Linearizability Checking:
Reductions to State Reachability

Ahmed Bouajjani
Univ Paris Diderot - Paris 7

Joint work with

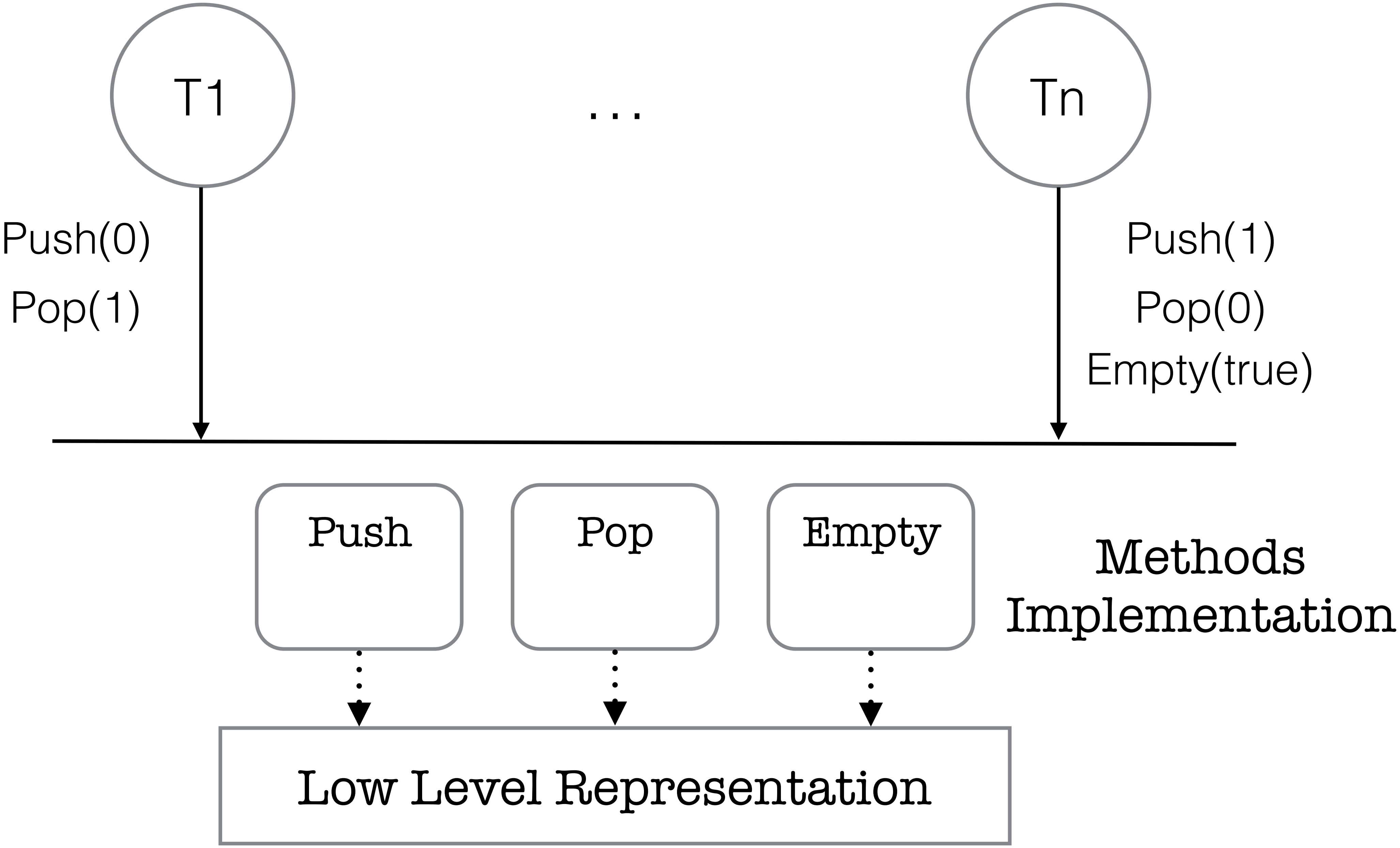
Michael Emmi
Bell Labs, Nokia

Constantin Enea
Univ Paris Diderot

Jad Hamza
EPFL

IMS-NUS, Singapore, September 20, 2016

Concurrent Data Structures



T1

…

Tn

Push(0)

Pop(1)

Push(1)

Pop(0)

Empty(true)

Push

Pop

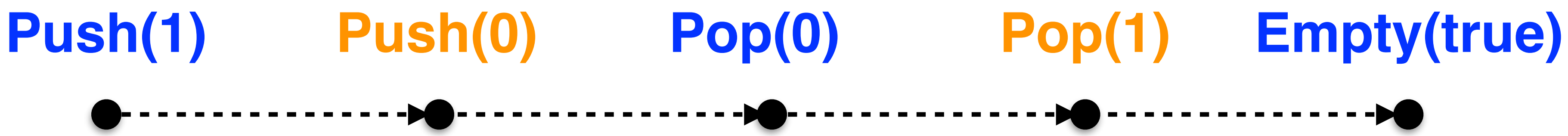
Empty

Methods Implementation

Low Level Representation

Abstract (Client) View

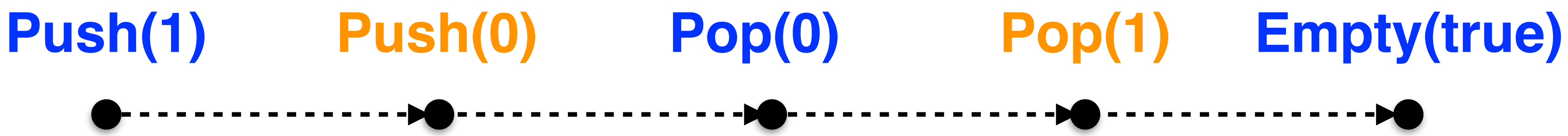
- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications



Push(1) **Push(0)** **Pop(0)** **Pop(1)** **Empty(true)**

Abstract (Client) View

- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications



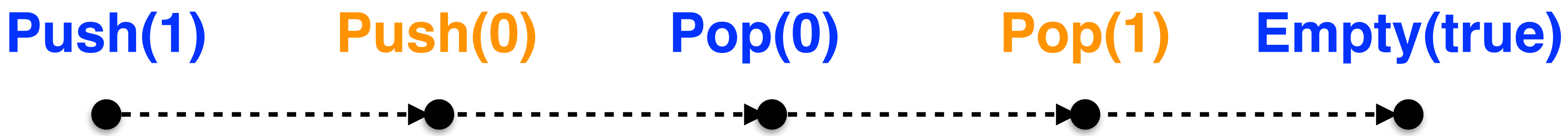
Push(1) **Push(0)** **Pop(0)** **Pop(1)** **Empty(true)**

A “simple” implementation: **Coarse-grain Locking**

- Take a sequential implementation
- **Lock** at the beginning, **unlock** at the end of each method

Abstract (Client) View

- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications


Push(1) **Push(0)** **Pop(0)** **Pop(1)** **Empty(true)**

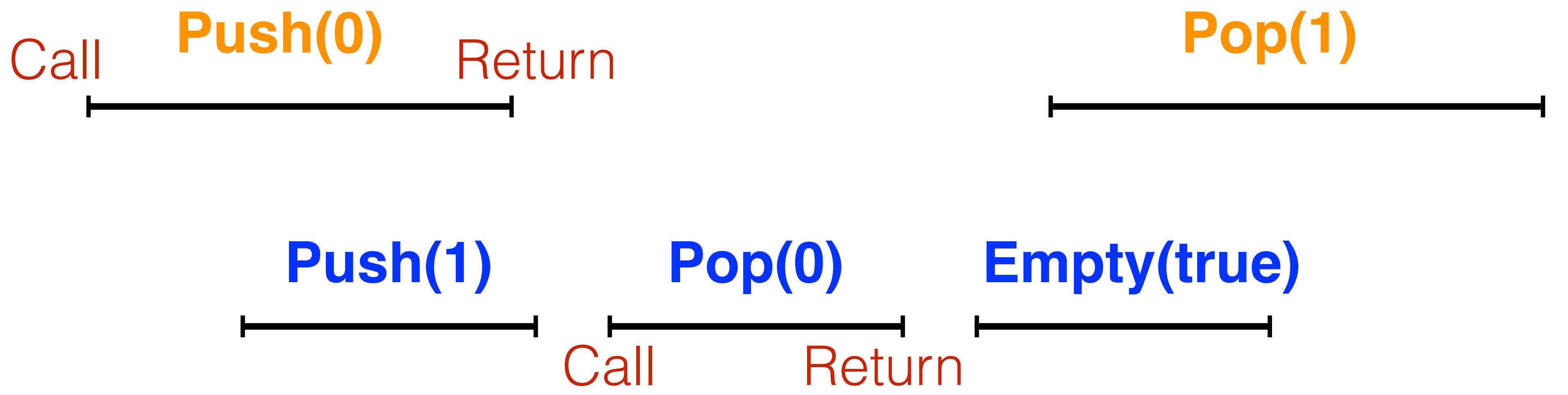
A "simple" implementation: **Coarse-grain Locking**

- Take a sequential implementation
- **Lock** at the beginning, **unlock** at the end of each method

- + **Reference Implementation**: simple to understand
- - **Low performances**

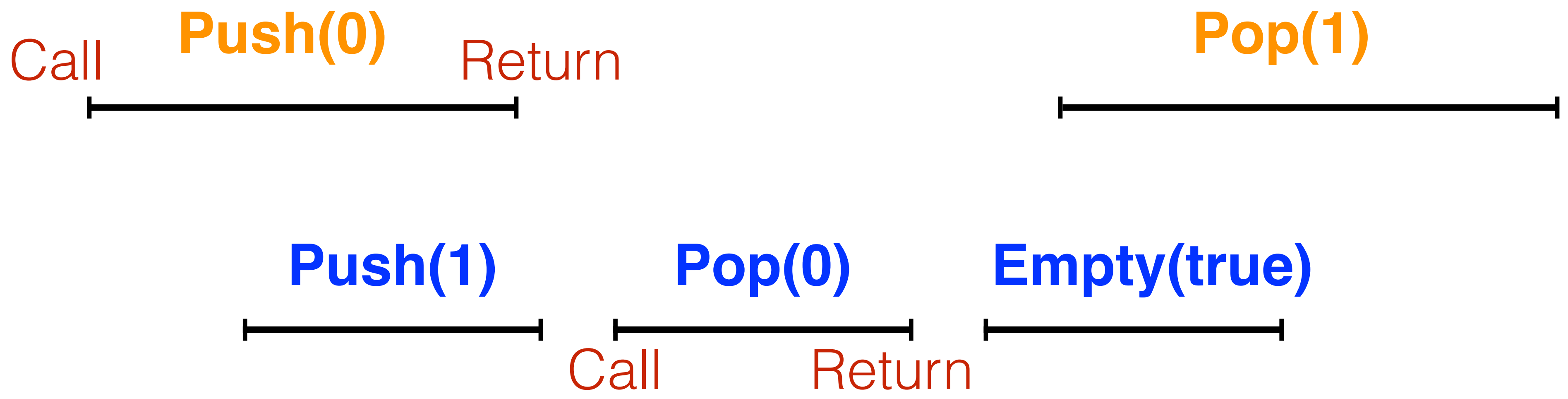
Efficient Concurrent Implementations

Allow **parallelism** between operations



Efficient Concurrent Implementations

Allow **parallelism** between operations



Call **Push(0)** Return

Pop(1)

