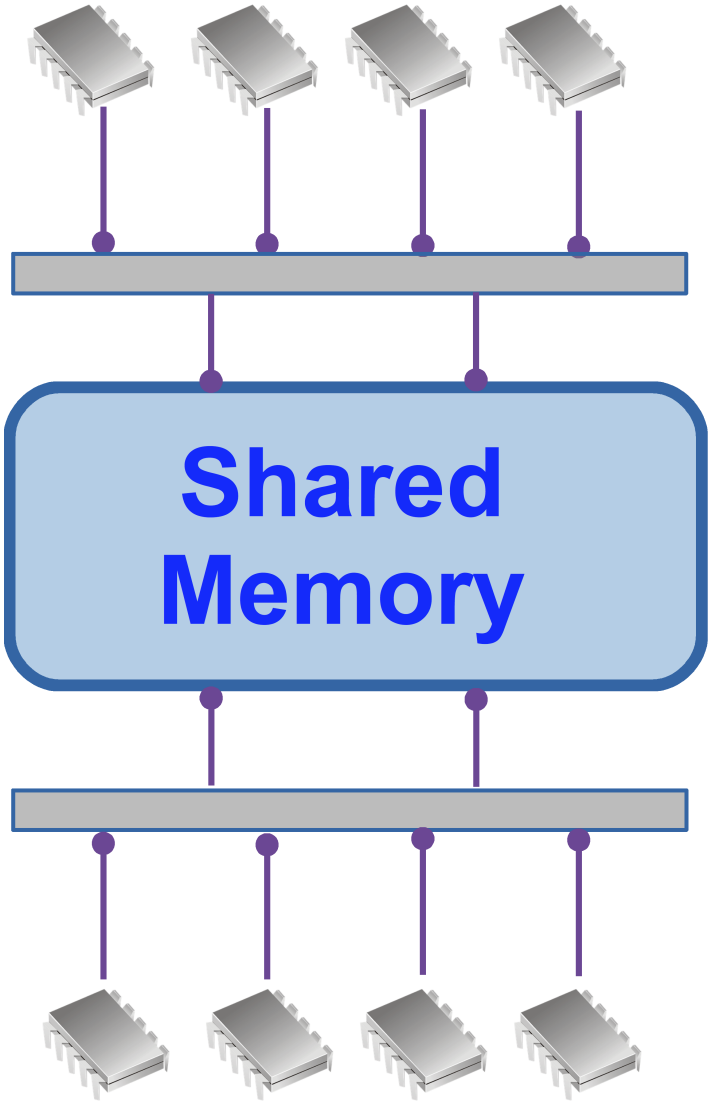
CDSSPEC: Checking Concurrent Data Structures Under the C/C++11 Memory Model

