {
2    node *nodelast = NULL;
3    node *node = NULL;
4
5    while (make_symbolic_int()) {
6      node = malloc(sizeof *node);
7      *node = {NULL, nodelast, NULL, make_symbolic_int()};
8
9      if (nodelast) {
10       nodelast->parent = node;
11     }
12
13     nodelast = node;
14   }
15
16   while (node != NULL) {
17     node->left = malloc(sizeof *node);
18     *node->left = {NULL, NULL, node, 42};
19
20     node = node->right;
21   }
22
23   return nodelast;
24 }
```

- Tedious to write
- Detached from code under test
- Usually incomplete in terms of generated instances



Lazy Initialization

Lazy Initialization

- Idea: The first time a symbolic pointer is dereferenced, allocate a memory object corresponding to that pointer

Lazy Initialization

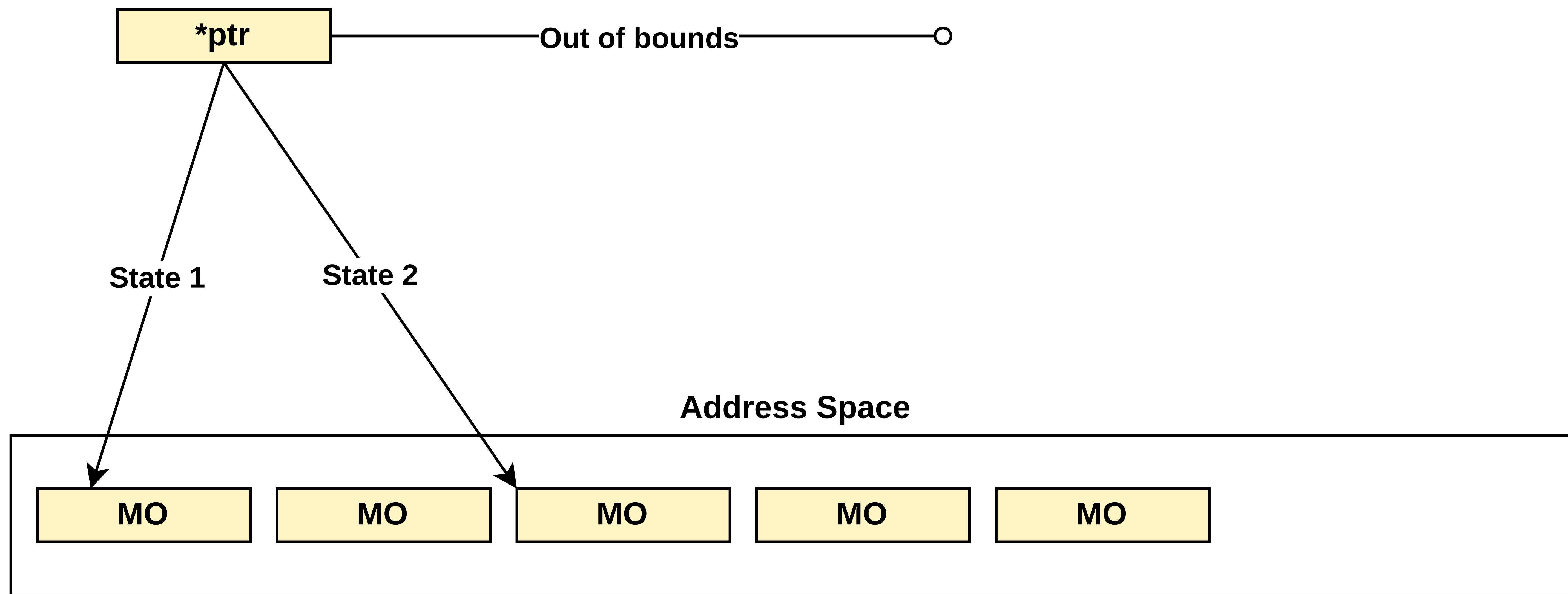
- Idea: The first time a symbolic pointer is dereferenced, allocate a memory object corresponding to that pointer
- Allows for automatic generation of recursive structures (lists, trees, etc.) by the code that operates on them

Lazy Initialization

- Idea: The first time a symbolic pointer is dereferenced, allocate a memory object corresponding to that pointer
- Allows for automatic generation of recursive structures (lists, trees, etc.) by the code that operates on them
- Reference paper: Khurshid, S., Păsăreanu, C.S. and Visser, W. Generalized symbolic execution for model checking and testing. TACAS 2003 (pp. 553-568)