Push(1) **Pop(0)** **Empty(true)**

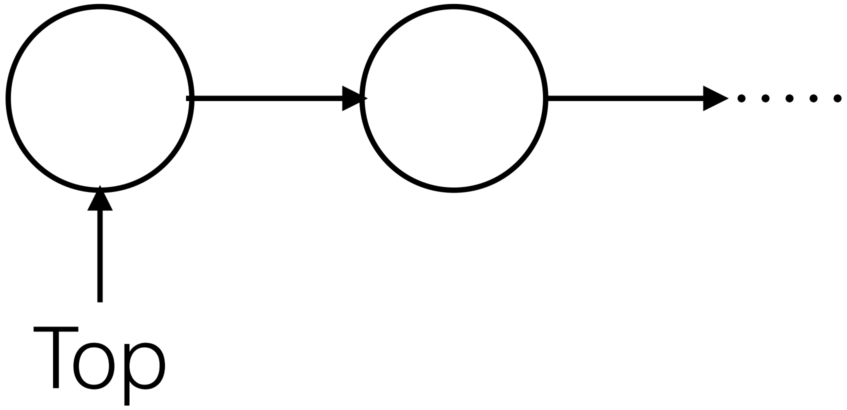
Call Return

Fine-grain locking (Lock-free algorithms)

- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)

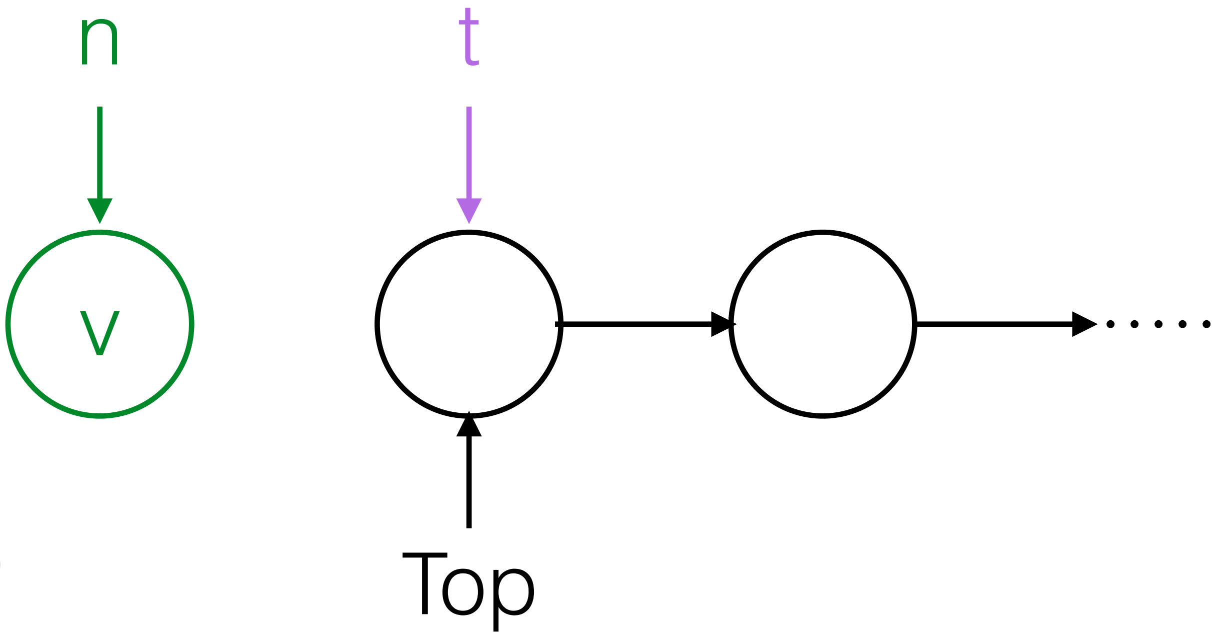
Treiber’s Stack

```
void Push (int v) {
        node *n, *t
        node n = new node(v)
        do {
                node *t = Top
                n.next = t
        } while (not CAS(&Top, t, n))
}
```



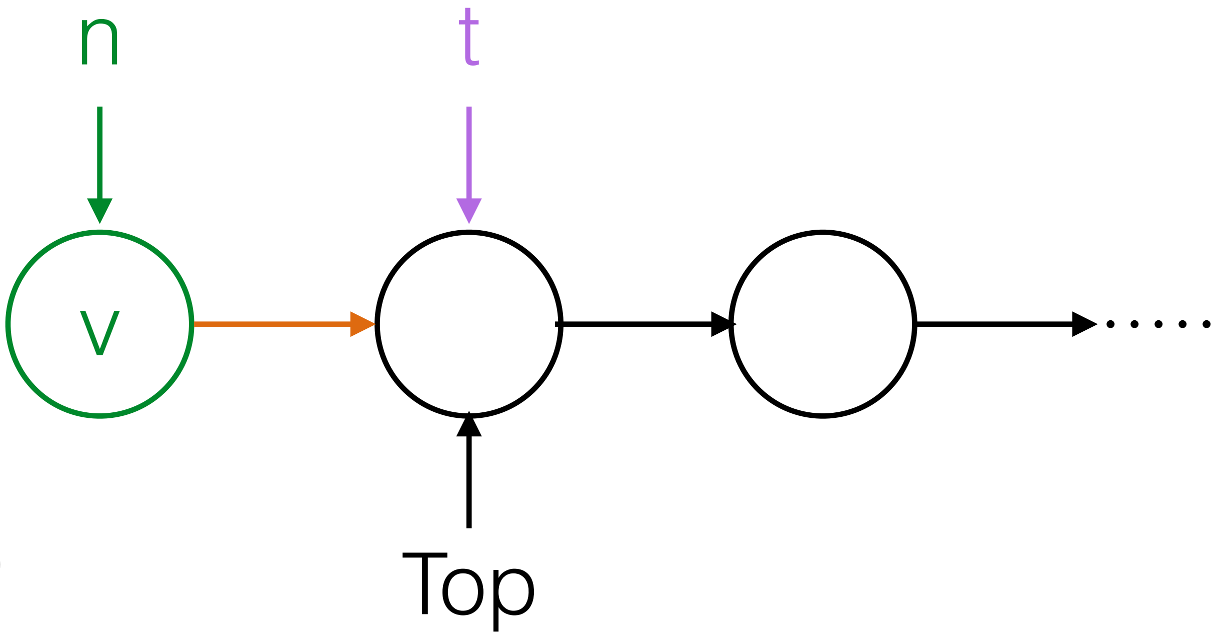
Treiber's Stack

```
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}
```



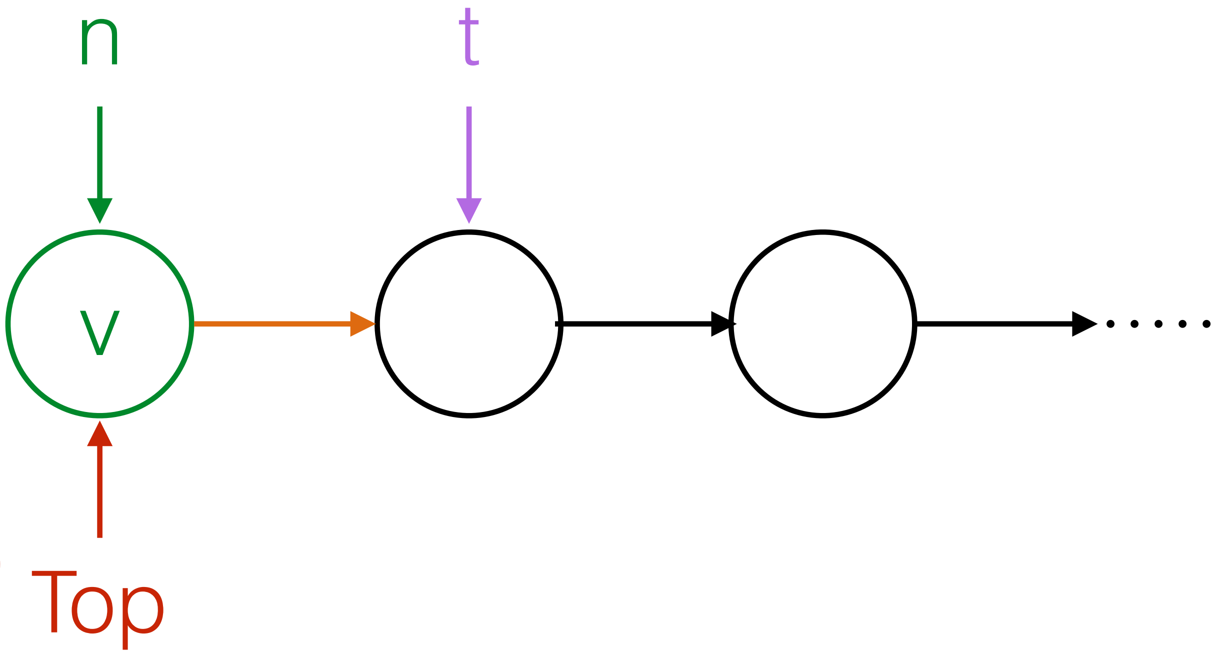
Treiber's Stack

```
void Push (int v) {
        node *n, *t
        node n = new node(v)
        do {
                node *t = Top
                n.next = t
        } while (not CAS(&Top, t, n))
}
```



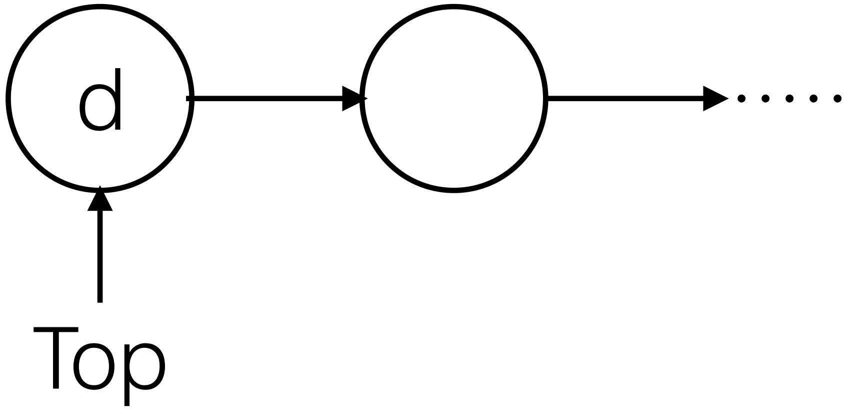
Treiber’s Stack

```
void Push (int v) {
        node *n, *t
        node n = new node(v)
        do {
                node *t = Top
                n.next = t
        } while (not CAS(&Top, t, n))
}
```