Peizhao Ou and Brian Demsky

University of California, Irvine

Feb 6, 2017

Programming Multi-core Systems



Building Blocks

Concurrent Data Structures

Concurrent Data Structures with Atomics

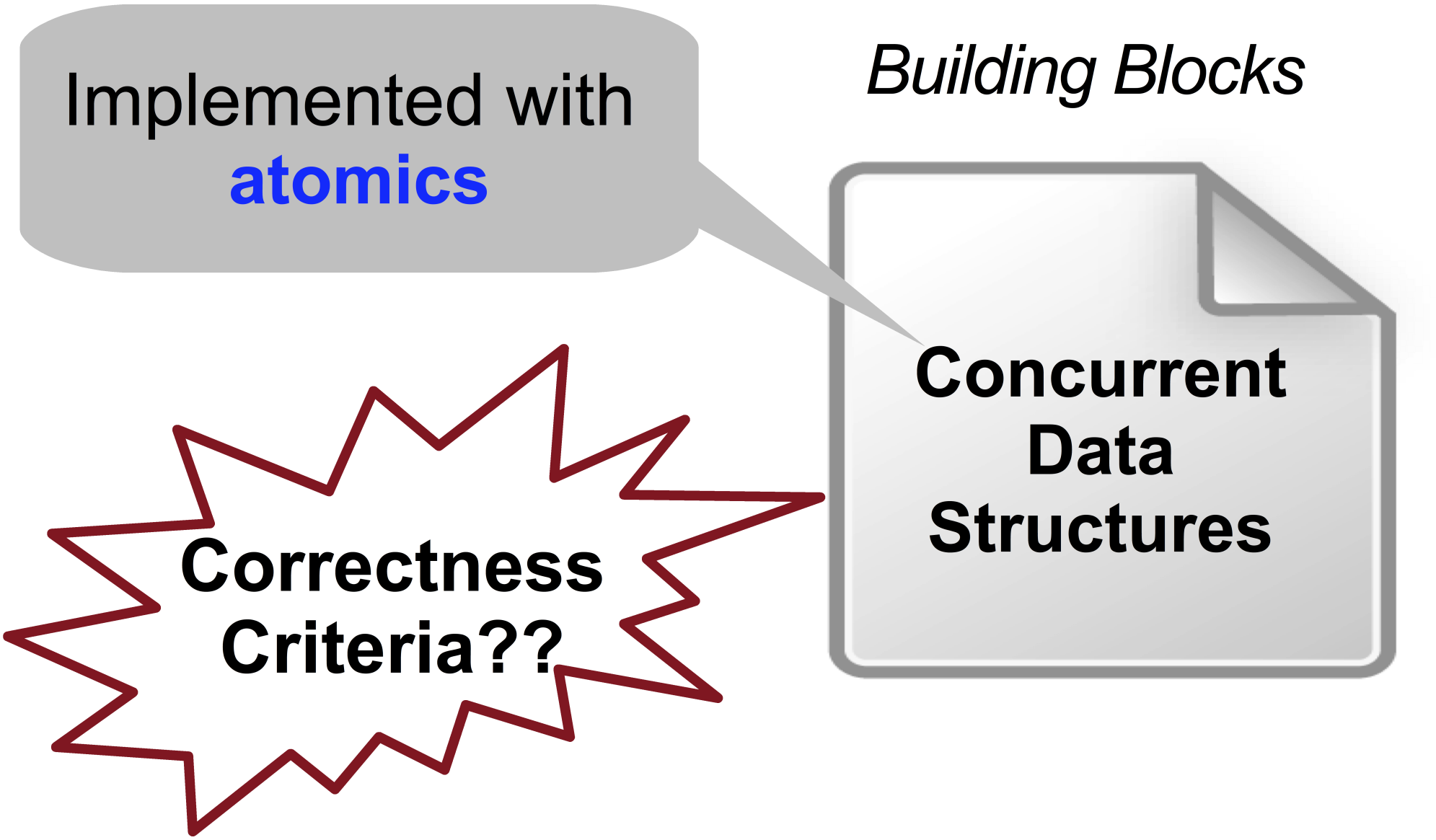
Implemented with **atomics**

Building Blocks

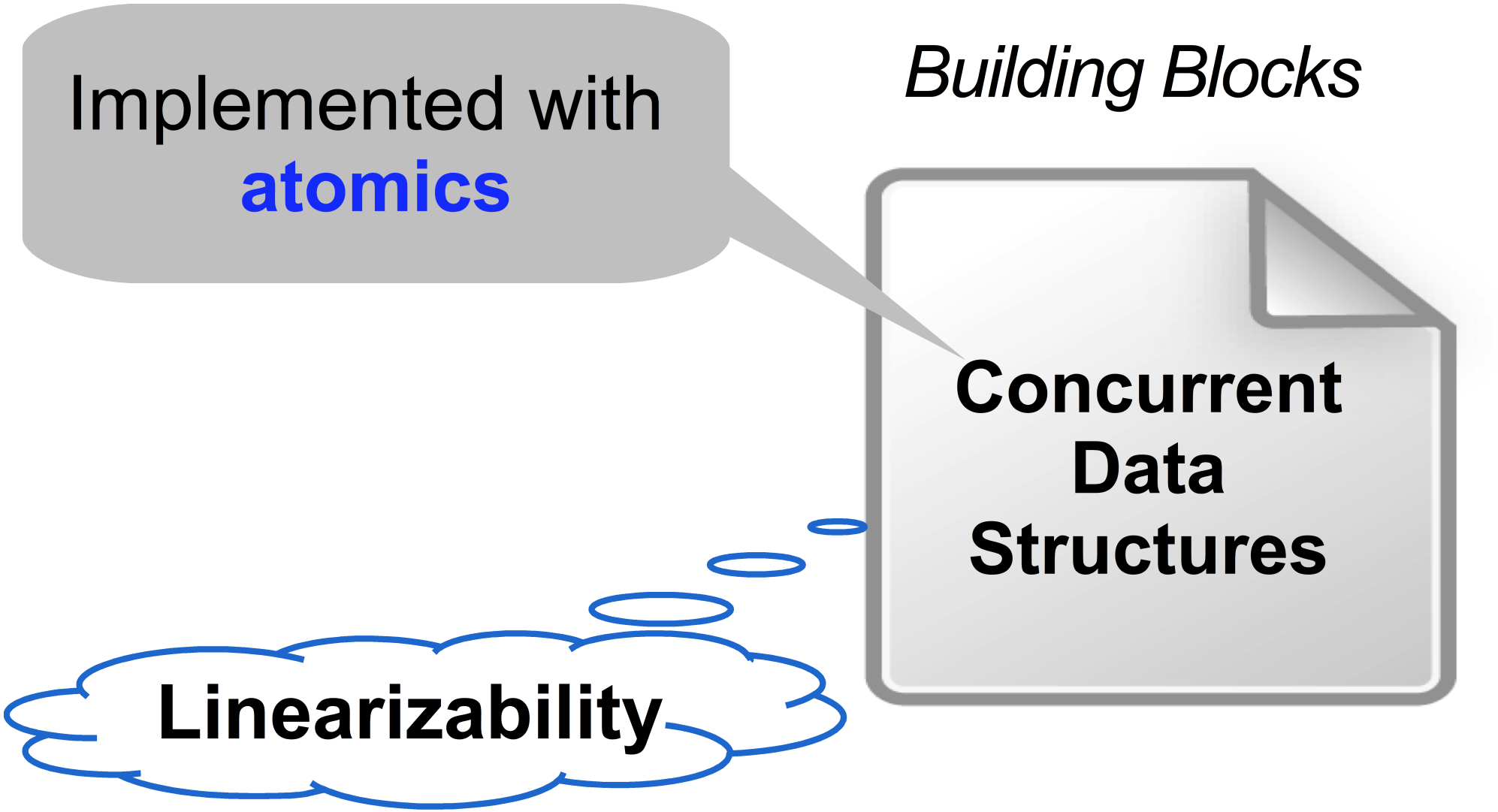
Concurrent Data Structures

Scalability!!

Concurrent Data Structure Correctness



Concurrent Data Structure Correctness

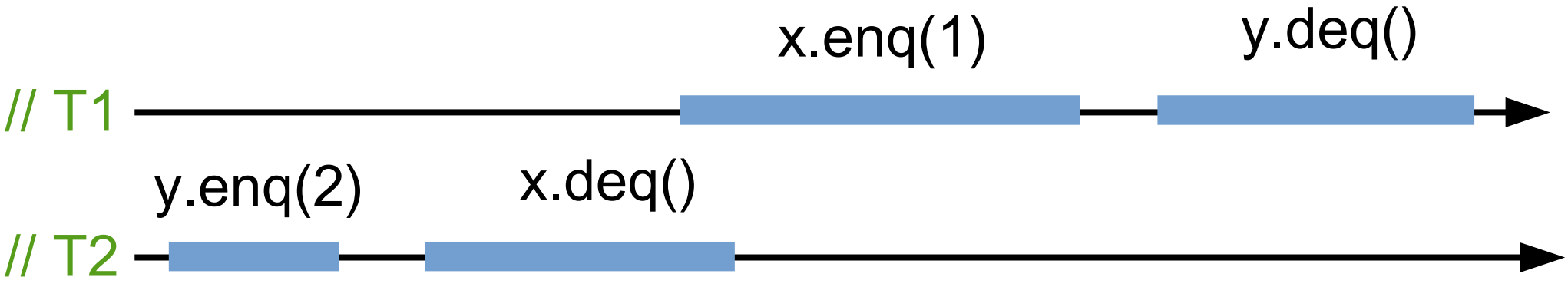


Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```

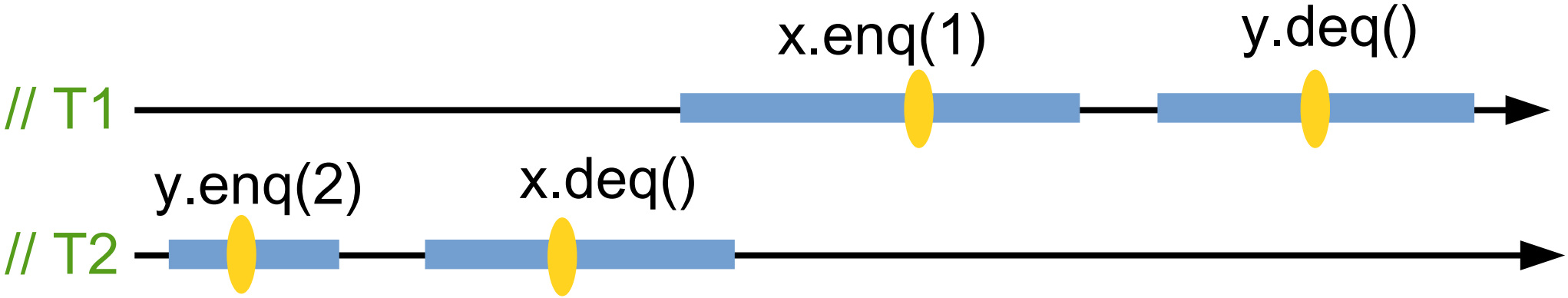
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



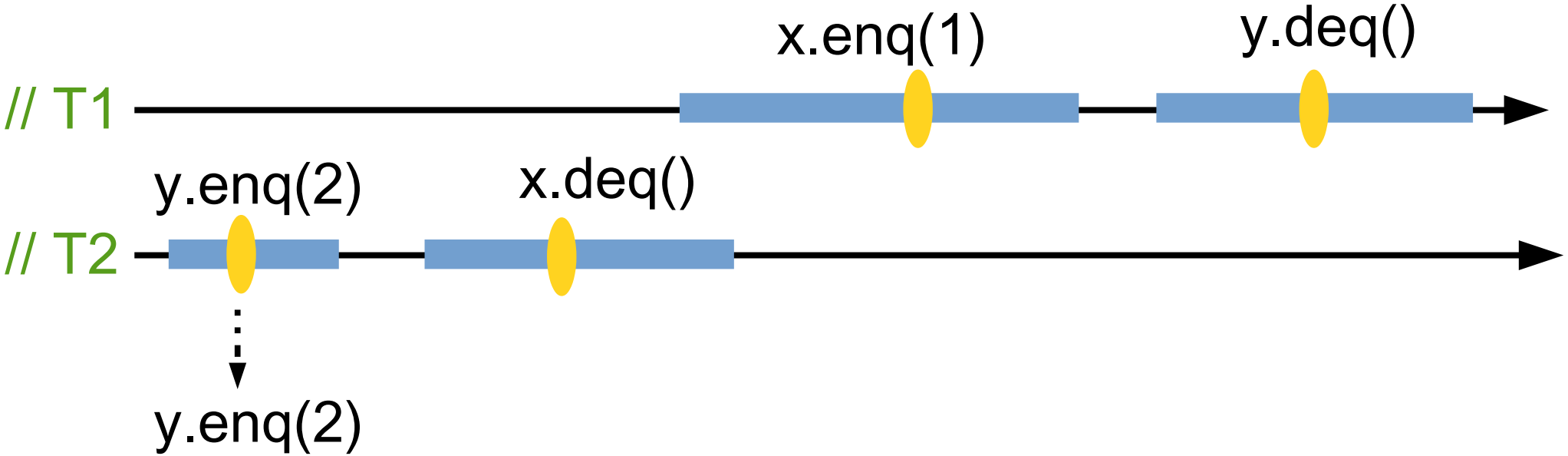
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