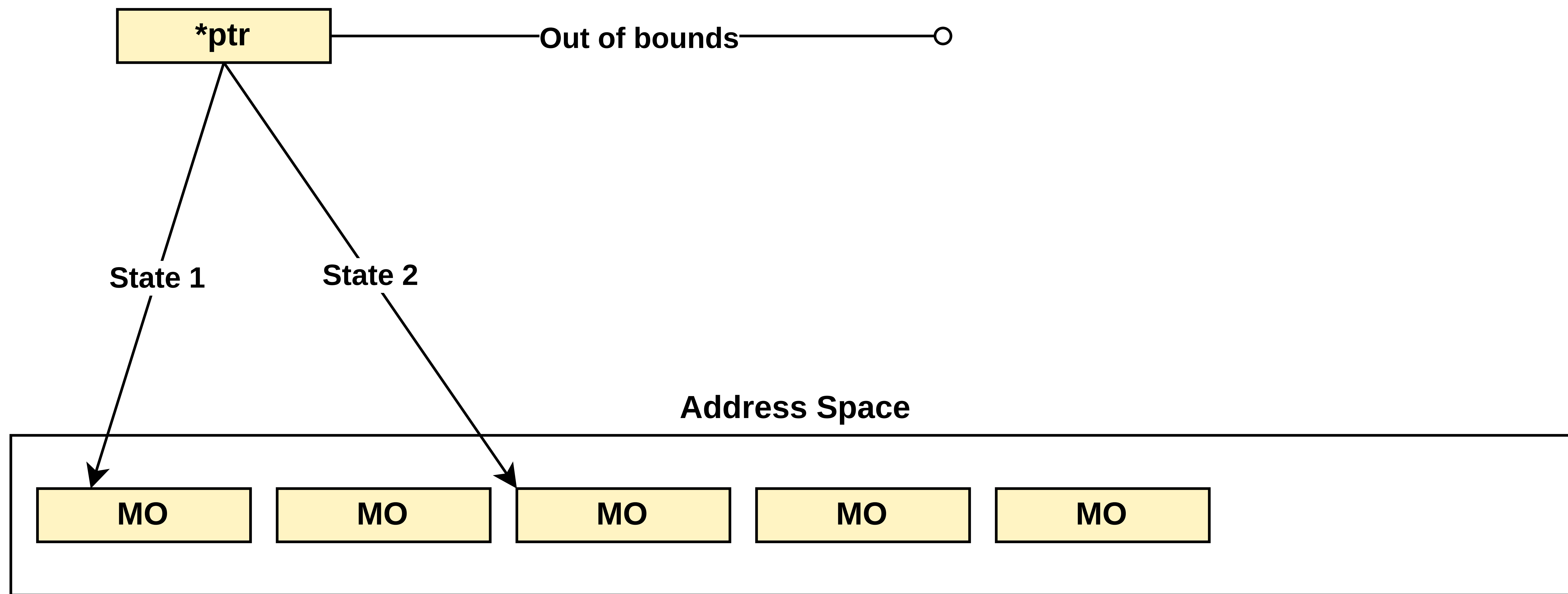
Lazy Initialization OFF (Vanilla KLEE)

Lazy Initialization OFF (Vanilla KLEE)



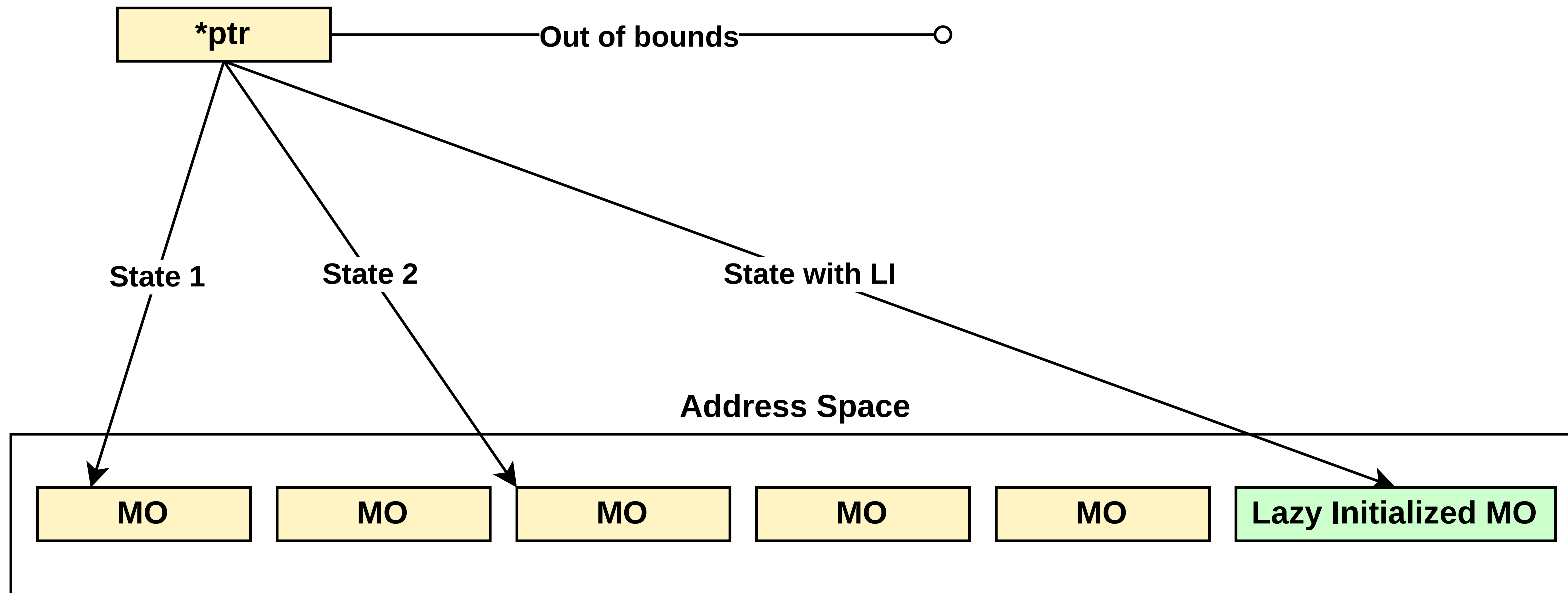
- A state for every suitable object + possible out-of-bounds state

Lazy Initialization OFF (Vanilla KLEE)



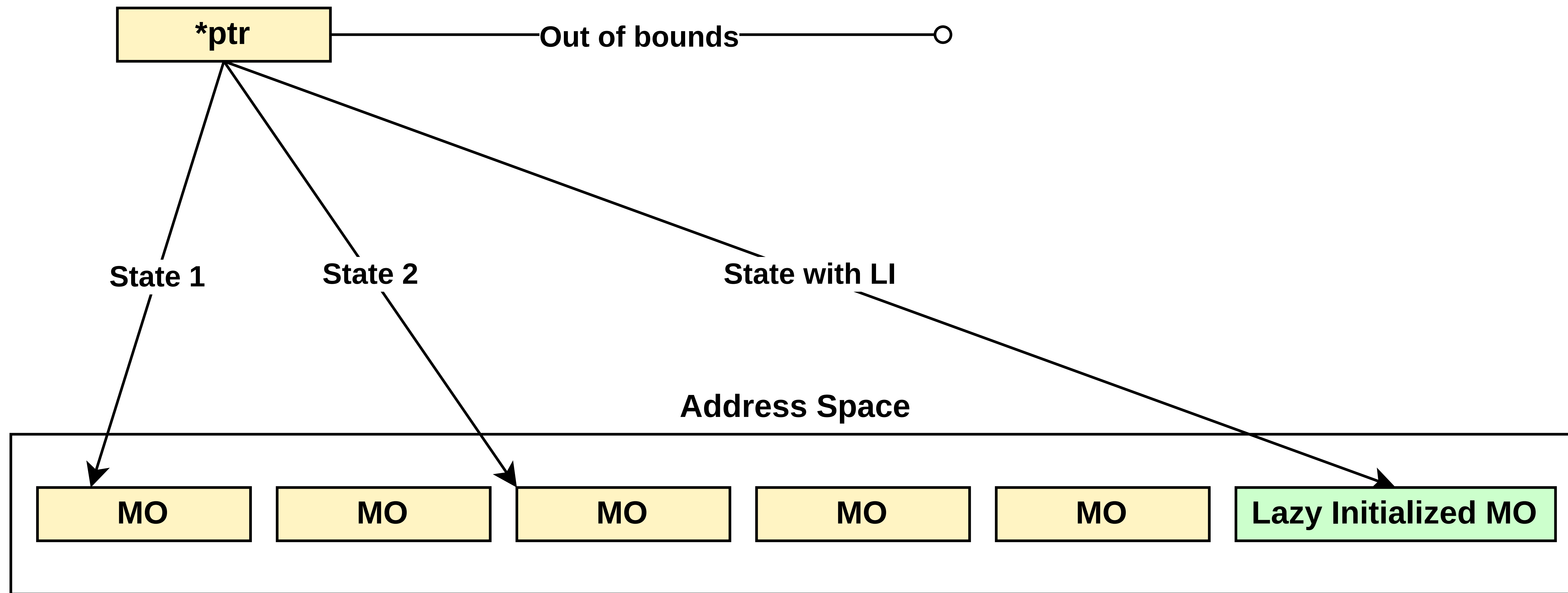
- A state for every suitable object + possible out-of-bounds state
- Each object separate in the address space, path constraints ensure that

Lazy Initialization ON



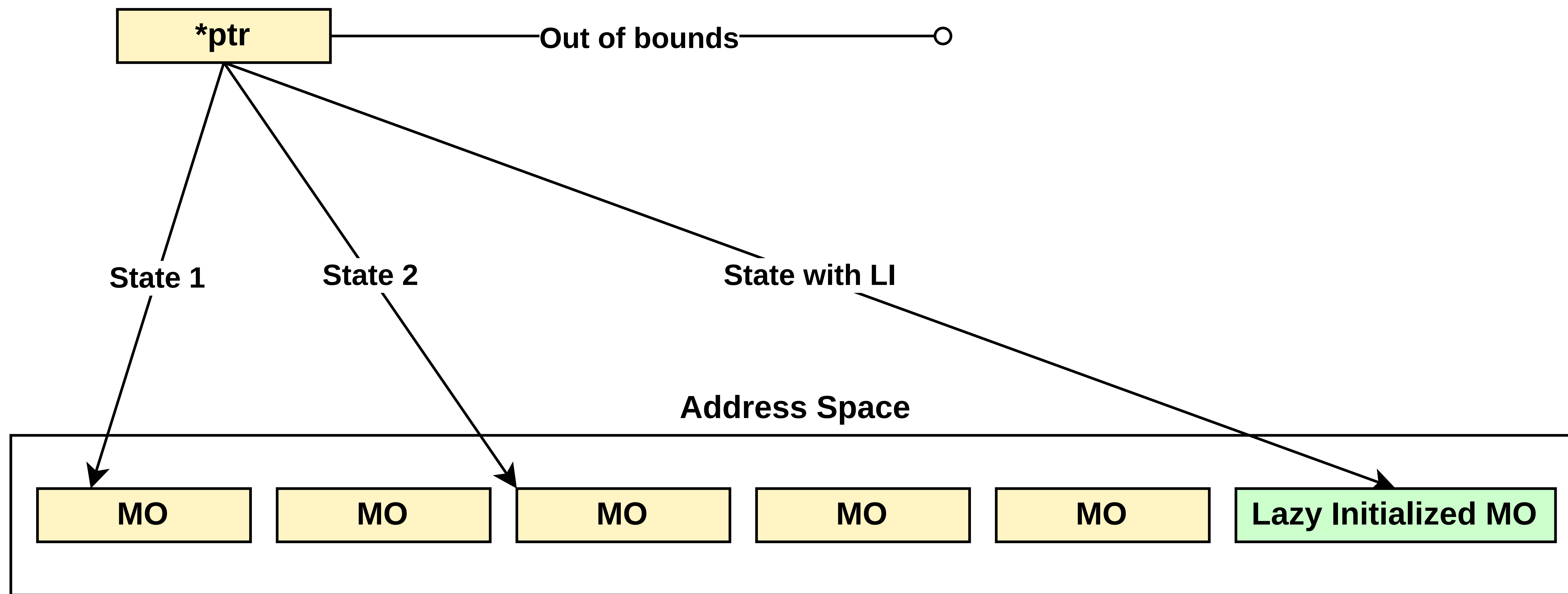
- A state for every suitable object + possible out-of-bounds state + **state with a new lazy initialized (LI) symbolic memory object**

Lazy Initialization ON



- A state for every suitable object + possible out-of-bounds state + **state with a new lazy initialized (LI) symbolic memory object**
- The LI object is associated with the pointer that is being dereferenced

Lazy Initialization ON



- A state for every suitable object + possible out-of-bounds state + **state with a new lazy initialized (LI) symbolic memory object**
- The LI object is associated with the pointer that is being dereferenced
- The pointer points inside the object:

$$\mathtt{LIAddress(ptr)} \leqslant \mathtt{ptr} < \mathtt{LIAddress(ptr)} + \mathtt{LISize(ptr)}$$

GetElementPtr and Pointer Resolution

GetElementPtr and Pointer Resolution