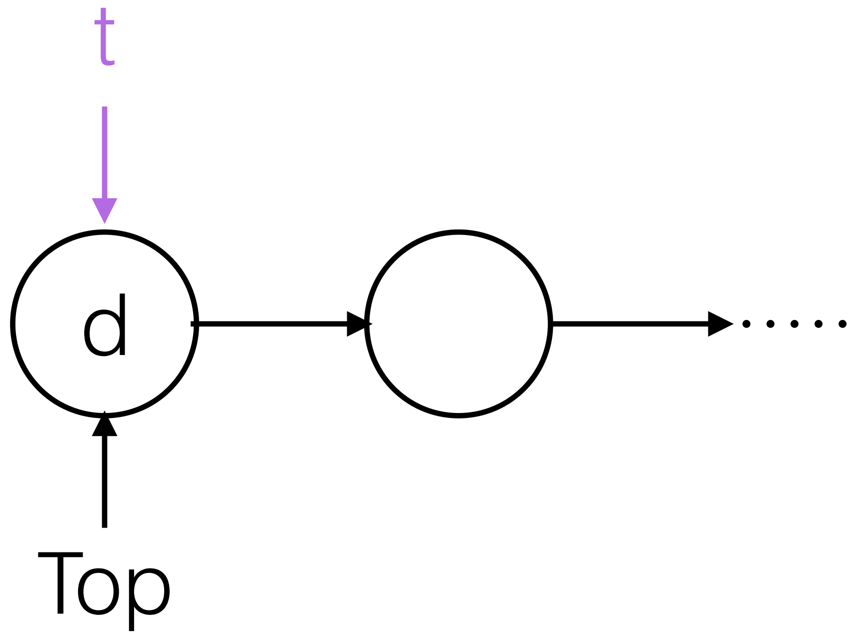
Treiber’s Stack

```
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```



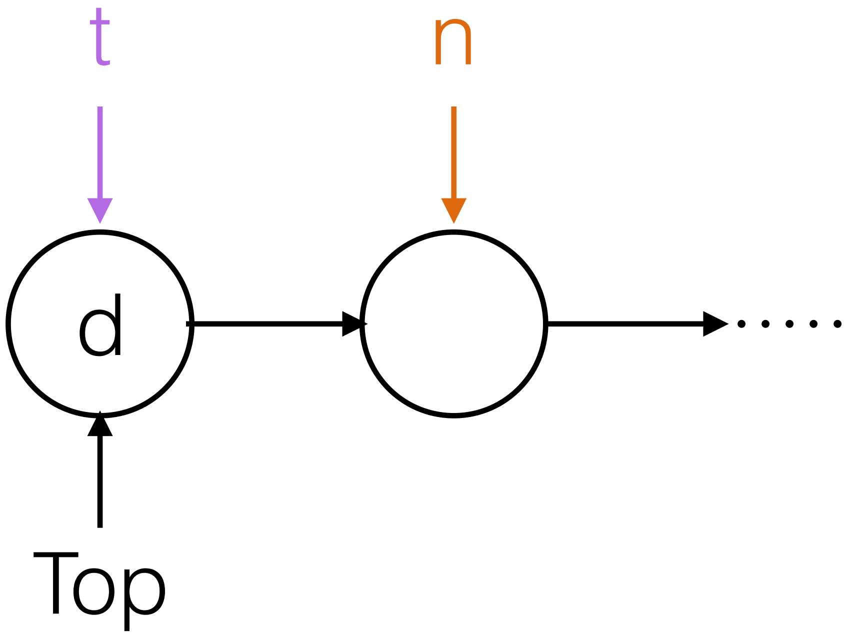
Treiber's Stack

```
int Pop () {
        node *n, *t
        do {
                node *t = Top
                if (t==NULL) return Ø
                n = t.next
        } while (not CAS(&Top, t, n))
        int result = t.data
        free (t)
        return result
}
```



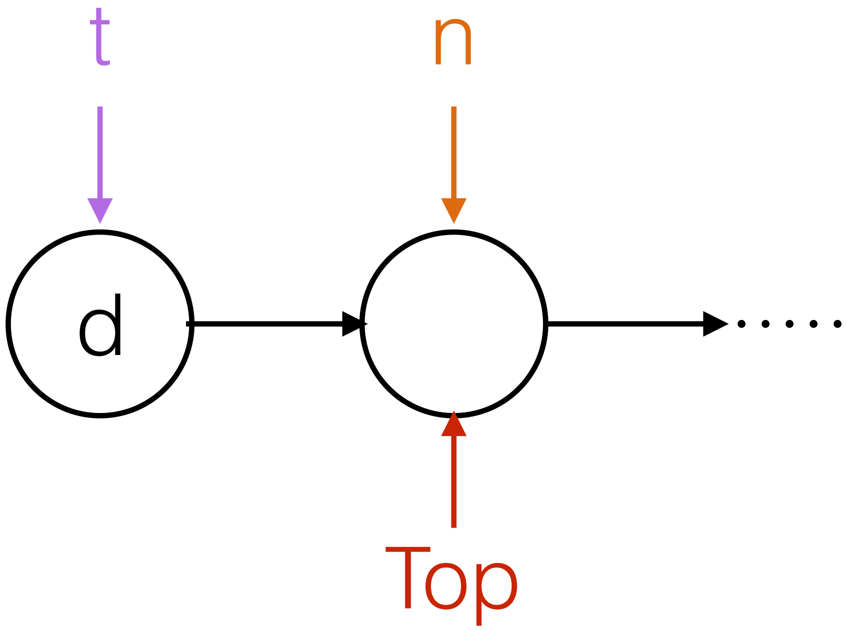
Treiber's Stack

```
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```



Treiber's Stack

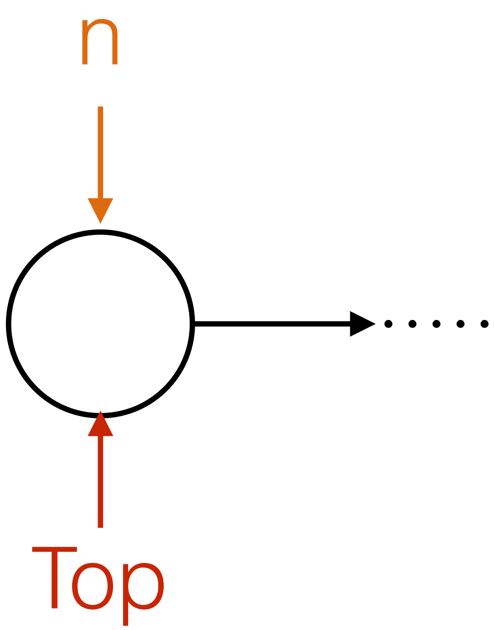
```
int Pop () {
        node *n, *t
        do {
                node *t = Top
                if (t==NULL) return Ø
                n = t.next
        } while (not CAS(&Top, t, n))
        int result = t.data
        free (t)
        return result
}
```



result = d

Treiber's Stack

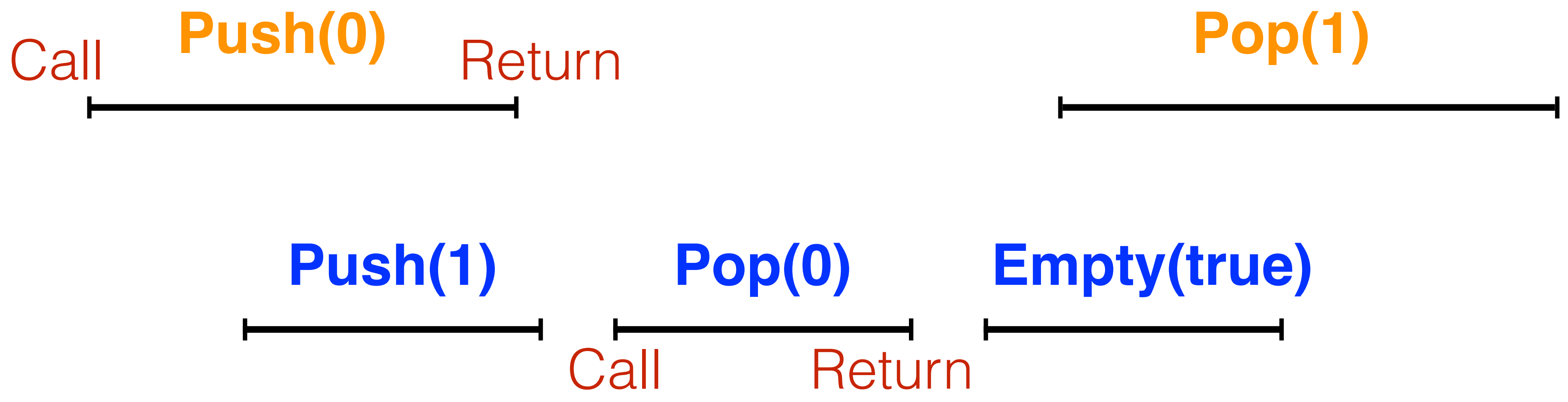
```
int Pop () {
        node *n, *t
        do {
                node *t = Top
                if (t==NULL) return Ø
                n = t.next
        } while (not CAS(&Top, t, n))
        int result = t.data
        free (t)
        return result
}
```



result = d

Efficient Concurrent Implementations

Allow **parallelism** between operations

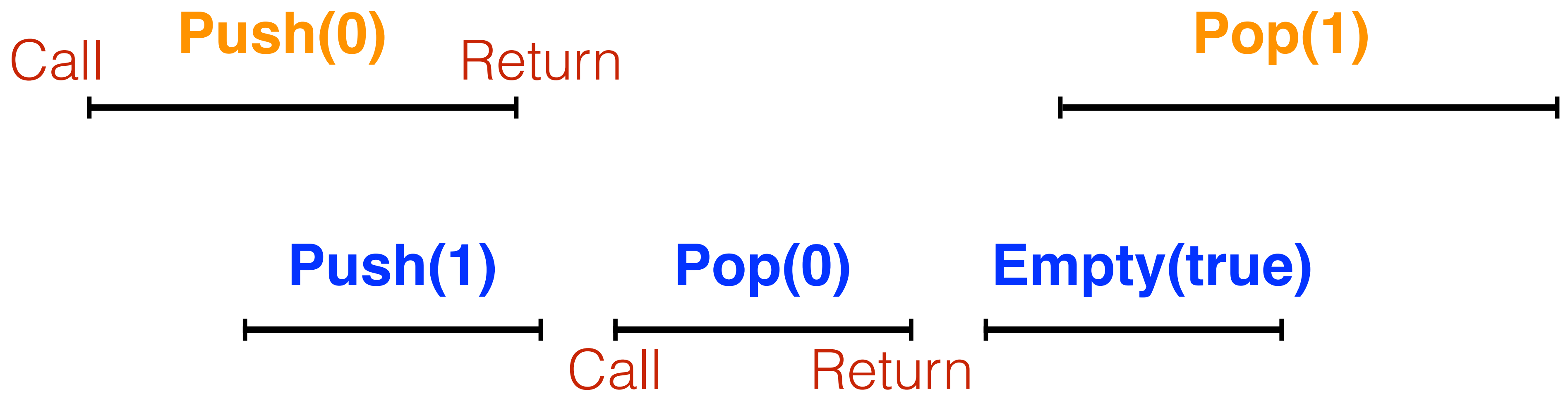


Fine-grain locking (Lock-free algorithms)

- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)

Efficient Concurrent Implementations

Allow **parallelism** between operations



Call **Push(0)** Return

Pop(1)

Push(1)

Pop(0)

Call Return

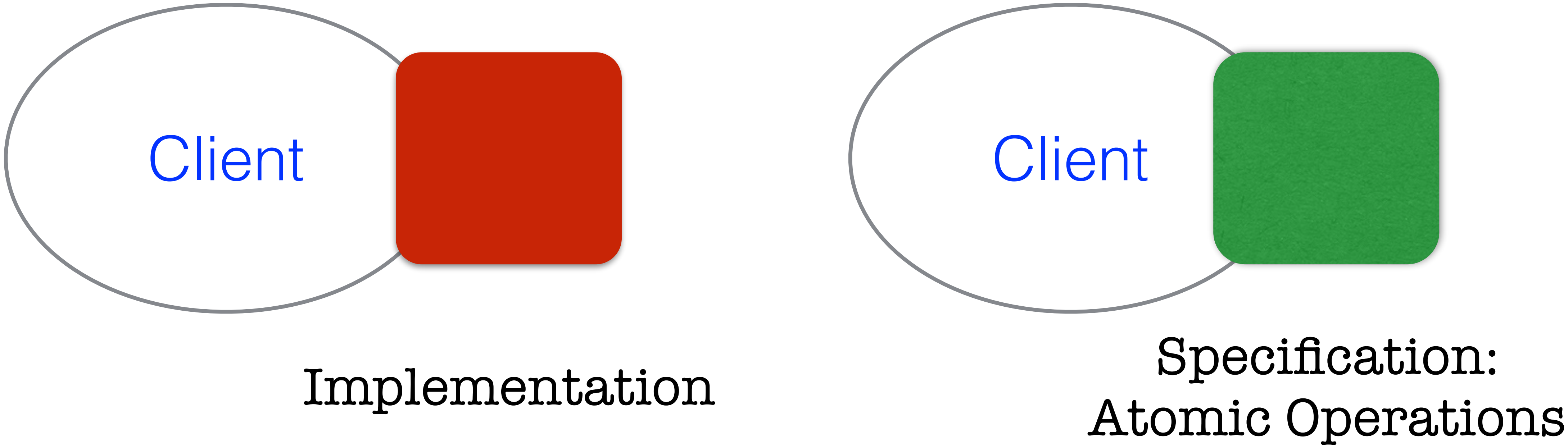
Empty(true)

Fine-grain locking (Lock-free algorithms)

- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)

Consistency ??

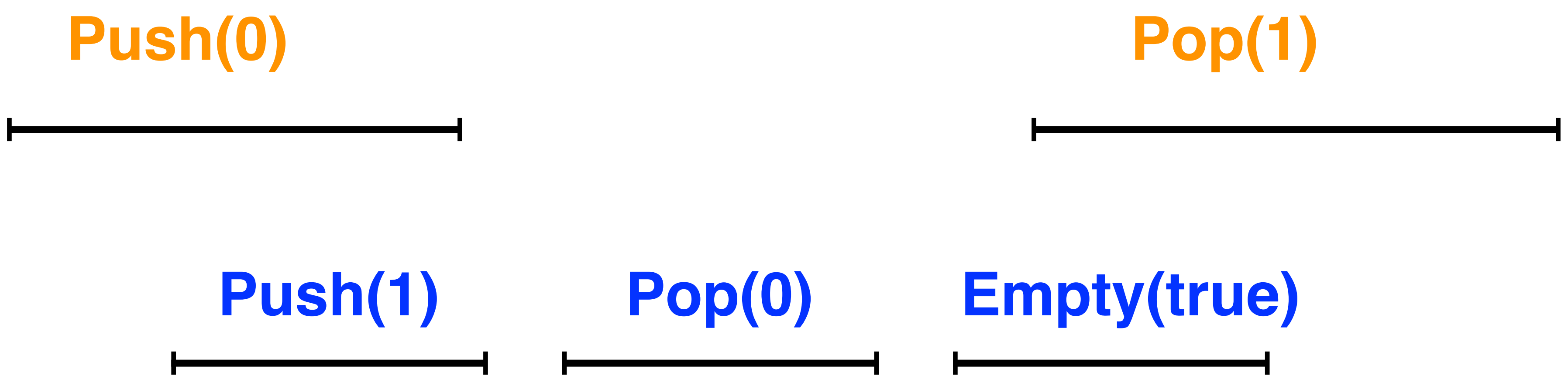
Observational Refinement



For every Client,

Exec (**Client [Impl]**) **is included in** Exec (**Client [Spec]**)

Linearizability [Herlihy, Wing, 1990]



Push(0)

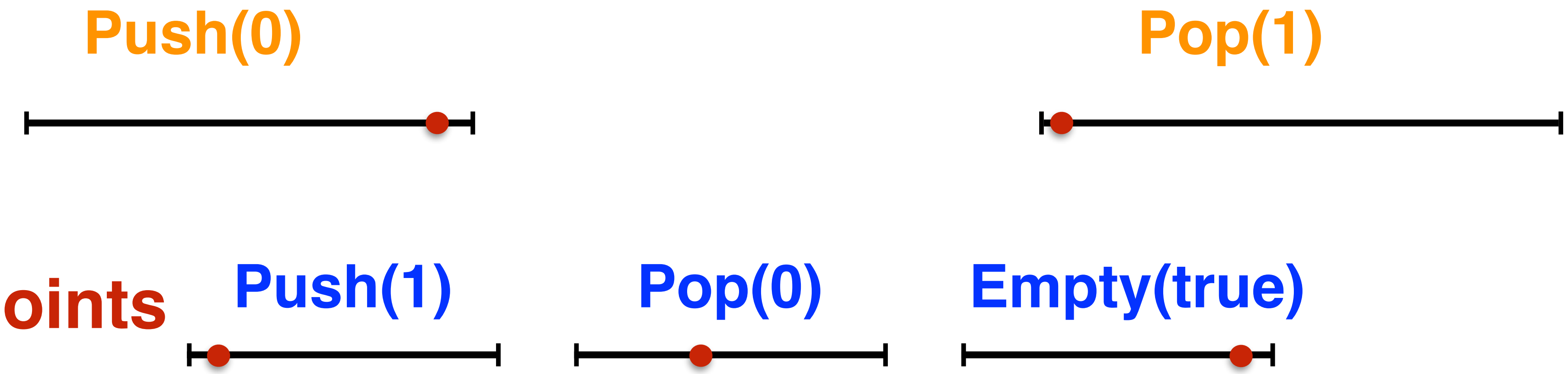
Pop(1)

Push(1)

Pop(0)

Empty(true)

Linearizability [Herlihy, Wing, 1990]



For every execution

Find Linearisation Points

Push(0)

Pop(1)

Push(1)

Pop(0)

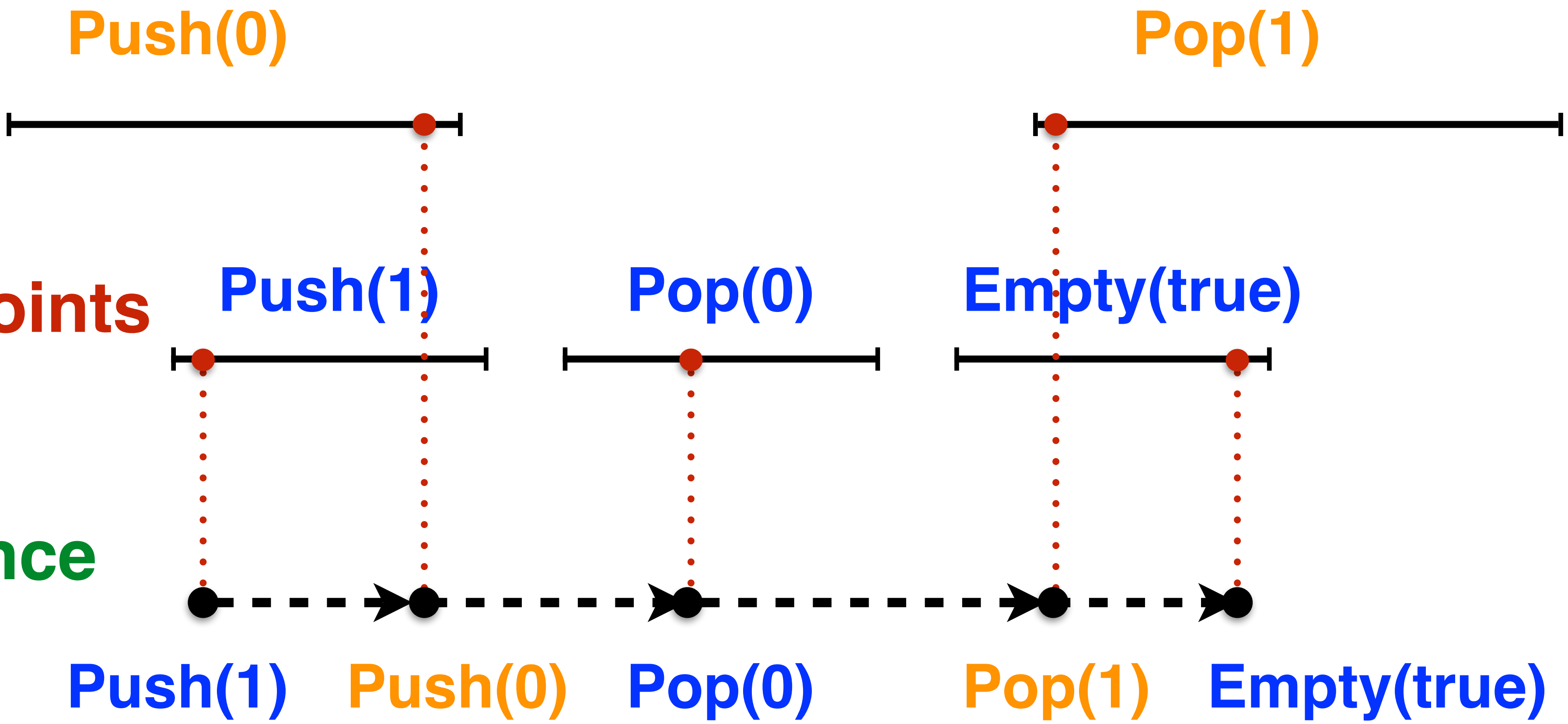
Empty(true)

Linearizability [Herlihy, Wing, 1990]