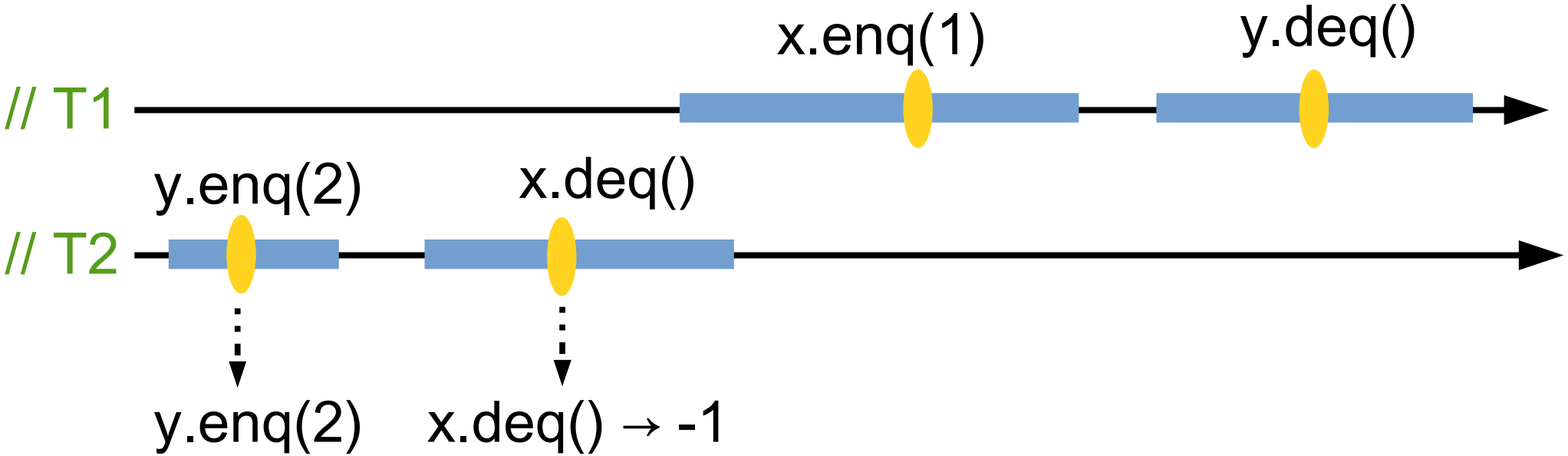
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



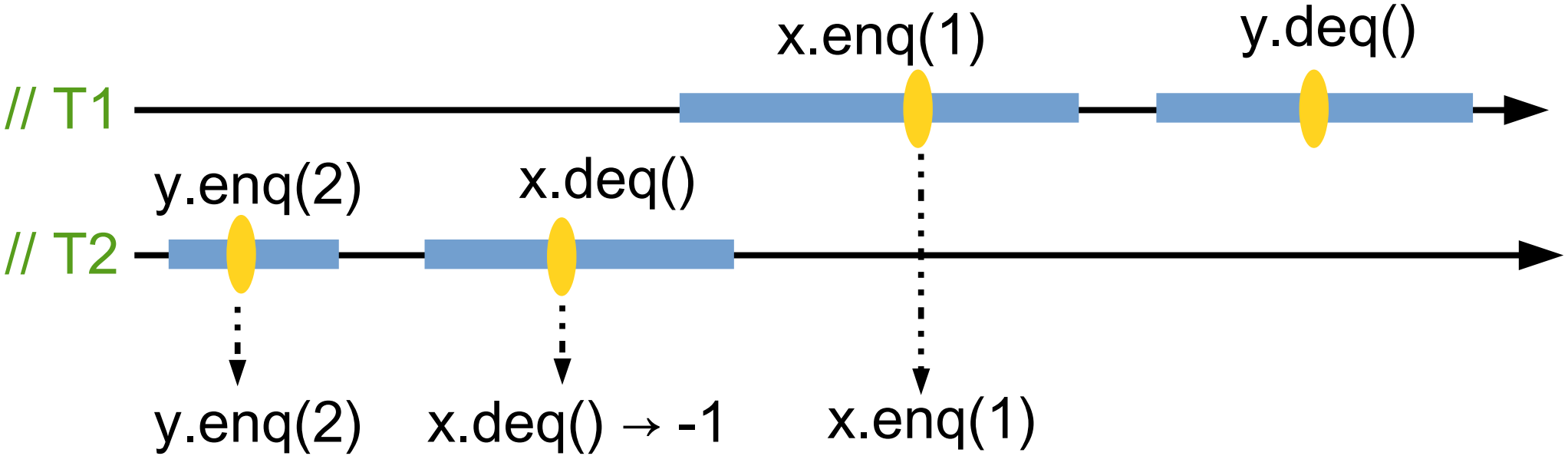
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



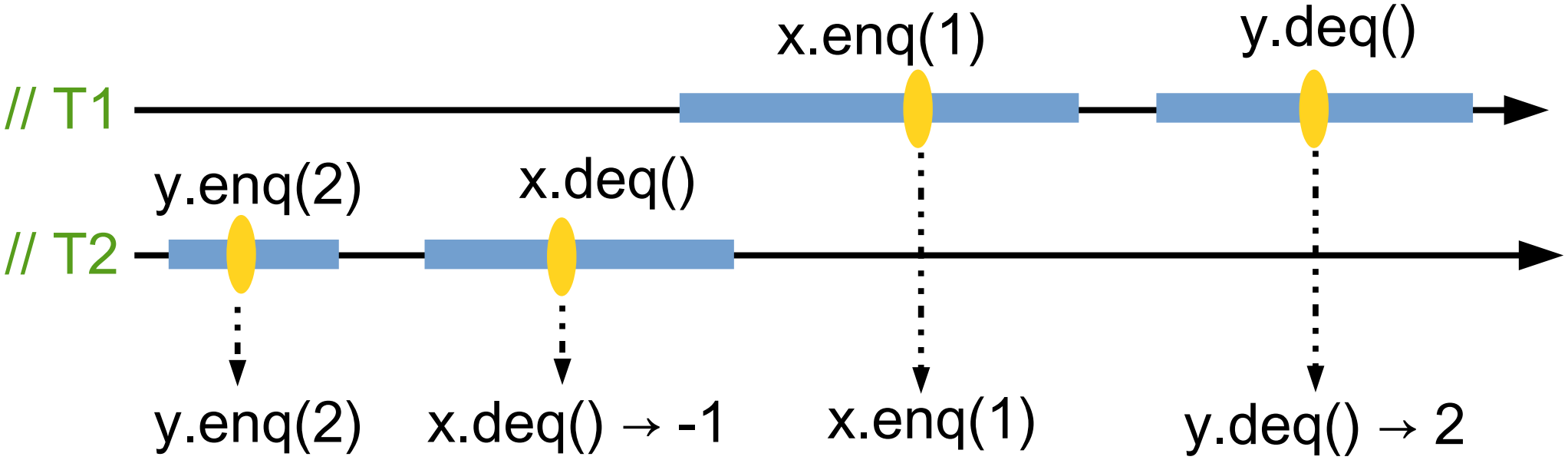
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



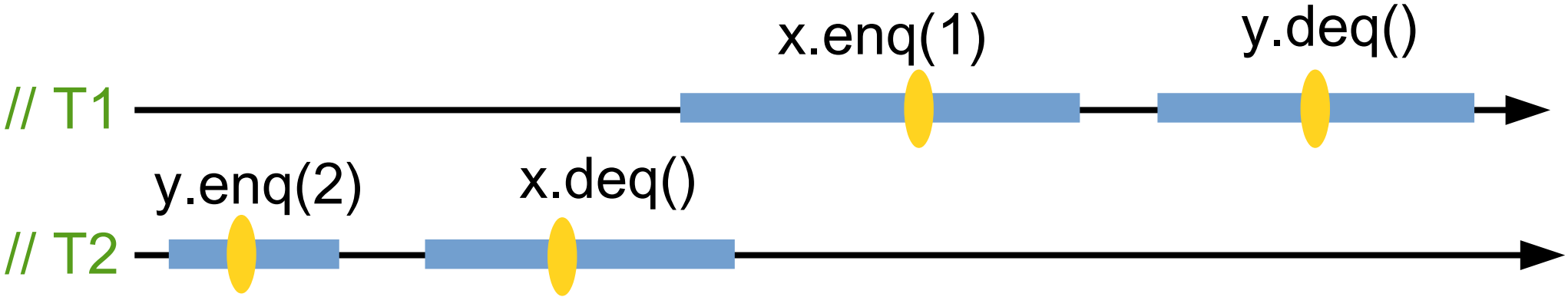
Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



Linearizability Example

```
SomeQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → 2             r2=x.deq(); // → -1
```