```
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

GetElementPtr and Pointer Resolution

```
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

```
1 %offset_ptr_b = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 1
2 %b = load i32, ptr %offset_ptr_b
3 ...
4 %offset_ptr_a = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 0
5 %a = load i32, ptr %offset_ptr_a
```

GetElementPtr and Pointer Resolution

```c
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

```llvm
1 %offset_ptr_b = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 1
2 %b = load i32, ptr %offset_ptr_b
3 ...
4 %offset_ptr_a = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 0
5 %a = load i32, ptr %offset_ptr_a
```

- Naive implementation would make two lazy initialized objects of type **i32**

GetElementPtr and Pointer Resolution

```c
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

```llvm
1 %offset_ptr_b = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 1
2 %b = load i32, ptr %offset_ptr_b
3 ...
4 %offset_ptr_a = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 0
5 %a = load i32, ptr %offset_ptr_a
```

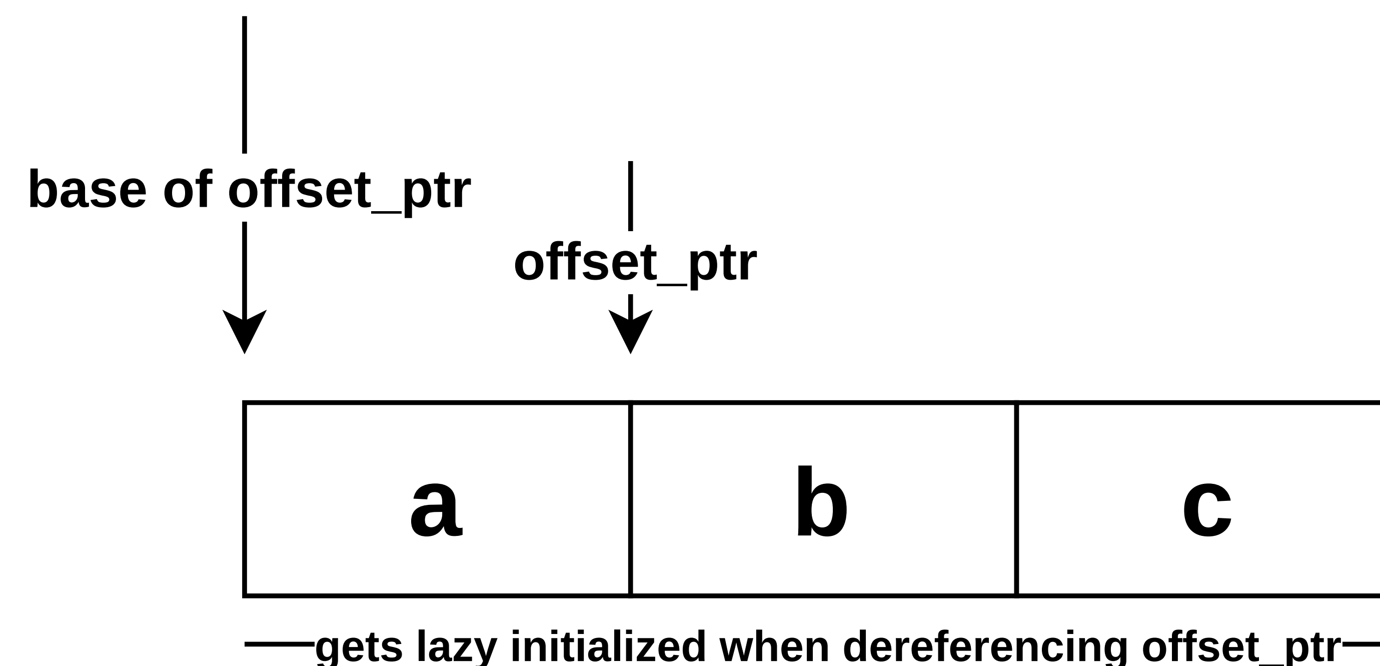
- Naive implementation would make two lazy initialized objects of type **i32**
- We would like one lazy initialized **Foo**

GetElementPtr and Pointer Resolution

```
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

```
1 %offset_ptr_b = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 1
2 %b = load i32, ptr %offset_ptr_b
3 ...
4 %offset_ptr_a = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 0
5 %a = load i32, ptr %offset_ptr_a
```

- Naive implementation would make two lazy initialized objects of type **i32**
- We would like one lazy initialized **Foo**
- Need to keep track of **GetElementPtr** instructions and lazy initialize accordingly