For every execution

Find Linearisation Points

Match a valid sequence

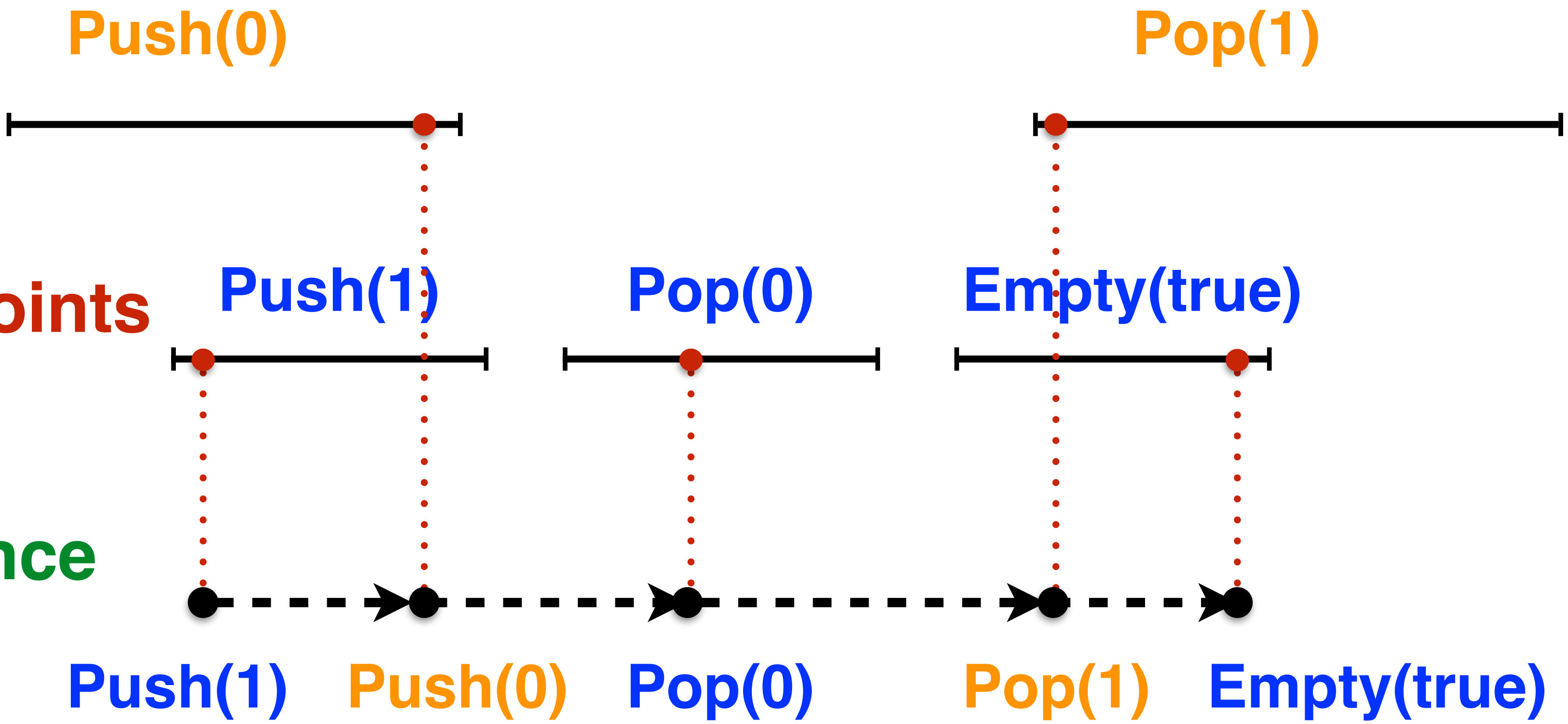


Linearizability [Herlihy, Wing, 1990]

For every execution

Find Linearisation Points

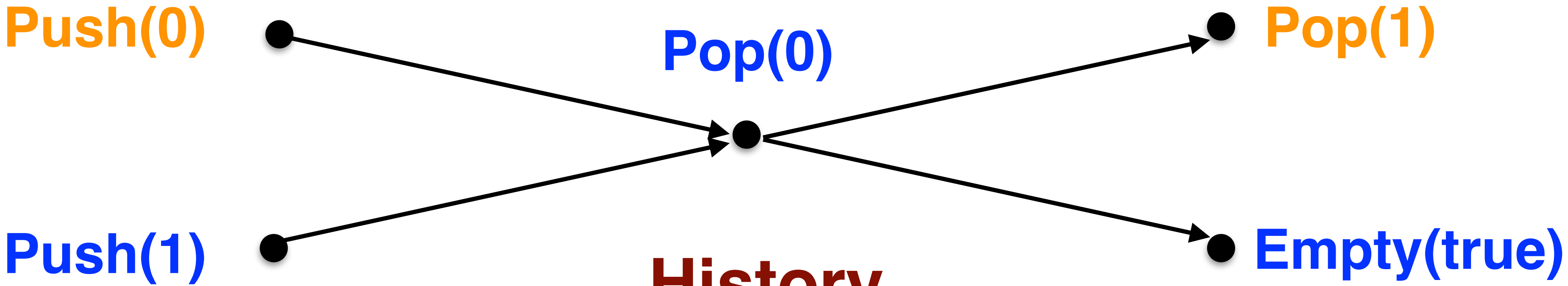
Match a valid sequence



Push(0) **Pop(1)**

Push(1) **Pop(0)** **Empty(true)**

Push(1) **Push(0)** **Pop(0)** **Pop(1)** **Empty(true)**



Push(0) **Pop(0)** **Pop(1)**

Push(1) **Empty(true)**

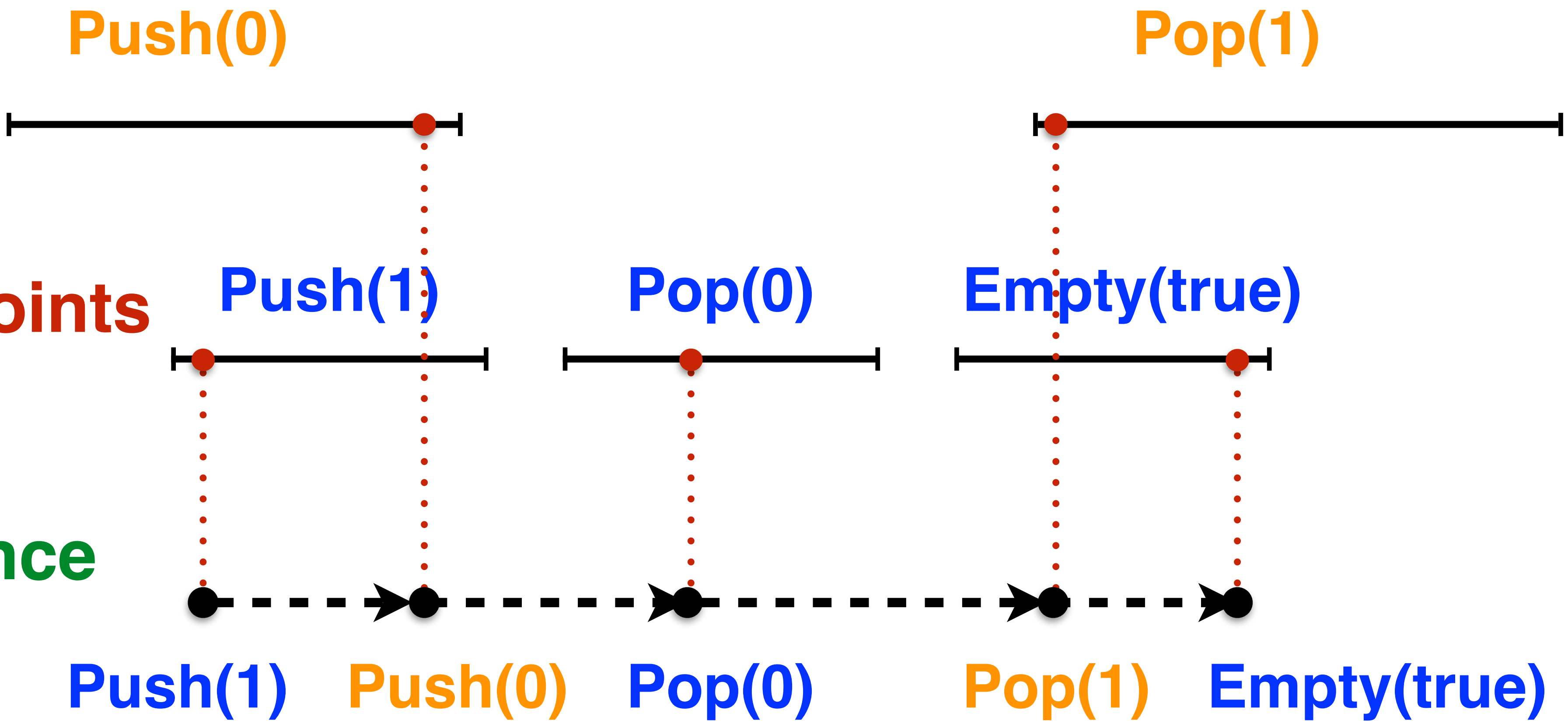
History (Return *Before* Call)

Linearizability [Herlihy, Wing, 1990]

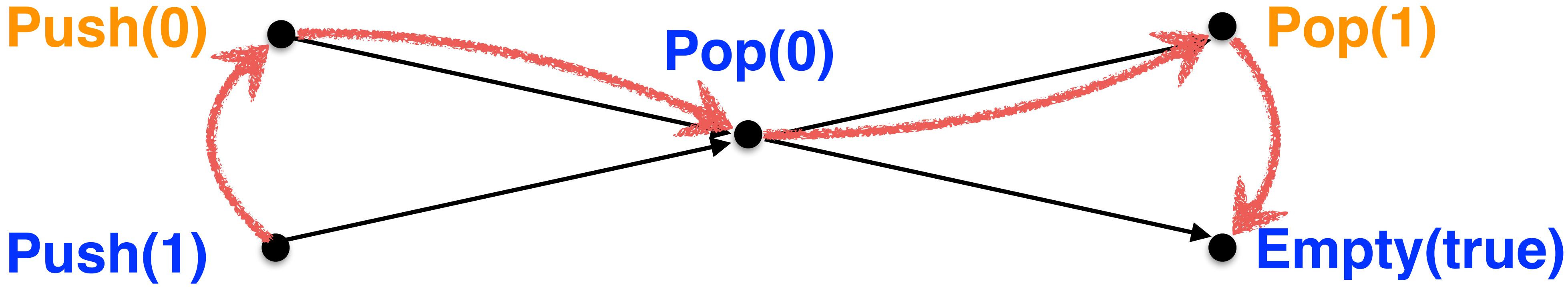
For every execution

Find Linearisation Points

Match a valid sequence



For every history



Find a valid compatible total order

Histories

History of a library execution *e* :

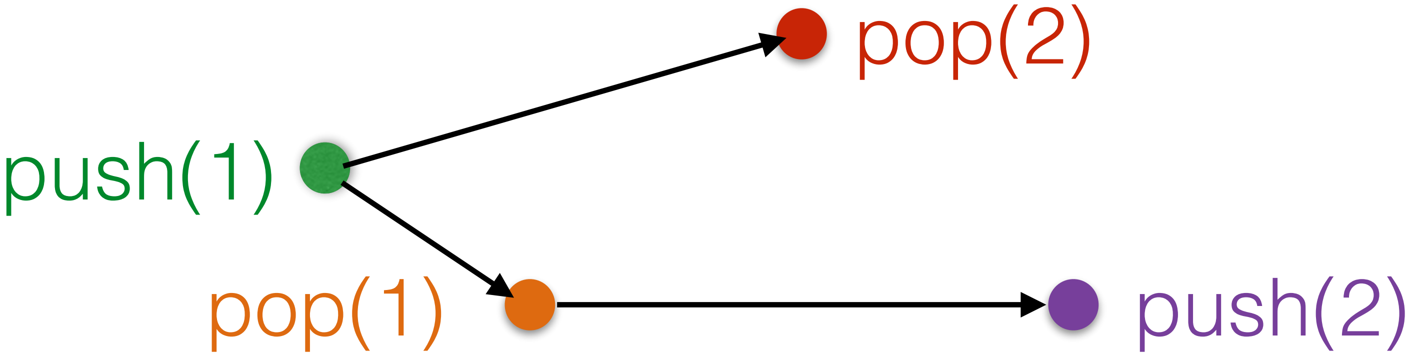
H(*e*) = (O, label, <)

where

- O = Operations(e)
- label: O —> M x V x V
- < is a partial order s.t.