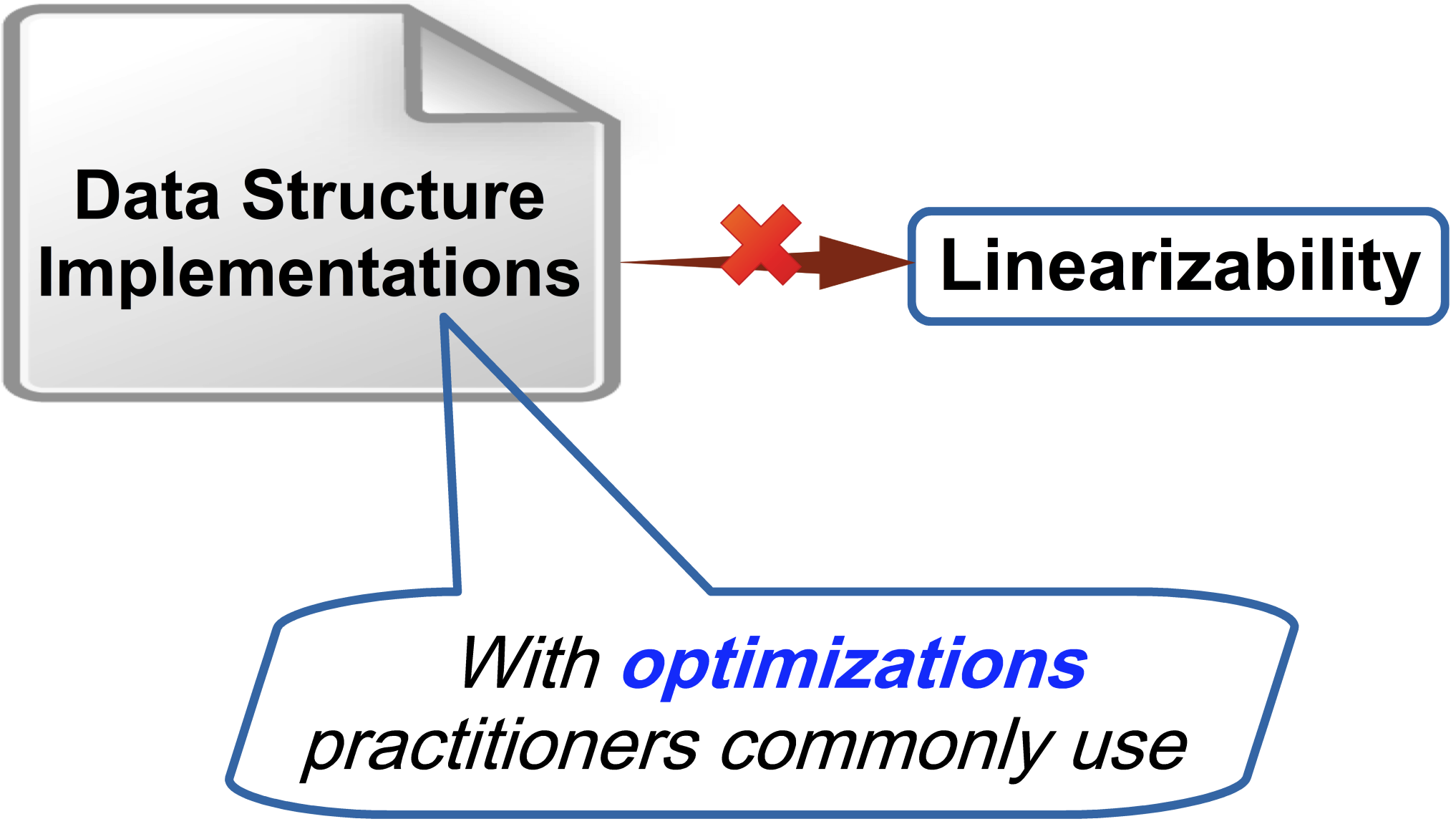
y.enq(2) x.deq() → -1 x.enq(1) y.deq() → 2

Linearizability Highlights

Analogy to sequential executions

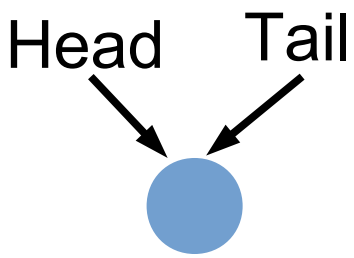
Composability

Optimizations Can Break Linearizability



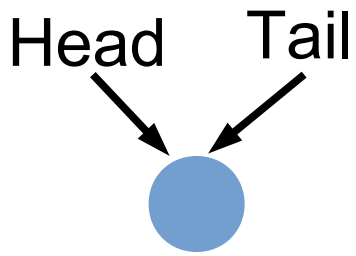
C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



C++11 SPSC Queue Example

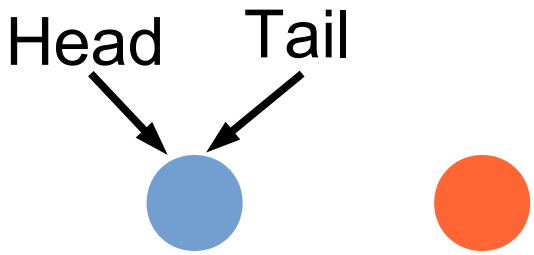
```
struct Node {
    atomic<Node*> next;
    int data;
};
```




```
void enq(int val) {
   Node *n = new Node(val);
   Node *t = Tail.load(relaxed);
   t->next.store(n, release);
   Tail.store(n, relaxed);
}
```

C++11 SPSC Queue Example

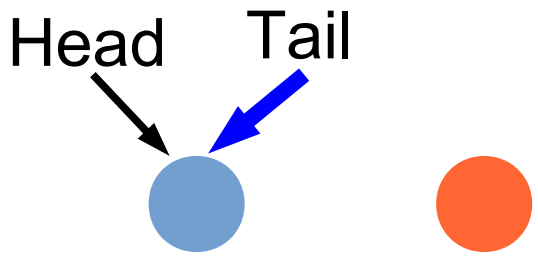
```
struct Node {
    atomic<Node*> next;
    int data;
};
```




```
void enq(int val) {
  Node *n = new Node(val);
  Node *t = Tail.load(relaxed);
  t->next.store(n, release);
  Tail.store(n, relaxed);
}
```

C++11 SPSC Queue Example

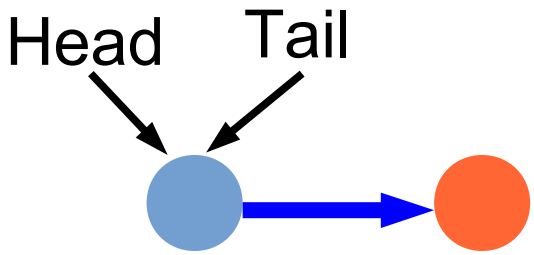
```
struct Node {
    atomic<Node*> next;
    int data;
};
```




```
void enq(int val) {
  Node *n = new Node(val);
  Node *t = Tail.load(relaxed);
  t->next.store(n, release);
  Tail.store(n, relaxed);
}
```

C++11 SPSC Queue Example

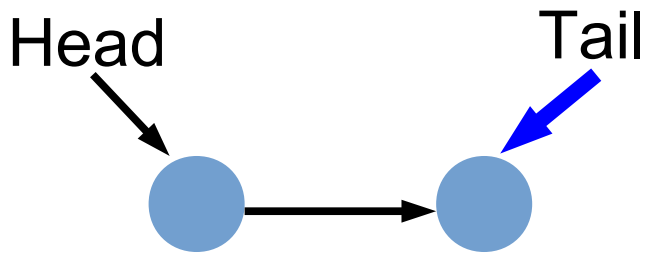
```
struct Node {
    atomic<Node*> next;
    int data;
};
```




```
void enq(int val) {
  Node *n = new Node(val);
  Node *t = Tail.load(relaxed);
  t->next.store(n, release);
  Tail.store(n, relaxed);
}
```

C++11 SPSC Queue Example

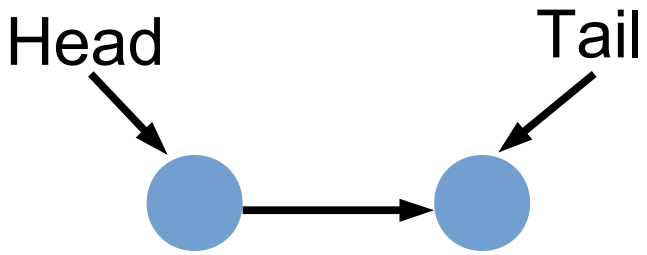
```
struct Node {
    atomic<Node*> next;
    int data;
};
```