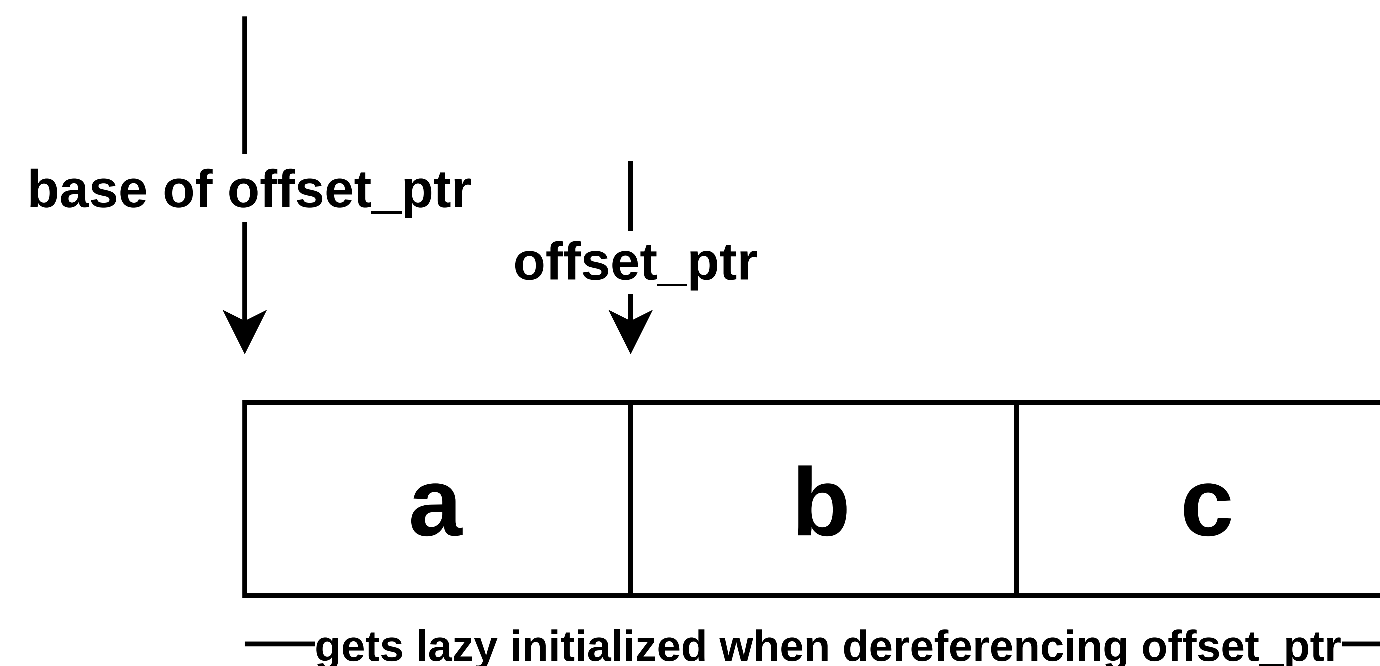


GetElementPtr and Pointer Resolution

```
1 struct Foo {
2   int a;
3   int b;
4   int c;
5 };
6
7 Foo *foo = make_symbolic_pointer();
8
9 assert(foo->b == 1 && foo->a == 2);
```

```
1 %offset_ptr_b = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 1
2 %b = load i32, ptr %offset_ptr_b
3 ...
4 %offset_ptr_a = getelementptr %struct.Foo, ptr %ptr, i32 0, i32 0
5 %a = load i32, ptr %offset_ptr_a
```

- Naive implementation would make two lazy initialized objects of type **i32**
- We would like one lazy initialized **Foo**
- Need to keep track of **GetElementPtr** instructions and lazy initialize accordingly
- GEPExprBases support already in KLEE 3.1 thanks to @tkuchta



Lazy Initialization for Arrays

Lazy Initialization for Arrays

```
1 int *array = make_symbolic_pointer();
2
3 int sum = 0;
4 for (size_t i = 0; i < 10; ++i) {
5   sum += array[i];
6 }
7
8 assert(sum > 30);
```

- When accessing **T***, the underlying object might be an array

Lazy Initialization for Arrays

```c
1 int *array = make_symbolic_pointer();
2
3 int sum = 0;
4 for (size_t i = 0; i < 10; ++i) {
5   sum += array[i];
6 }
7
8 assert(sum > 30);
```

- When accessing **T***, the underlying object might be an array
- Heuristic: allocate an array of some fixed number of elements and hope it's going to be big enough (variable by CLI option from run to run)

Lazy Initialization for Arrays

```c
1 int *array = make_symbolic_pointer();
2
3 int sum = 0;
4 for (size_t i = 0; i < 10; ++i) {
5   sum += array[i];
6 }
7
8 assert(sum > 30);
```

- When accessing **T***, the underlying object might be an array
- Heuristic: allocate an array of some fixed number of elements and hope it's going to be big enough (variable by CLI option from run to run)
- Possible solution: symbolic sizes

Restricted aliasing

Restricted aliasing

- KLEE would try to resolve to all objects in the address space

Restricted aliasing

- KLEE would try to resolve to all objects in the address space
- The resulting test cases are often not replayable (e.g. addresses of globals are not fixed if ASLR is enabled)

Restricted aliasing

- KLEE would try to resolve to all objects in the address space
- The resulting test cases are often not replayable (e.g. addresses of globals are not fixed if ASLR is enabled)
- Too many objects result in path explosion

Restricted aliasing

- KLEE would try to resolve to all objects in the address space
- The resulting test cases are often not replayable (e.g. addresses of globals are not fixed if ASLR is enabled)
- Too many objects result in path explosion
- LI with restricted aliasing produces replayable tests

Restricted aliasing

- KLEE would try to resolve to all objects in the address space
- The resulting test cases are often not replayable (e.g. addresses of globals are not fixed if ASLR is enabled)
- Too many objects result in path explosion
- LI with restricted aliasing produces replayable tests

Available options:

- Resolve only to symbolic objects (make_symbolic and lazy initialized) or try to lazy initialize
- Resolve only to lazy initialized objects or try to lazy initialize
- Always try to lazy initialize without resolving to already existing objects

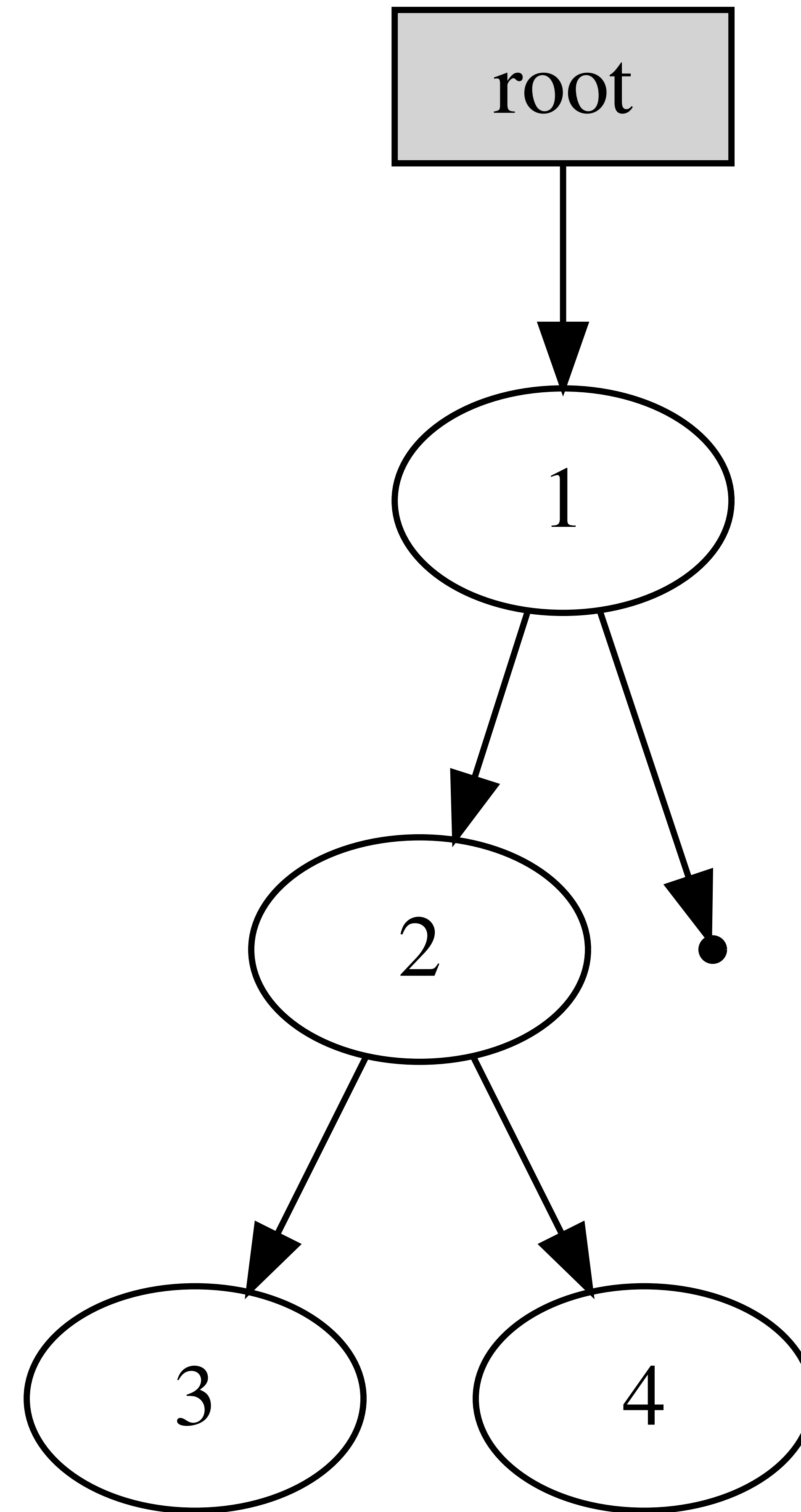
Example

Example

```
1  struct node {
2    node *left;
3    node *right;
4    int64 value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```

Example