O1 < O2 iff Return(O1) is *before* Call(O2) in *e*

c(push,1) r(push,tt) c(pop,-) c(pop,-) r(pop,1) c(push,2) r(push,tt) r(pop,2)



Linearizability as a History Inclusion

Consider an **abstract data structure**,
let **S** be its **sequential specification**,
and let **L_S** be a **sequential implementation** of S,
i.e., ***L_S satisfies S***

L_C reference concurrent implementation =
L_S + lock/unlock at beginning/end of each method

Linearizability as a History Inclusion

Consider an **abstract data structure**,
let **S** be its **sequential specification**,
and let **L_S** be a **sequential implementation** of S,
i.e., ***L_S satisfies S***

L_C reference concurrent implementation =
L_S + lock/unlock at beginning/end of each method

Lemma:
$H(L_S)$ is the set histories that are linearised to a sequence in S

Thm: **L is linearisable wrt S iff H(L) is included in $H(L_S)$**

History Inclusion vs OR vs Linearizability

History Inclusion vs OR

Thm: **L_1 refines L_2 iff $H(L_1)$ is included in $H(L_2)$**

History Inclusion vs OR vs Linearizability

History Inclusion vs OR

Thm: **L_1 refines L_2 iff $H(L_1)$ is included in $H(L_2)$**

- (=>) Given *h* in $H(L_1)$, construct a client P_h that imposes all the happen-before constraints of *h*.
- (<=) Clients cannot distinguish executions with the same history.

History Inclusion vs OR vs Linearizability

History Inclusion vs OR

Thm: **L_1 refines L_2 iff $H(L_1)$ is included in $H(L_2)$**

- (=>) Given *h* in $H(L_1)$, construct a client P_h that imposes all the happen-before constraints of *h*.
- (<=) Clients cannot distinguish executions with the same history.
  (clients and libraries do not share variables)

OR vs Linearizability

Coro: **L is linearisable w.r.t. S iff L refines L_S**

since: L is linearisable w.r.t. S iff H(L) is included in $H(L_S)$

Verifying Linearizability?

Verifying Linearizability?

Reduction to State Reachability Checking?

- **Reuse existing tools** for Invariance/Reachability checking
- **Complexity, Decidability**

Verifying Linearizability?

Reduction to State Reachability Checking?

- **Reuse existing tools** for Invariance/Reachability checking
- **Complexity, Decidability**

General Approach:

Given a **library L** and a **specification S**,
define a **monitor M** (**+** designated **bad states**) s.t.
L is linearisable wrt S iff
L x M does not reach a bad state

Verifying Linearizability?

Reduction to State Reachability Checking?

- **Reuse existing tools** for Invariance/Reachability checking
- **Complexity, Decidability**

General Approach:

Given a **library L** and a **specification S**,
define a **monitor M** (**+** designated **bad states**) s.t.
L is linearisable wrt S iff
L x M does not reach a bad state

Issue:

- The **computational power** of **M**?
- **Size** of **M**?
- Ideally, finite-state, polynomial size, but …

A Monitor for Linearizability

Given a specification: a state machine

Memory of the monitor

- Set of all possible *linearizations*
- A linearization is represented as a pair:
    - state of the specification
    - set of pending (not yet linearised) methods

Actions of the monitor

- Observe calls and returns: call —> pending —> return
- Guess linearisation points for pending methods in each linearisation (store expected return values by the spec.)
- Checks that returns indeed match the specification
- Fail if all linearizations violate the specification

Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Fixed number of threads

- => EXPSPACE algorithm (see also [Alur, McMillan, Peled 96])

Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Fixed number of threads

- => EXPSPACE algorithm (see also [Alur, McMillan, Peled 96])
- EXPSPACE-hard problem [Hamza 2015]
- Contrasts with State Reachability: PSPACE-complete

Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Unbounded number of threads

- => Unbounded memory

Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Unbounded number of threads

- => Unbounded memory
- Undecidable problem [B., Emmi, Enea, Hamza 2013]
- Contrasts with State Reachability: EXPSPACE-complete

Checking Linearizability: Undecidability

- Reduction of the reachability problem in 2-counter machines
- Given a Machine **M**, build a library **L$_M$** and a specification **S$_M$**

There is a computation of **M** reaching a state **q$_f$**

iff

L$_M$ is **not linearisable** w.r.t. **S$_M$**

Checking Linearizability: Undecidability

- Reduction of the reachability problem in 2-counter machines
- Given a Machine **M**, build a library **L$_M$** and a specification **S$_M$**

There is a computation of **M** reaching a state **q$_f$**

iff

L$_M$ is **not linearisable** w.r.t. **S$_M$**

- **M** has methods **Inc(i), Dec(i), Zero(i), and m(q)**
- Encoding of a **counter**: a **multi-set of parallel Inc's and Dec's**
- **S$_M$** corresponds to "non acceptable" computations
  - **Zero(i)** occurs when **#Inc(i) ≠ #Dec(i)**
  - State **q$_f$** is not reached (**do not contain m(q$_f$)**)

Checking Linearizability: Undecidability

- Reduction of the reachability problem in 2-counter machines
- Given a Machine **M**, build a library **L_M** and a specification **S_M**

There is a computation of **M** reaching a state **q_f**

iff

L_M is **not linearisable** w.r.t. **S_M**

- **M** has methods **Inc(i), Dec(i), Zero(i), and m(q)**
- Encoding of a **counter**: a **multi-set of parallel Inc's and Dec's**
- **S_M** corresponds to "non acceptable" computations
  - **Zero(i)** occurs when **#Inc(i) ≠ #Dec(i)**
  - State **q_f** is not reached (**do not contain m(q_f)**)

- **S_M** can be a **regular language** (particular order Inc's and Dec's)
- Checking linearizability => consider all orders

Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. *Line-up* [Burckhardt et al. 2010]

Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. *Line-up* [Burckhardt et al. 2010]

Fixed linearization points in the code (verif. correctness)

e.g., [Vafeiadis, CAV'10], [B., Emmi, Enea, Hamza 2013],
[Abdulla et al., TACAS 2013]

Fixed Linearisation Points: Treiber Stack

```
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}
```

```
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```

Checking Linearizability: Fixed Linearisation Points

[B., Emmi, Enea, Hamza 2013]

- No need to guess linearisation points
- => Monitor keeps track of only ***one*** linearisation
- Linearisation = state of the spec. + set of pending op.
- Linearisation point => Move the state of the spec. + record the expected return value
- Return => check the value is conform to the spec.

Checking Linearizability: Fixed Linearisation Points

[B., Emmi, Enea, Hamza 2013]

- No need to guess linearisation points
- => Monitor keeps track of only ***one*** linearisation
- Linearisation = state of the spec. + set of pending op.
- Linearisation point => Move the state of the spec. + record the expected return value
- Return => check the value is conform to the spec.

- Fixed number of FS threads, FS spec. : PSPACE-complete

- Unbounded number of FS threads, FS spec. :
  - Count the number of pending methods of each type
  - => State reachability in VASS (Petri Nets): EXPSPACE-complete

Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. *Line-up* [Burckhardt et al. 2010]

Fixed linearization points in the code (verif. correctness)

e.g., [Vafeiadis, CAV'10], [B., Emmi, Enea, Hamza 2013],
[Abdulla et al., TACAS 2013]

Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. *Line-up* [Burckhardt et al. 2010]

Scalability issues!

Fixed linearization points in the code (verif. correctness)

e.g., [Vafeiadis, CAV'10], [B., Emmi, Enea, Hamza 2013], [Abdulla et al., TACAS 2013]

Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. *Line-up* [Burckhardt et al. 2010]

Scalability issues!

Fixed linearization points in the code (verif. correctness)

e.g., [Vafeiadis, CAV'10], [B., Emmi, Enea, Hamza 2013],
[Abdulla et al., TACAS 2013]

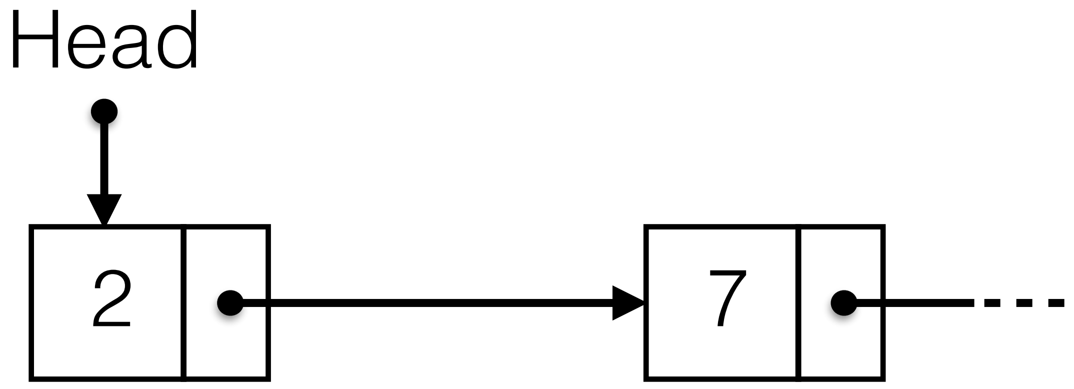
Fixing linearisation points in not always possible!

e.g.,
Helping mechanisms based stacks/queues
Time-stamping based stack [Dodds, Haas, Kirsch, 2015]

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

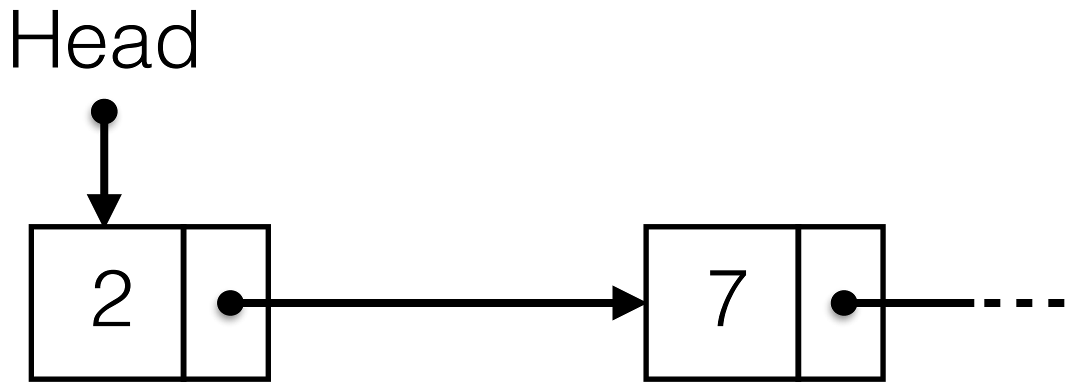
```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