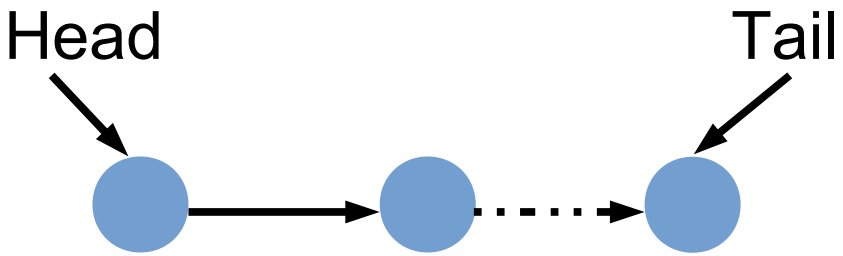



```
void enq(int val) {
   Node *n = new Node(val);
   Node *t = Tail.load(relaxed);
   t->next.store(n, release);
   Tail.store(n, relaxed);
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



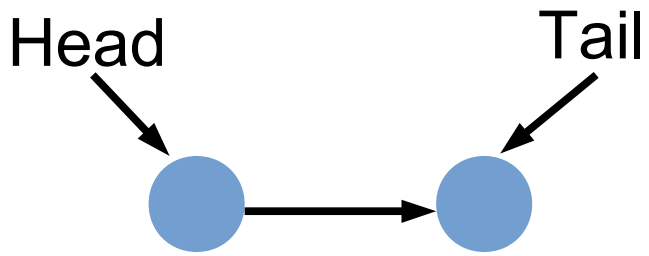


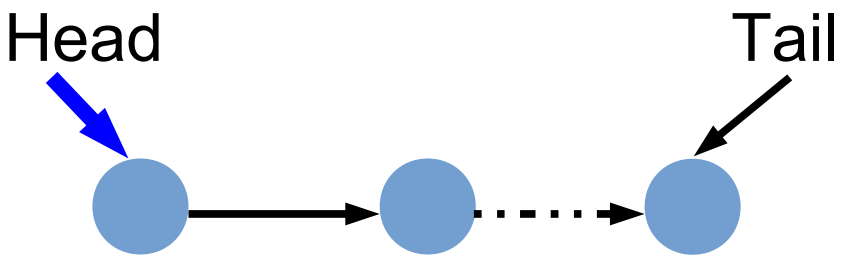

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



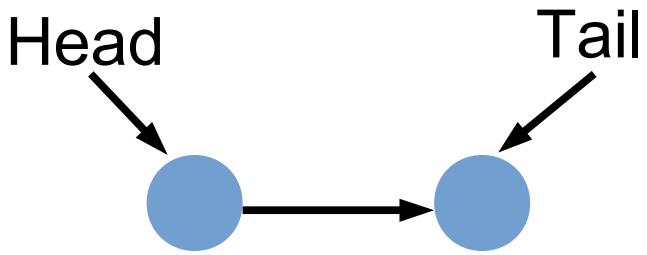


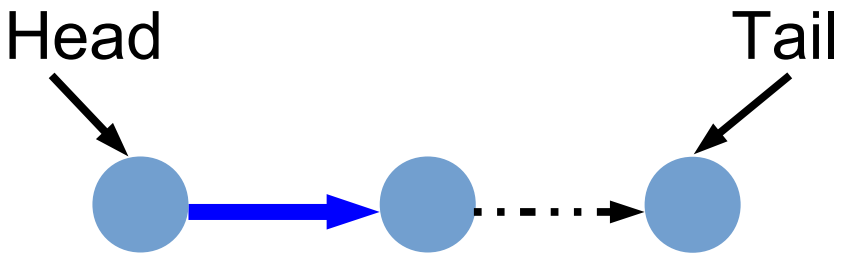

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



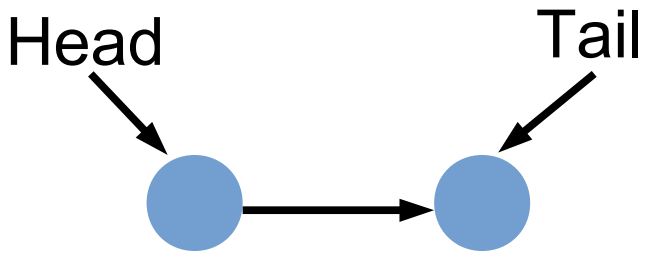


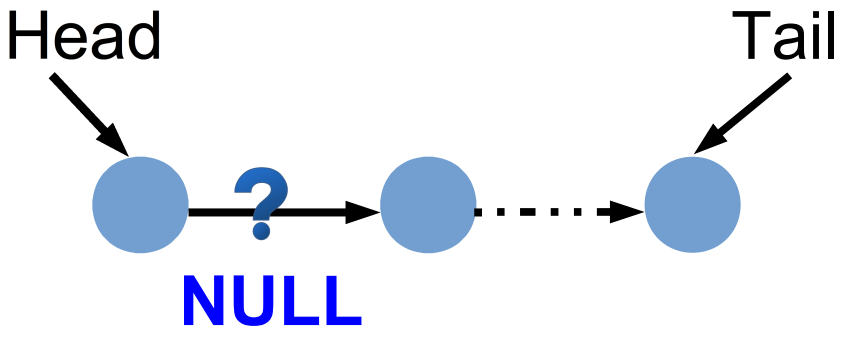

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



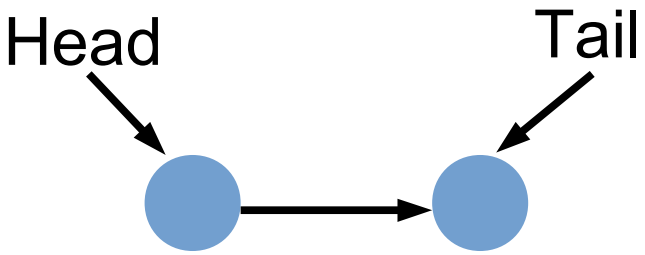


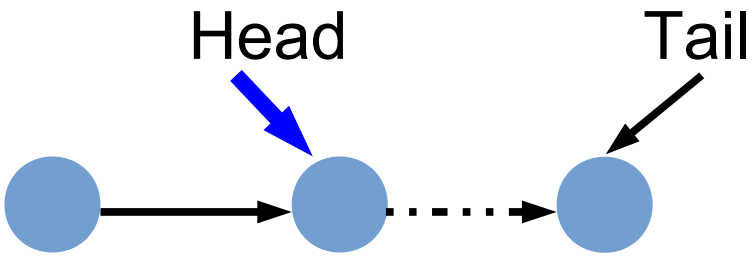

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



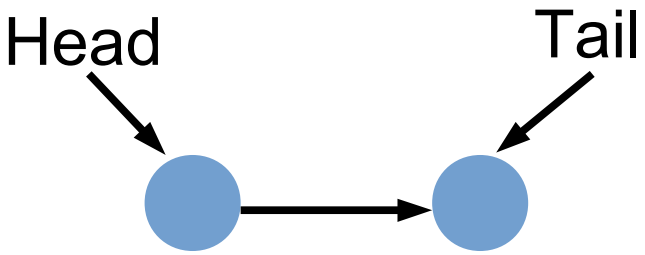


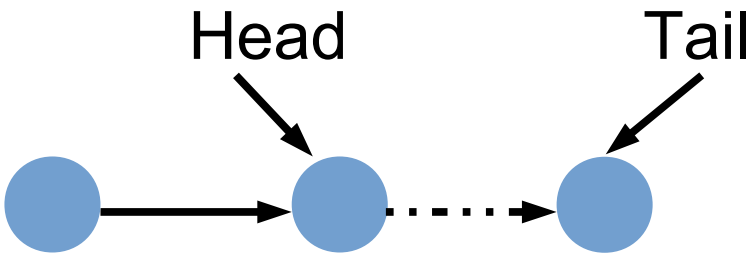

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```



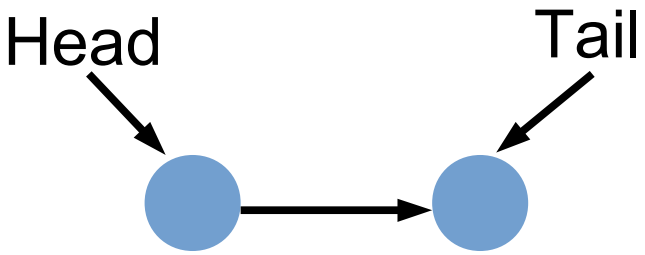


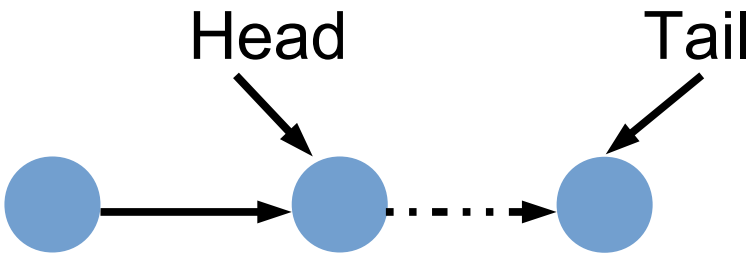