```
1  struct node {
2    node *left;
3    node *right;
4    int64 value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



KTest Extension

KTest Extension

```
ktest file : 'test000009.ktest'
args       : ['tree.bc']
num objects: 5  // Root pointer and 4 vertices

object 0: name: 'root'
object 0: size: 8
object 0: hex : 0x0000000000000000
object 0: pointers: [(0, 1, 0)] // Pointer at offset 0 points to object 1 at offset 0

object 1: name: 'lazy_initialized'
object 1: size: 24
object 1: hex : 0x000000000000000000000000000000000100000000000000
object 1: pointers: [(0, 2, 0)]

object 2: name: 'lazy_initialized'
object 2: size: 24
object 2: hex : 0x000000000000000000000000000000000200000000000000
object 2: pointers: [(0, 3, 0), (8, 4, 0)]

object 3: name: 'lazy_initialized'
object 3: size: 24
object 3: hex : 0x000000000000000000000000000000000300000000000000

object 4: name: 'lazy_initialized'
object 4: size: 24
object 4: hex : 0x000000000000000000000000000000000400000000000000
```

KTest Extension

```
ktest file : 'test000009.ktest'
args       : ['tree.bc']
num objects: 5  // Root pointer and 4 vertices

object 0: name: 'root'
object 0: size: 8
object 0: hex : 0x0000000000000000
object 0: pointers: [(0, 1, 0)] // Pointer at offset 0 points to object 1 at offset 0

object 1: name: 'lazy_initialized'
object 1: size: 24
object 1: hex : 0x000000000000000000000000000000000100000000000000
object 1: pointers: [(0, 2, 0)]

object 2: name: 'lazy_initialized'
object 2: size: 24
object 2: hex : 0x000000000000000000000000000000000200000000000000
object 2: pointers: [(0, 3, 0), (8, 4, 0)]

object 3: name: 'lazy_initialized'
object 3: size: 24
object 3: hex : 0x000000000000000000000000000000000300000000000000

object 4: name: 'lazy_initialized'
object 4: size: 24
object 4: hex : 0x000000000000000000000000000000000400000000000000
```

KTest Extension

```
ktest file : 'test000009.ktest'
args       : ['tree.bc']
num objects: 5  // Root pointer and 4 vertices

object 0: name: 'root'
object 0: size: 8
object 0: hex : 0x0000000000000000
object 0: pointers: [(0, 1, 0)] // Pointer at offset 0 points to object 1 at offset 0

object 1: name: 'lazy_initialized'
object 1: size: 24
object 1: hex : 0x000000000000000000000000000000000100000000000000
object 1: pointers: [(0, 2, 0)]

object 2: name: 'lazy_initialized'
object 2: size: 24
object 2: hex : 0x000000000000000000000000000000000200000000000000
object 2: pointers: [(0, 3, 0), (8, 4, 0)]

object 3: name: 'lazy_initialized'
object 3: size: 24
object 3: hex : 0x000000000000000000000000000000000300000000000000

object 4: name: 'lazy_initialized'
object 4: size: 24
object 4: hex : 0x000000000000000000000000000000000400000000000000
```

KTest Extension

```
ktest file : 'test000009.ktest'
args       : ['tree.bc']
num objects: 5  // Root pointer and 4 vertices

object 0: name: 'root'
object 0: size: 8
object 0: hex : 0x0000000000000000
object 0: pointers: [(0, 1, 0)] // Pointer at offset 0 points to object 1 at offset 0

object 1: name: 'lazy_initialized'
object 1: size: 24
object 1: hex : 0x000000000000000000000000000000000100000000000000
object 1: pointers: [(0, 2, 0)]

object 2: name: 'lazy_initialized'
object 2: size: 24
object 2: hex : 0x000000000000000000000000000000000200000000000000
object 2: pointers: [(0, 3, 0), (8, 4, 0)]

object 3: name: 'lazy_initialized'
object 3: size: 24
object 3: hex : 0x000000000000000000000000000000000300000000000000

object 4: name: 'lazy_initialized'
object 4: size: 24
object 4: hex : 0x000000000000000000000000000000000400000000000000
```

KTest Extension

```
ktest file : 'test000009.ktest'
args       : ['tree.bc']
num objects: 5  // Root pointer and 4 vertices

object 0: name: 'root'
object 0: size: 8
object 0: hex : 0x0000000000000000
object 0: pointers: [(0, 1, 0)] // Pointer at offset 0 points to object 1 at offset 0

object 1: name: 'lazy_initialized'
object 1: size: 24
object 1: hex : 0x000000000000000000000000000000000100000000000000
object 1: pointers: [(0, 2, 0)]

object 2: name: 'lazy_initialized'
object 2: size: 24
object 2: hex : 0x000000000000000000000000000000000200000000000000
object 2: pointers: [(0, 3, 0), (8, 4, 0)]

object 3: name: 'lazy_initialized'
object 3: size: 24
object 3: hex : 0x000000000000000000000000000000000300000000000000

object 4: name: 'lazy_initialized'
object 4: size: 24
object 4: hex : 0x000000000000000000000000000000000400000000000000
```

Replay

Replay

- Keep track of allocated objects and their addresses during replay

Replay

- Keep track of allocated objects and their addresses during replay
- Recursively allocate objects the current object points to (if not already allocated) and write their addresses to respective pointers in the current object