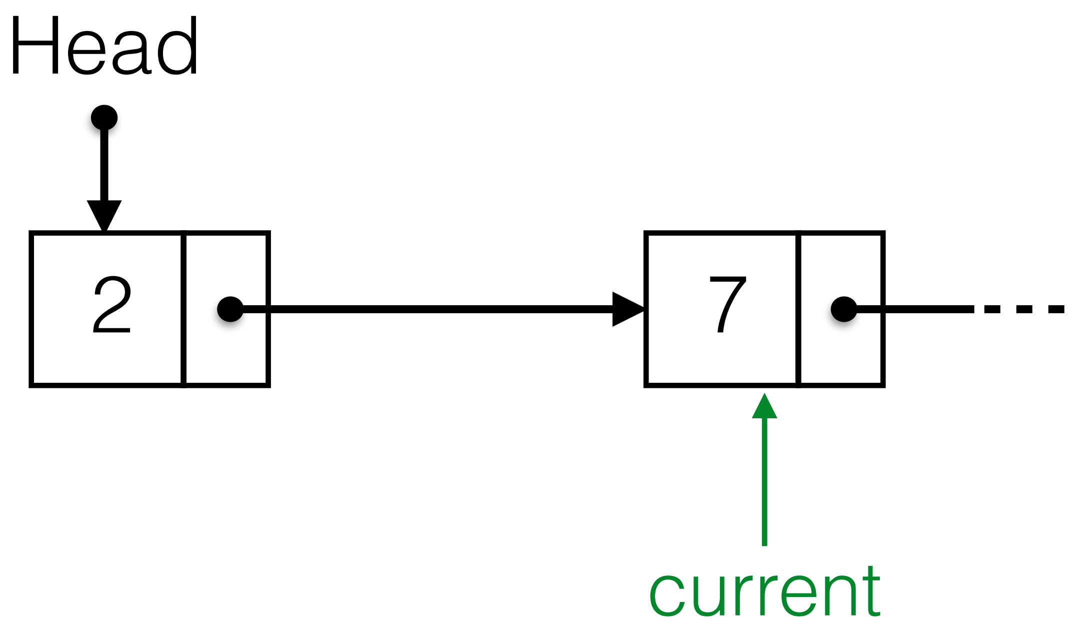
```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



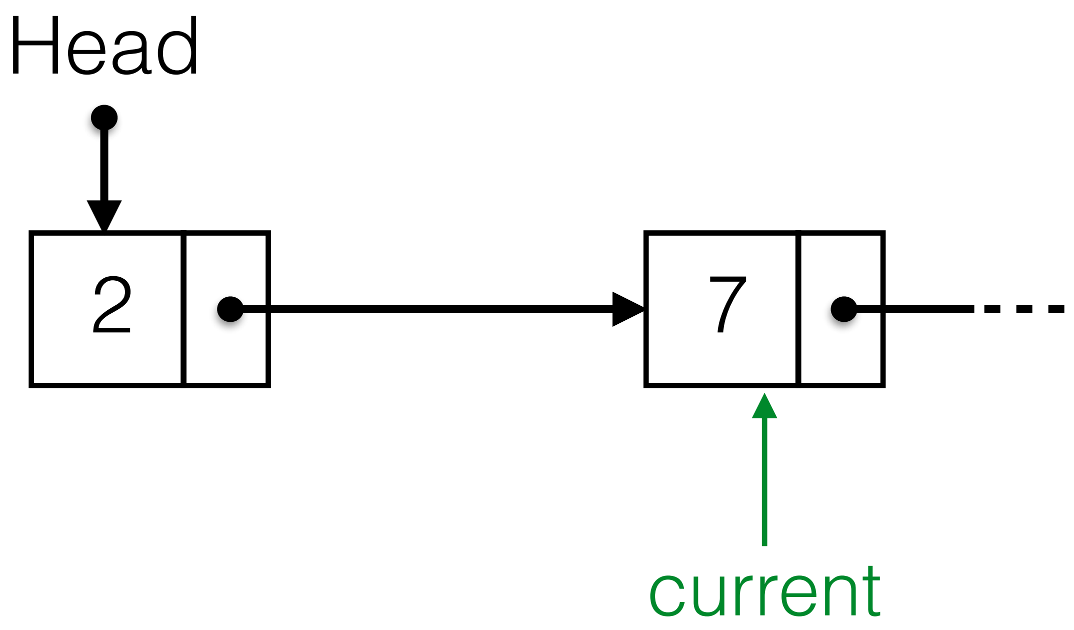
Contains(5)

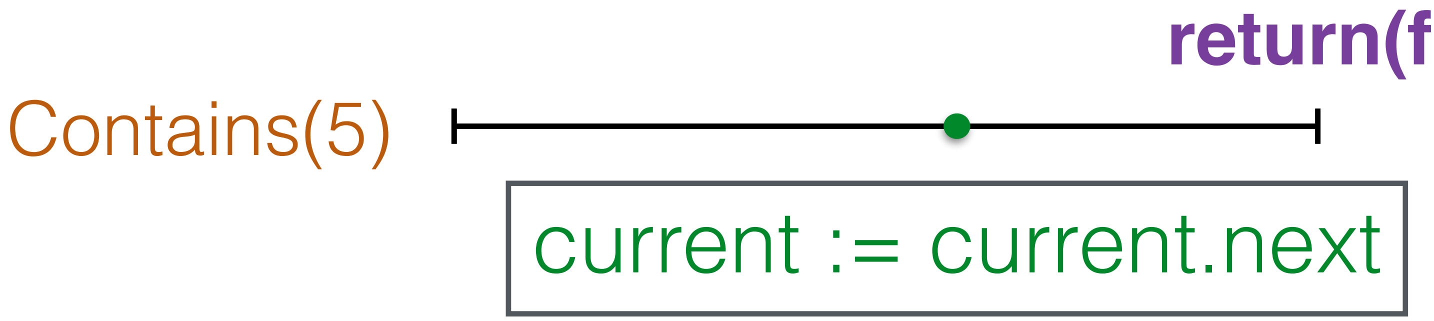
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```





return(false)

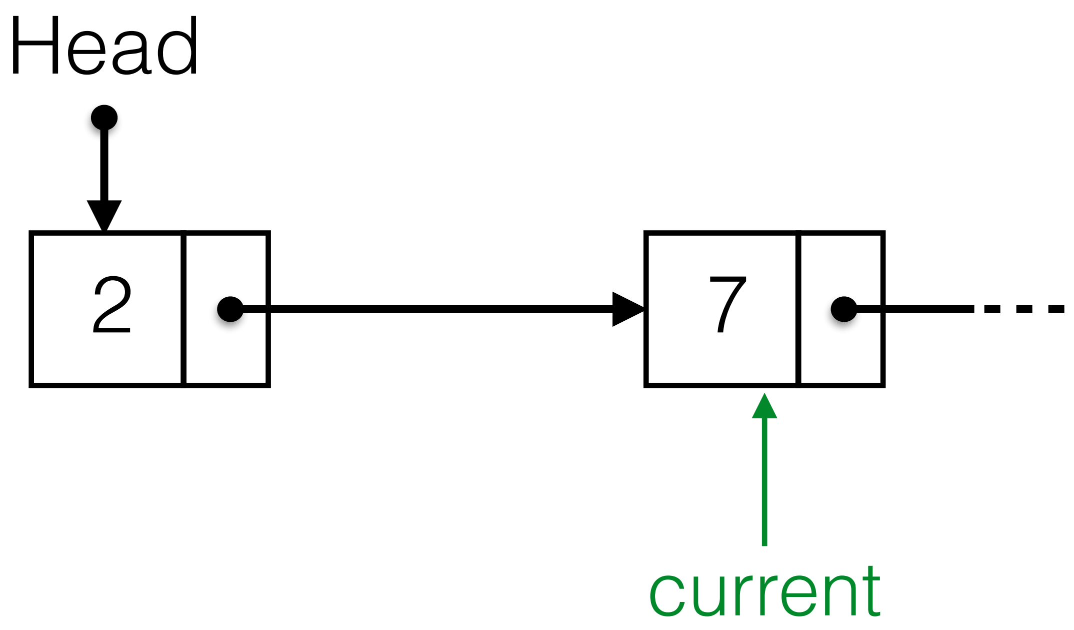
Contains(5)

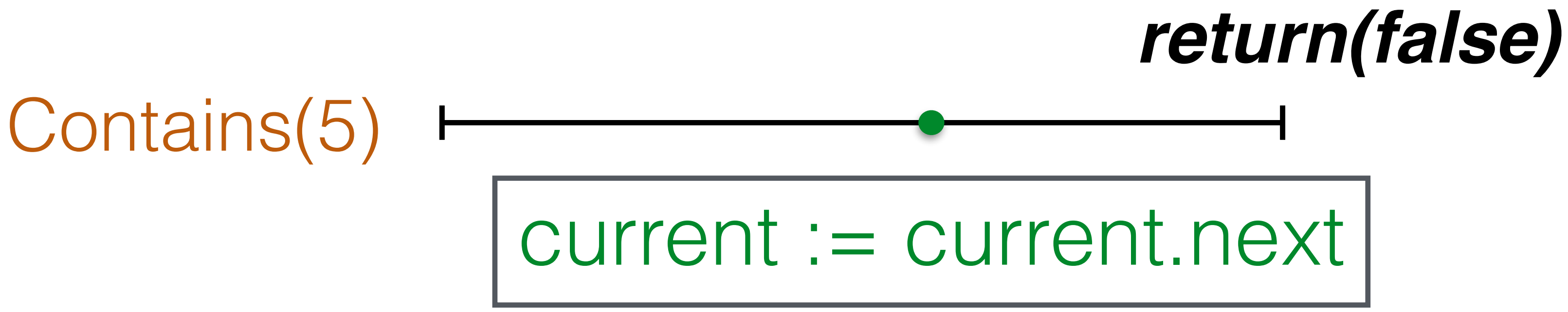
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```





Contains(5)