```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

C++11 SPSC Queue Example

```
struct Node {
    atomic<Node*> next;
    int data;
};
```






```
void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

```
int deq() {
    Node *h = Head.load(relaxed),
    Node *n = h->next.load(acquire);
    if (!n)  return -1;
    Head.store(n, relaxed);
    return h->data;
}
```

Release-Acquire for Synchronization

```
// T1
ptr = new Object();
x.enq(ptr);
```


```
// T2
r1 = x.deq();
if (r1 != -1)
    r2 = r1->field;
```

Ensures reading a fully initialized object

Non-linearizable SPSC Execution

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

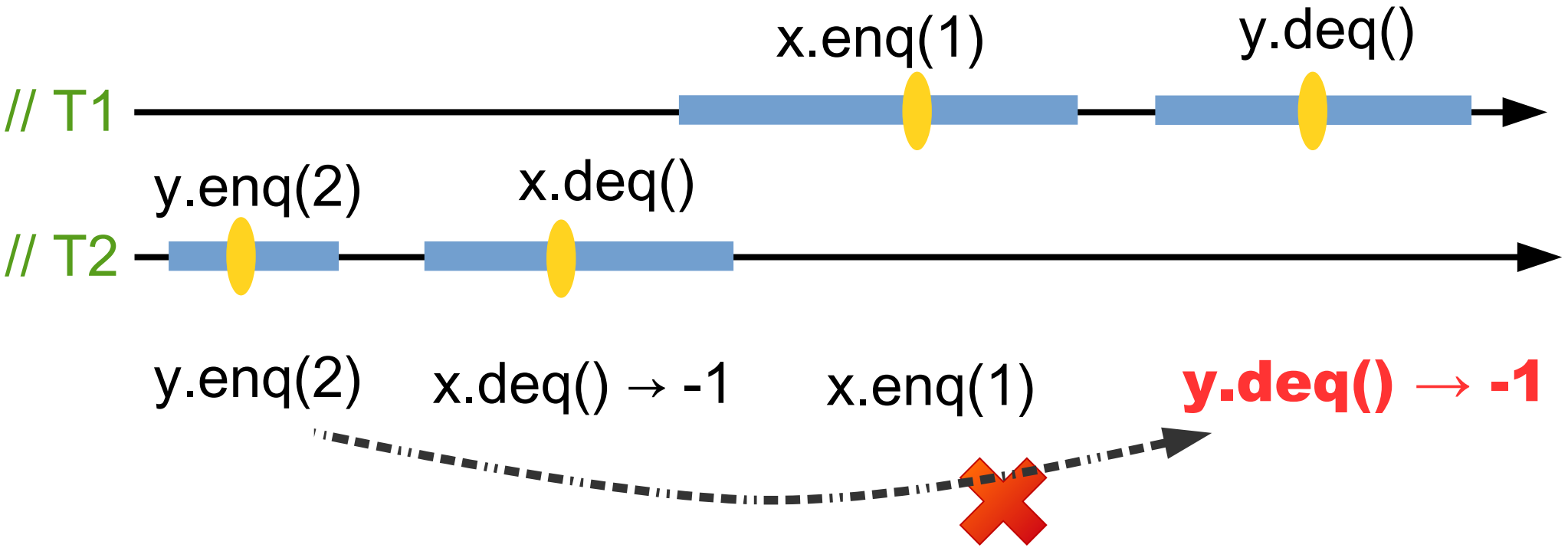
Non-linearizable SPSC Execution

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

Both return empty (-1)

Non-linearizable SPSC Execution

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```



Non-linearizable SPSC Execution

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

Under the hood:

```
// Thread 1                                  // Thread 2
next1.store(n1, release);                    next2.store(n2, release);
…                                               ...
next2->load(acquire); //0                    next1->load(acquire); //0
return -1;                                   return -1;
```

Alternative 1: Use the “seq_cst” ordering

```
SPSCQueue x, y; // Initially empty
// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

Only SC executions (in absence of data races)

Alternative 1: Use the “seq_cst” ordering

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

Only SC executions (in absence of data races)

⟶ Problematic execution not allowed

Alternative 1: Use the “seq_cst” ordering

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

Only SC executions (in absence of data races)

→ Problematic execution not allowed

→ Linearizability applies

→ **Performance loss**

Alternative 2: Constrain Usage Patterns

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

enq() and deq() on the same queue must conflict with each other

Alternative 2: Constrain Usage Patterns

```
SPSCQueue x, y; // Initially empty
// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

enq() and deq() on the same queue must conflict with each other

⟶ Problematic execution not allowed

Alternative 2: Constrain Usage Patterns

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

enq() and deq() on the same queue must conflict with each other

→ Problematic execution not allowed

→ Linearizability applies

→ **Limited usefulness**

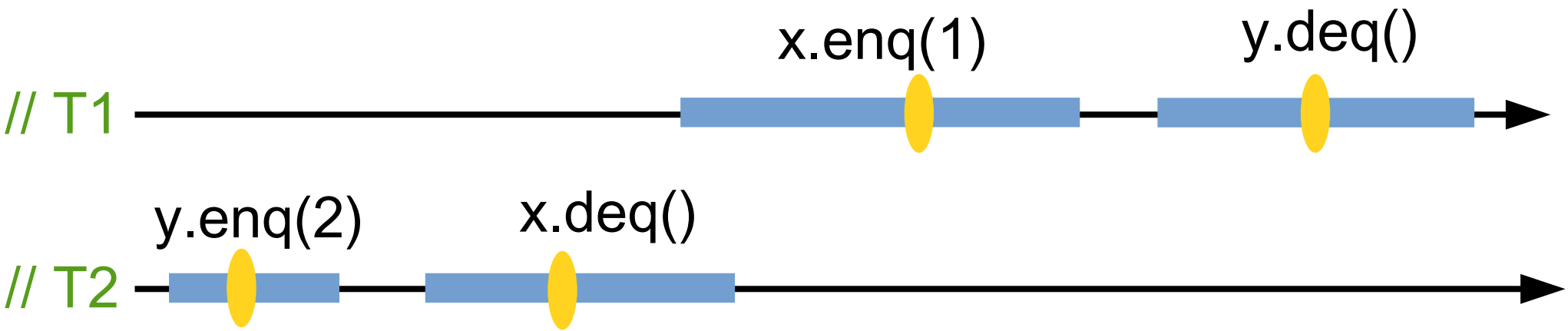
Alternative 3: Weaken the Specification

```
SPSCQueue x, y; // Initially empty
// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```

deq() of the sequential FIFO can spuriously return empty (-1)

Alternative 3: Weaken the Specification

```
SPSCQueue x, y; // Initially empty

// Thread 1                    // Thread 2
x.enq(1);                      y.enq(2);
r1=y.deq(); // → -1            r2=x.deq(); // → -1
```



y.enq(2) x.deq() → -1 x.enq(1) **y.deq() → -1**

Trade-off between Alternatives

	Use strong ordering	Constrain Usage	Weakened Specification
Performance	✗	✓	✓
Usefulness	✓	✗	✓
Clean Semantics	✓	✓	✗

Our Correctness Model

- **Admissibility**
  - Explicit condition under which the data structure's semantics is well defined