Replay

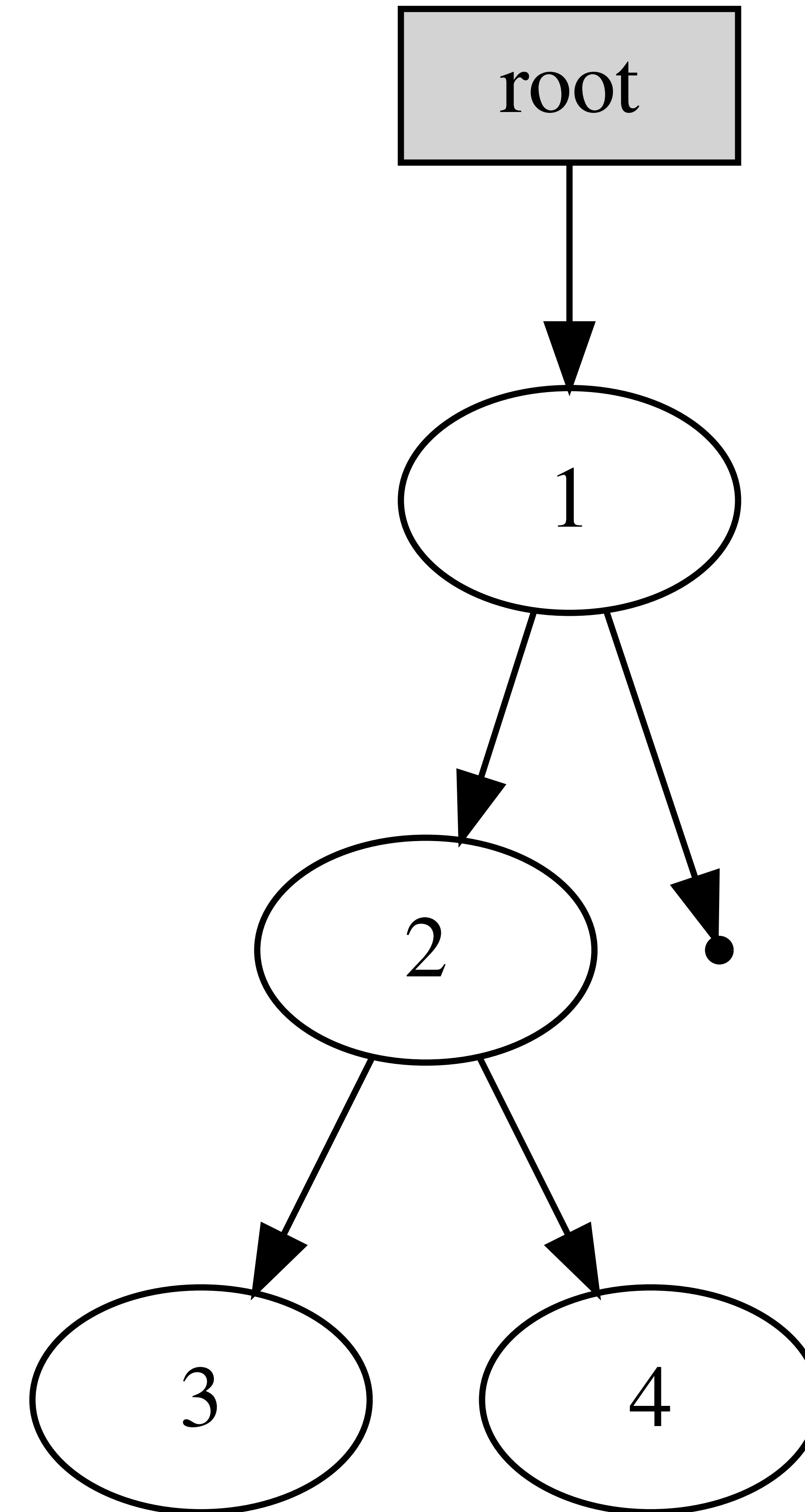
- Keep track of allocated objects and their addresses during replay
- Recursively allocate objects the current object points to (if not already allocated) and write their addresses to respective pointers in the current object
- When encountering a call to **klee_make_symbolic**, take next not lazy initialized object in the KTest

Replay

- Keep track of allocated objects and their addresses during replay
- Recursively allocate objects the current object points to (if not already allocated) and write their addresses to respective pointers in the current object
- When encountering a call to **klee_make_symbolic**, take next not lazy initialized object in the KTest
- Replay procedure otherwise unchanged

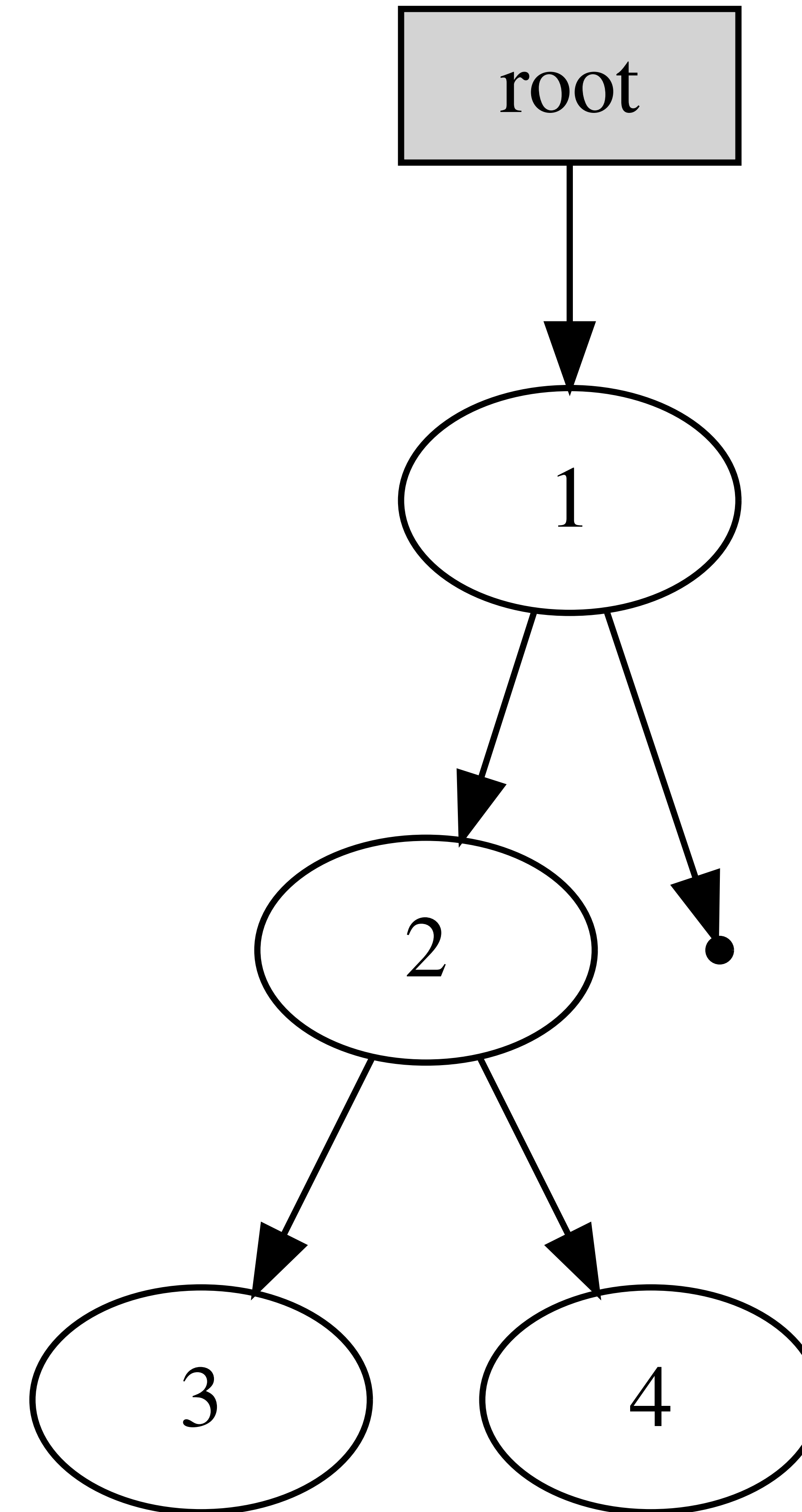
Replay Example

```
 1 struct node {
 2   node *left;
 3   node *right;
 4   int value;
 5 };
 6
 7 int main() {
 8   node *root = make_symbolic_pointer();
 9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



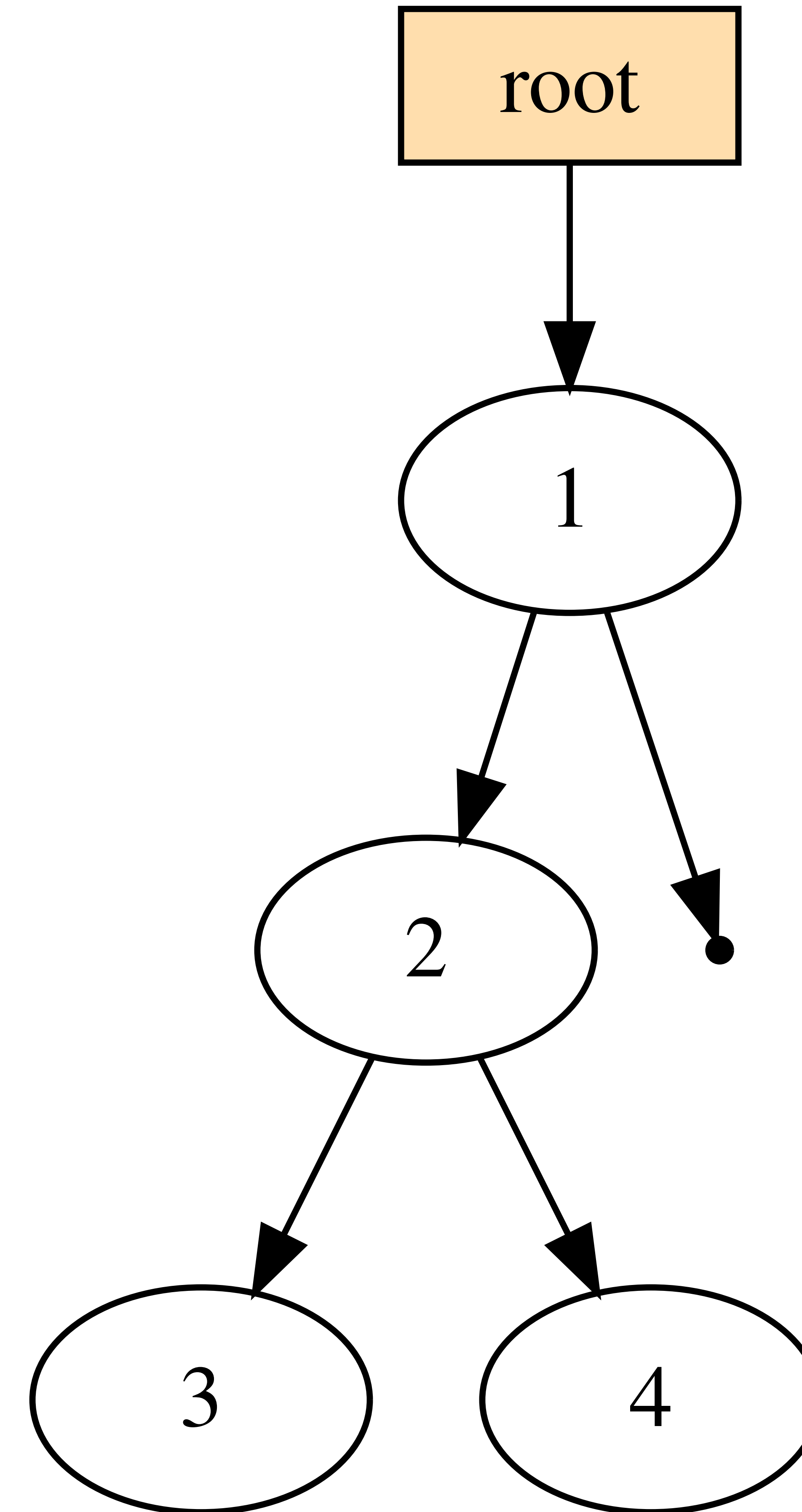
Replay Example

```
1  struct node {
2    node *left;
3    node *right;
4    int value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



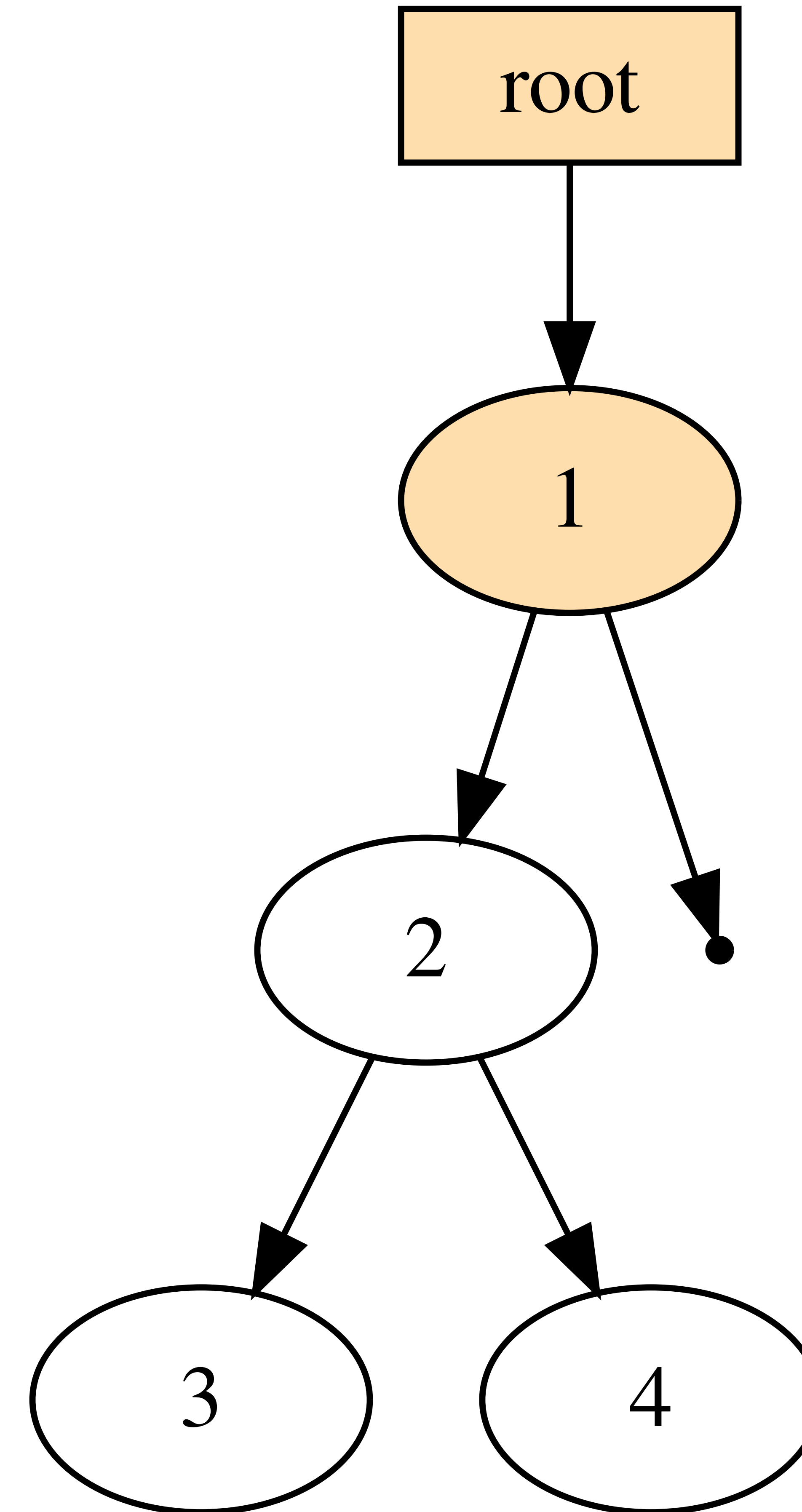
Replay Example

```
1  struct node {
2    node *left;
3    node *right;
4    int value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



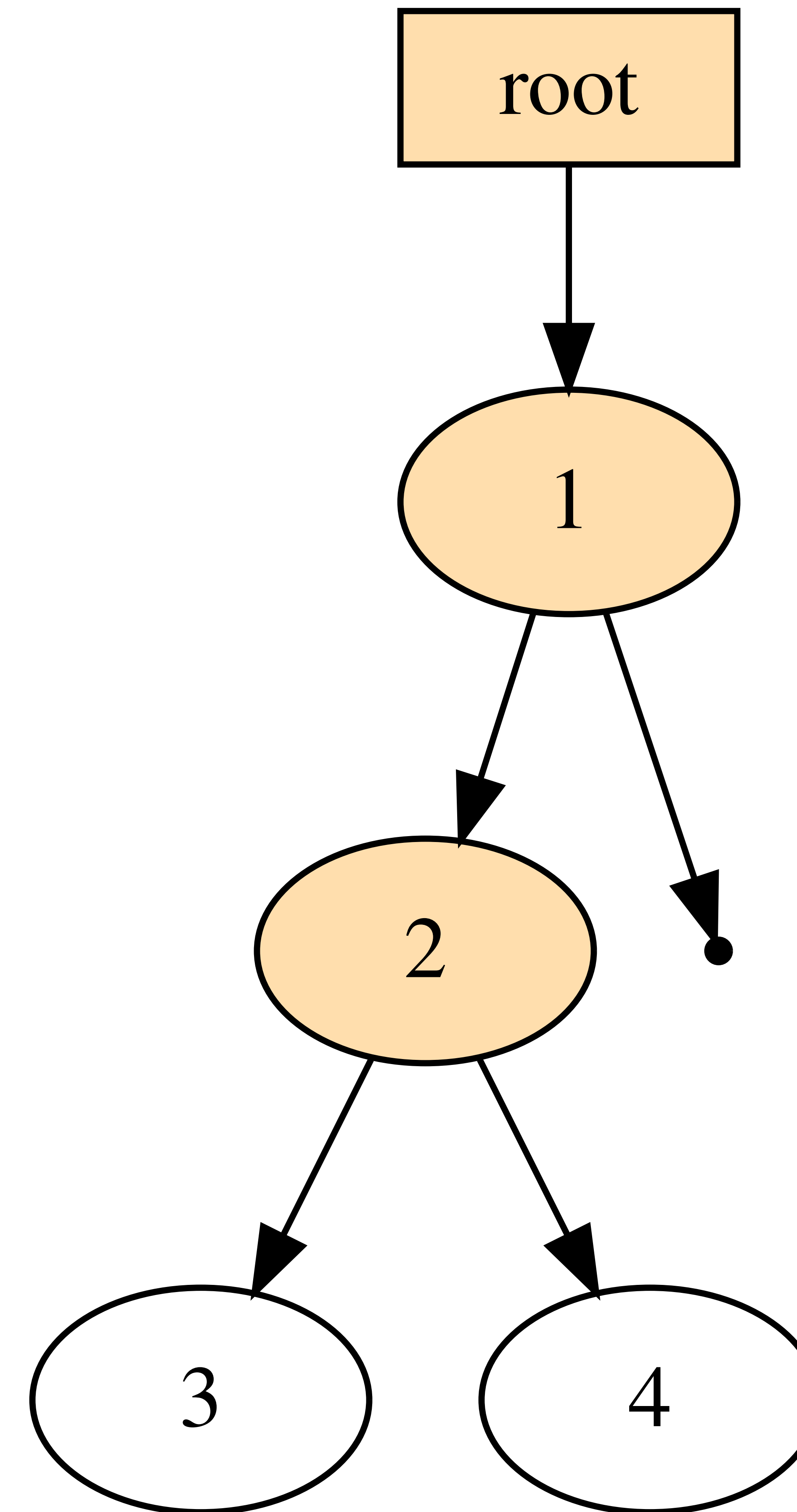
Replay Example