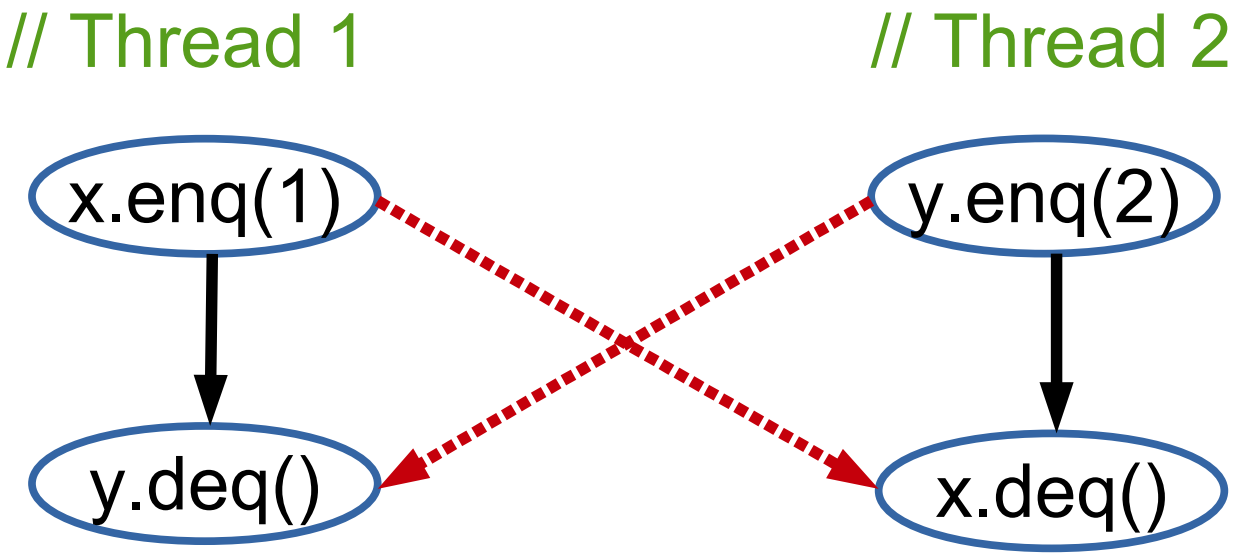
Admissibility – Option 1

enq() and deq() are not required to conflict with each other

Non-deterministic FIFO

Admissibility – Option 2

enq() and deq() must conflict with each other



Deterministic FIFO

Our Correctness Model

- **Non-deterministic specification**
  - Concurrent execution → equivalent sequential execution
  - Non-deterministic equivalent sequential data structure

Our Correctness Model

- **Non-deterministic specification**
  - Concurrent execution → equivalent sequential execution
  - Non-deterministic equivalent sequential data structure

A sequential FIFO, whose deq() operation can spuriously return -1

Problem: Too Weak Specification

```
// Same thread
x.enq(1);
r1=x.deq(); // → Spuriously return -1
```

→ **The spec can become too weak**

→ Can go beyond developer's expectation

Solution: Tighten the Weakened Spec

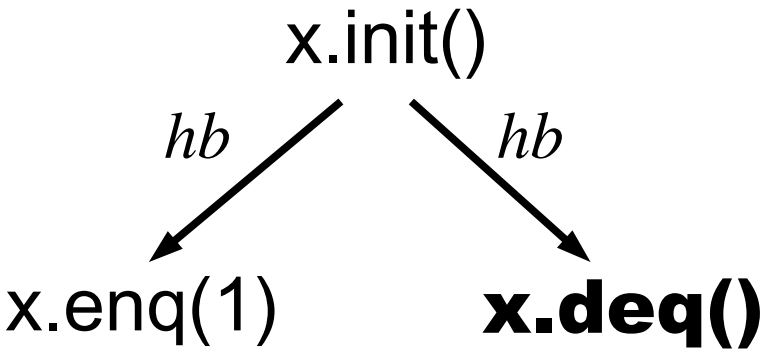
```
// Same thread
x.enq(1);
r1=x.deq(); // → Spuriously return -1
```

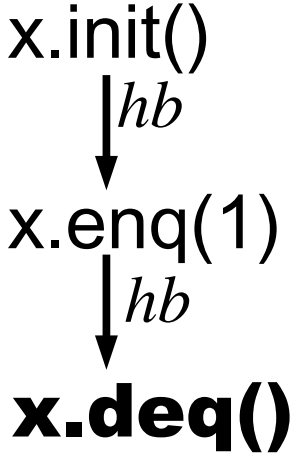
Specify when deq() can spuriously return empty (-1)!!

Tighten Spec with Justifying Prefix

Justifying prefix

→ sequence of method calls that are ordered by *happens-before* and that *happen before* a method call m

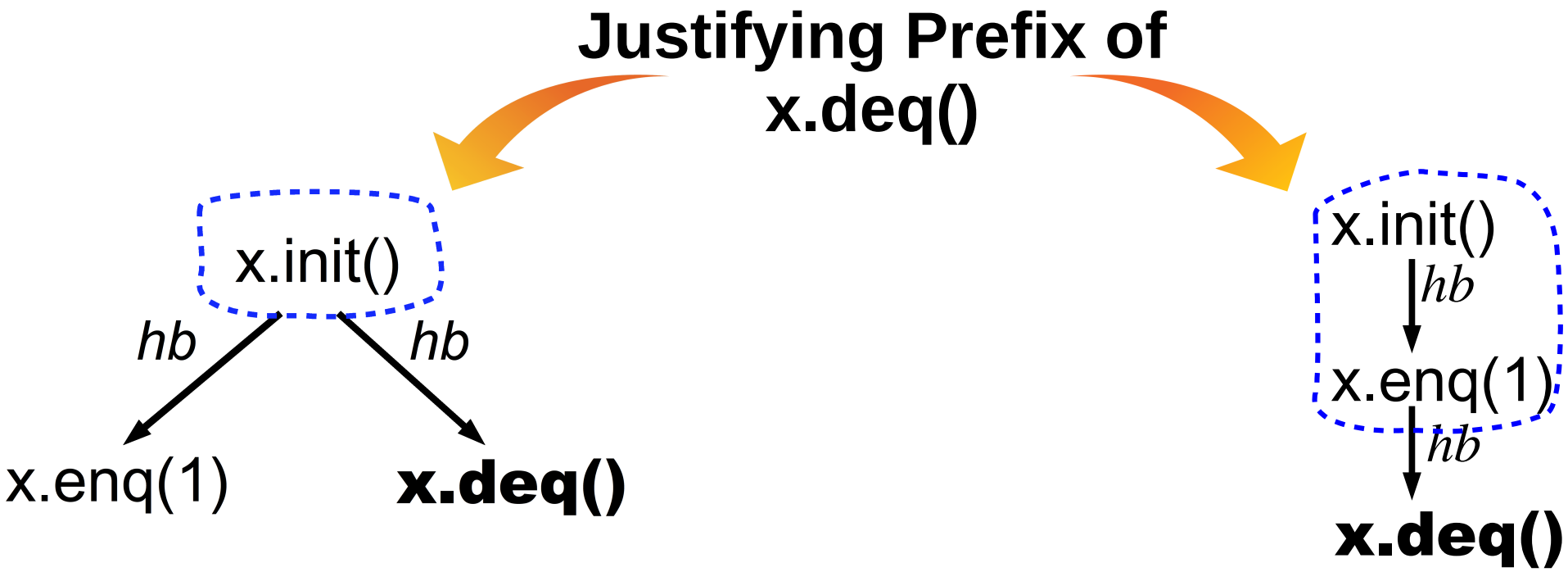




Tighten Spec with Justifying Prefix

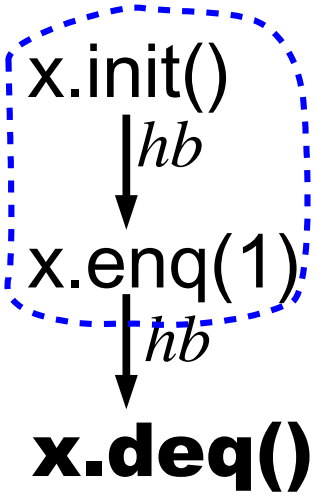
Justifying prefix

→ sequence of method calls that are ordered by *happens-before* and that *happen before* a method call m



Tighten Spec with Justifying Prefix

deq() spuriously return -1 **only when** one of its justifying prefix empties the queue

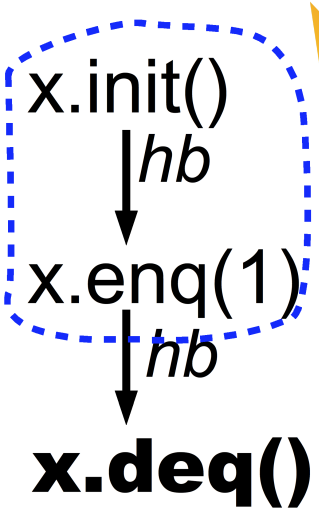



```
// Same thread
x.enq(1);
r1=x.deq(); // → Return -1
```

Tighten Spec with Justifying Prefix

deq() spuriously return -1 **only when** one of its justifying prefix empties the queue

Justifying prefix enqueues 1 to the queue




```
// Same thread
x.enq(1);
r1=x.deq(); // → Return -1
```

Non-deterministic Linearizability

- Admissibility
- Non-deterministic specification
- **Constrain non-determinism with justifying prefix**

Composability!