```
1  struct node {
2    node *left;
3    node *right;
4    int value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



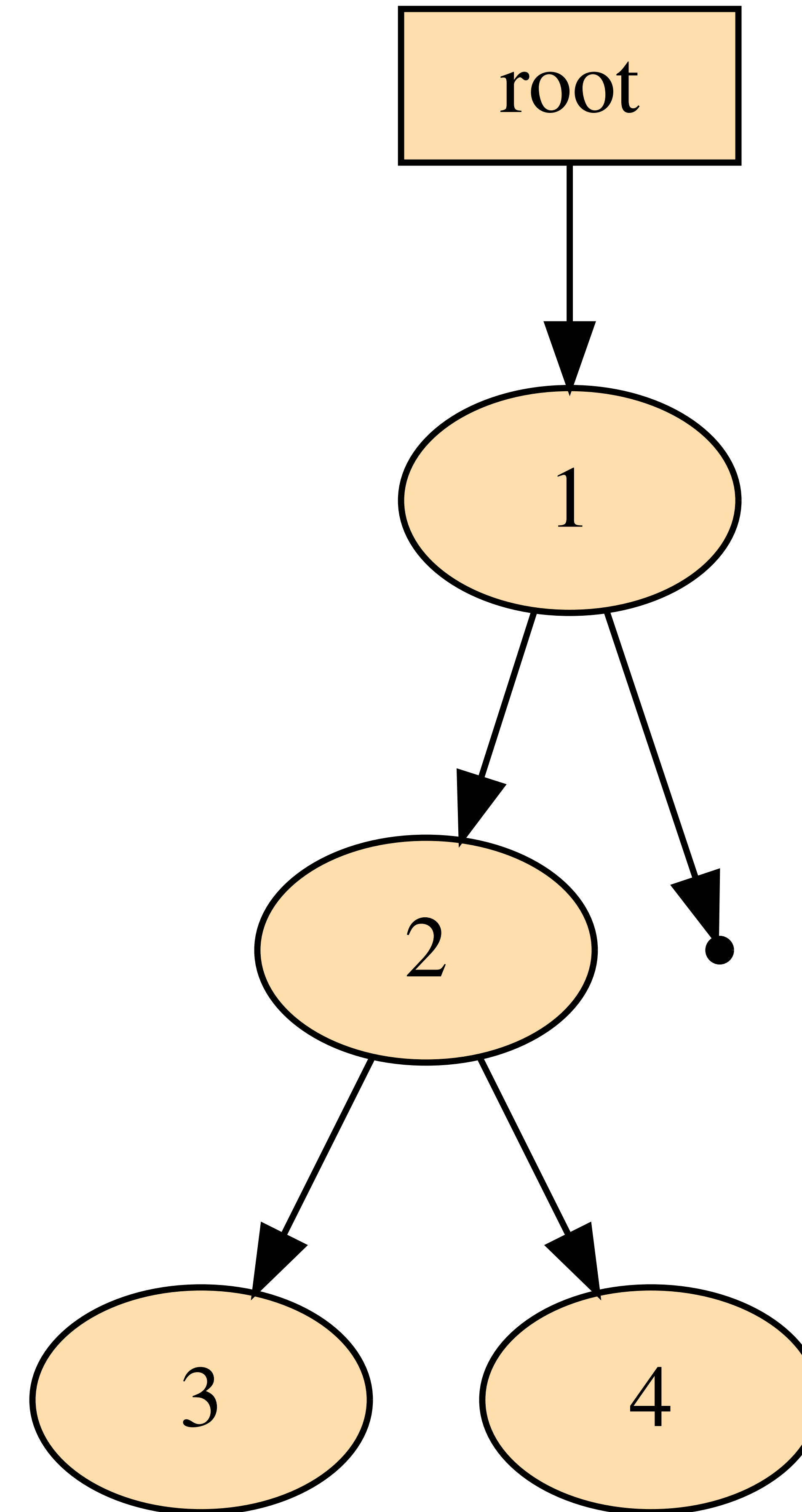
Replay Example

```
1  struct node {
2    node *left;
3    node *right;
4    int value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



Replay Example

```
1  struct node {
2    node *left;
3    node *right;
4    int value;
5  };
6
7  int main() {
8    node *root = make_symbolic_pointer();
9
10   klee_assert(root->value == 1);
11   klee_assert(root->left->value == 2);
12   klee_assert(root->left->left->value == 3);
13   klee_assert(root->left->right->value == 4);
14
15   return 0;
16 }
```



Evaluation on Dynamic Data Structures

Comparison against KLEE 3.0 (LI Off), 30 seconds test timeout, 5 seconds solver timeout, coverage computed by GCOV, "0" means all produced tests cause a segmentation fault

Test	LI Off Branch Coverage (%)	LI On Branch Coverage (%)	Branch Count
avl_balance	0	50	32
avl_insert	0	78.1	46
rb_insert_find	0	94.8	58
rb_remove	0	78.9	90
tree_delete_find	28.9	92.3	26

- Feasible way to test recursive data structures without generating procedures
- Full results: https://gitlab.com/ocelaiwo/klee_li_recursive_datastructs/
- Our fork of KLEE: https://github.com/UnitTestBot/klee
- Test-Comp 2024 (3rd in the Overall category): Misonizhnik, A., Morozov, S., Kostyukov, Y., Kalugin, V., Babushkin, A., Mordvinov, D. and Ivanov, D., 2024, April. KLEEF: Symbolic Execution Engine (Competition Contribution). In International Conference on Fundamental Approaches to Software Engineering (pp. 314-319)