CDSSPEC Specification Language

- A concurrent data structure specification language
  - Based on non-deterministic linearizability

Admissibility

```
/** @Admit: deq <-> deq (true);
@Admit: enq <-> enq (true) */
```

```
void enq(int val) {
   Node *n = new Node(val);
   Node *t = Tail.load(relaxed);
   t->next.store(n, release);
   Tail.store(n, relaxed);
}
```

Equivalent Sequential Data Structure

```
/** @DeclareState: IntList *q; */

void enq(int val) {
    Node *n = new Node(val);
    Node *t = Tail.load(relaxed);
    t->next.store(n, release);
    Tail.store(n, relaxed);
}
```

Ordering Points to Order Method Calls

```
/** @DeclareState: IntList *q; */

/** … */
int deq() {
  Node *h = Head.load(relaxed),
    *n = h->next.load(acquire);
    /** @OPDefine: true */
  if (!n)  return -1;
  Head.store(n, relaxed);
  return h->data;
}
```

Dequeue – Side Effect

```
/** @DeclareState: IntList *q; */

/** @SideEffect: if (C_RET==-1)  return;
S_RET = STATE(q)->pop_front();

*/
int deq() {
   Node *h = Head.load(relaxed),
      *n = h->next.load(acquire);
   if (!n)  return -1;
   Head.store(n, relaxed);
   return h->data;
}
```

Dequeue – Postcondition

```
/** @DeclareState: IntList *q; */

/** ...
@PostCondition: return C_RET==-1 || C_RET==S_RET;
*/
int deq() {
   Node *h = Head.load(relaxed),
      *n = h->next.load(acquire);
   if (!n)  return -1;
   Head.store(n, relaxed);
   return h->data;
}
```

CDSSPEC Checker

- **Back-end analysis** of the CDSChecker model checker
  - Exhaustively check a given test case against CDSSPEC specifications (under some constraints)

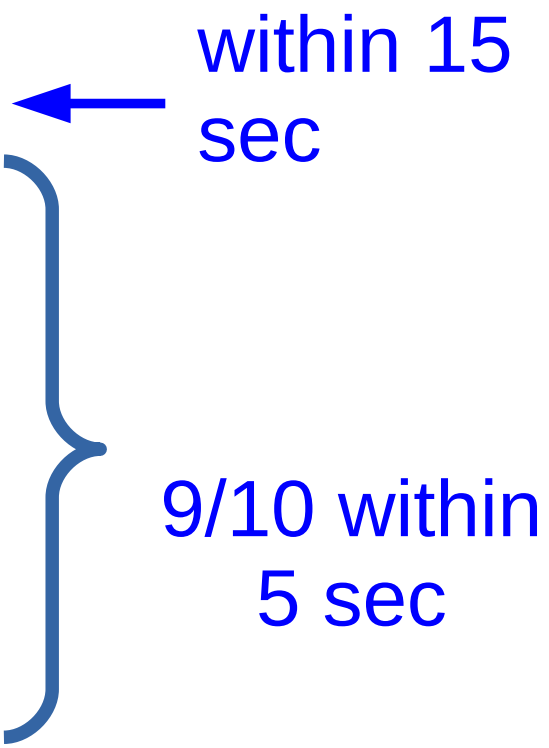
Expressiveness of CDSSPEC

- **10 real-world data structures**
  - 3 concurrent queues: SPSC, M&S queue & MPMC
  - 4 locks: Linux RW lock, Seqlock, MCS lock & Ticket lock
  - A read-copy-update implementation
  - Chase-Lev deque
  - Concurrent hashtable

CDSSEPEC Checker Performance

- Ubuntu 14.04 (Intel Xeon E3-1246 v3)

Benchmarks	Total Time (sec)
Linux RW lock	13.71
MPMC queue	4.83
MCS lock	3.00
Ticket lock	0.17
Chase-Lev deque	0.10
M&S queue	0.03
SPSC queue	0.01
Seqlock	0.01
RCU	0.01
Concurrent hashtable	0.01



Finding Know Bugs

- Found 3 known bugs in 2 benchmarks
  - Weaker than necessary ordering parameters

Finding Injected Bugs

Benchmarks	# Injection	# Built-in	# CDSSpec
MPMC queue	8	0	4
Linux RW lock	8	0	8
MCS lock	8	4	4
Ticket lock	2	0	2
Chase-Lev deque	7	3	4
M&S Queue	10	3	7
SPSC queue	2	0	2
Seqlock	5	0	5
RCU	3	3	0
Hashtable	4	2	2
Total	57	15	38

Detected by writing specifications

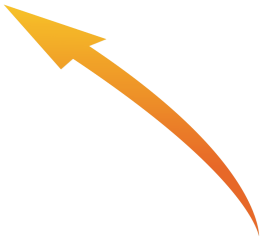
Finding Injected Bugs

Benchmarks	# Injection	# Built-in	# CDSSpec	# Rate
MPMC queue	8	0	4	50%
Linux RW lock	8	0	8	100%
MCS lock	8	4	4	100%
Ticket lock	2	0	2	100%
Chase-Lev deque	7	3	4	100%
M&S Queue	10	3	7	100%
SPSC queue	2	0	2	100%
Seqlock	5	0	5	100%
RCU	3	3	0	100%
Hashtable	4	2	2	100%
Total	57	15	38	93%

100% for 9/10 benchmarks

Finding Injected Bugs

Benchmarks	# Injection	# Built-in	# CDSSpec	# Rate
MPMC queue	8	0	4	50%
Linux RW lock	8	0	8	100%
MCS lock	8	4	4	100%
Ticket lock	2	0	2	100%
Chase-Lev deque	7	3	4	100%
M&S Queue	10	3	7	100%
SPSC queue	2	0	2	100%
Seqlock	5	0	5	100%
RCU	3	3	0	100%
Hashtable	4	2	2	100%
Total	57	15	38	93%



>100,000 threads & a 16-bit counter rollover

Ease of Use

- On average:
  - 11.5 lines per data structure
  - 1.22 lines per API method for ordering points

Related Work

- **Concurrent data structure specifications**
  - Refinement mapping, Commit atomicity, Concurrit, NDetermin
  - Relaxed memory model: Batty *el al.*, Tassarotti *el al.*
  - Bounded relaxation
- **Approaches based on linearizability**
  - Linearizability, Lineup, Paraglider, VYRD,
  - Techniques to automatically prove linearizability
  - Others: Concurrency-aware objects, list-based set
- **Enforce code to only admit SC behaviors**
  - Under TSO & PSO (Burckhardt *el al.,* Burnim *el al.*
  - For C/C++11: Meshman *el al.*, AutoMO
  - Dfence (infer fences for hardware memory model)
- **Others:** GAMBIT, RELAXED, CheckFence

Conclusion

- **CDSSPEC**

  A specification checker that allows developers to specify and check a range of concurrent data structures written with C/C++11

Questions

Enqueue – Side Effect

```
/** @DeclareState: IntList *q; */

/** @SideEffect: STATE(q)->push_back(val); */
void enq(int val) {
   Node *n = new Node(val);
   Node *t = Tail.load(relaxed);
   t->next.store(n, release);
   Tail.store(n, relaxed);
}
```

Ordering Points to Order Method Calls

```
/** @DeclareState: IntList *q; */

/** ... */

void enq(int val) {

   Node *n = new Node(val);
   Node *t = Tail.load(relaxed);
   t->next.store(n, release);
   /** @OPDefine: true */
   Tail.store(n, relaxed);
}
```

Dequeue – Justifying Postcondition

```
/** @DeclareState: IntList *q; */

/** ...;
@JustifyingPostcondition: if (C_RET==-1) return S_RET == -1;
*/
int deq() {
   Node *h = Head.load(relaxed),
      *n = h->next.load(acquire);
   if (!n)  return -1;
   Head.store(n, relaxed);
   return h->data;
}
```