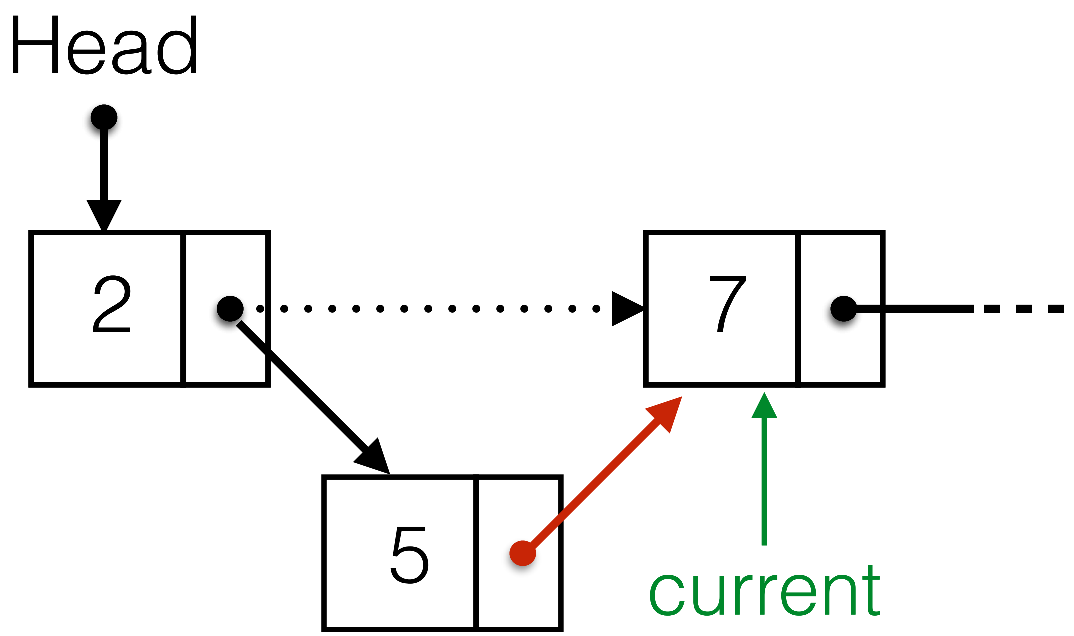
return(false)

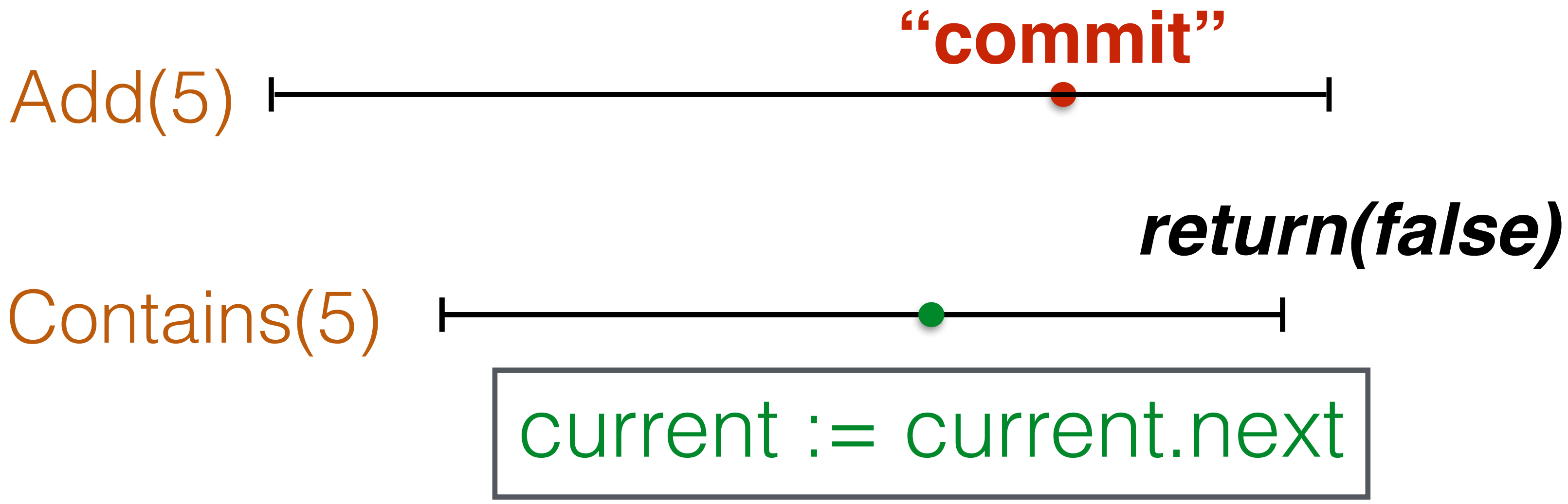
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```




“commit”

Add(5)

return(false)

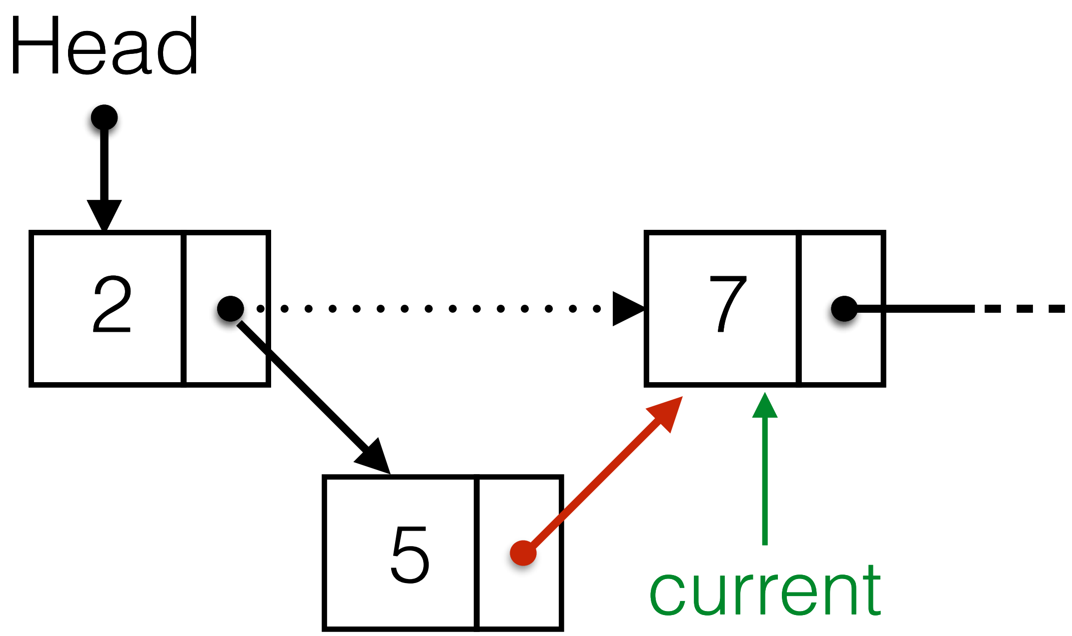
Contains(5)

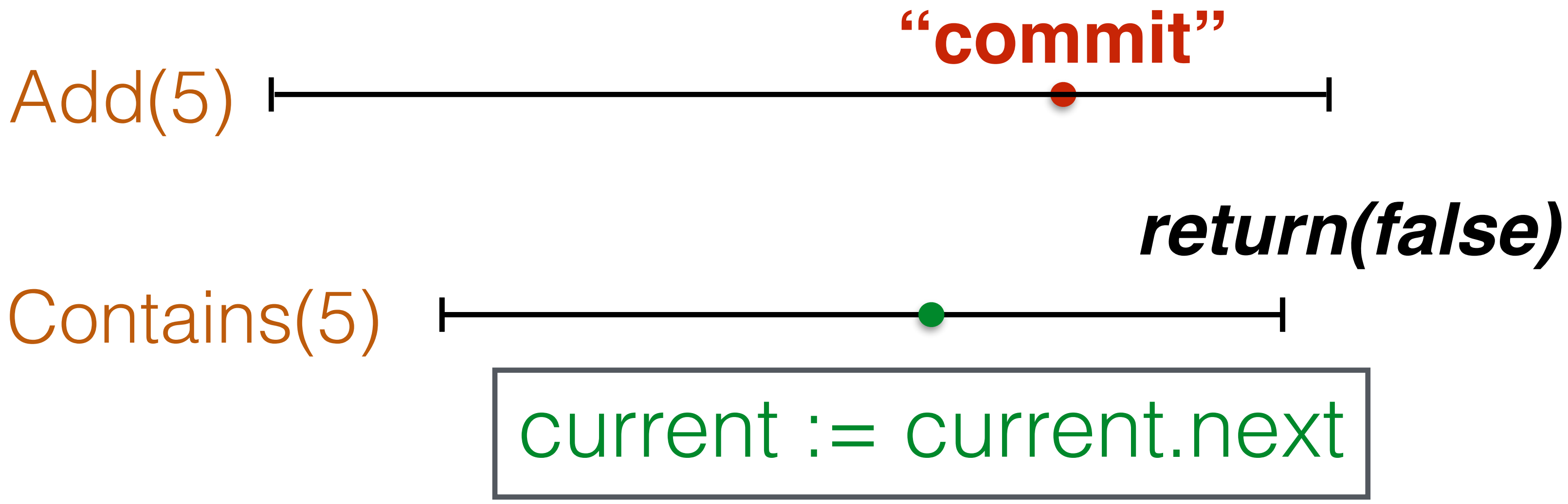
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



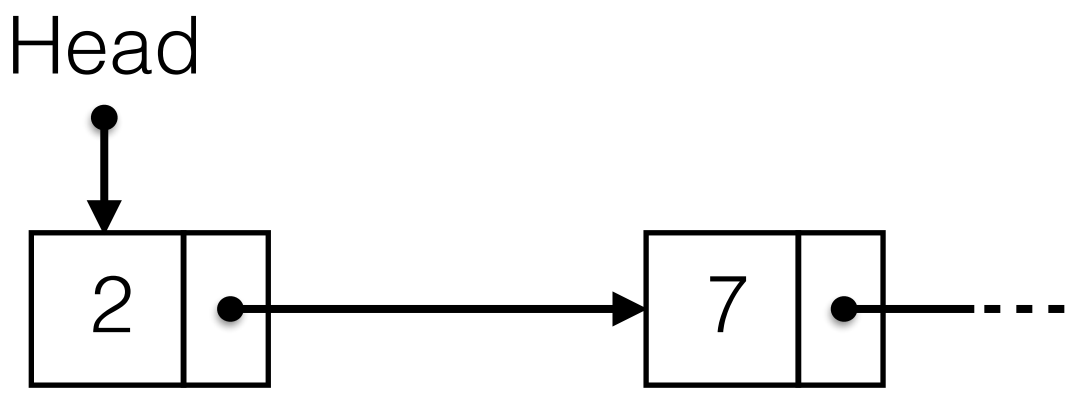


Linearize before Commit of Add(5)

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

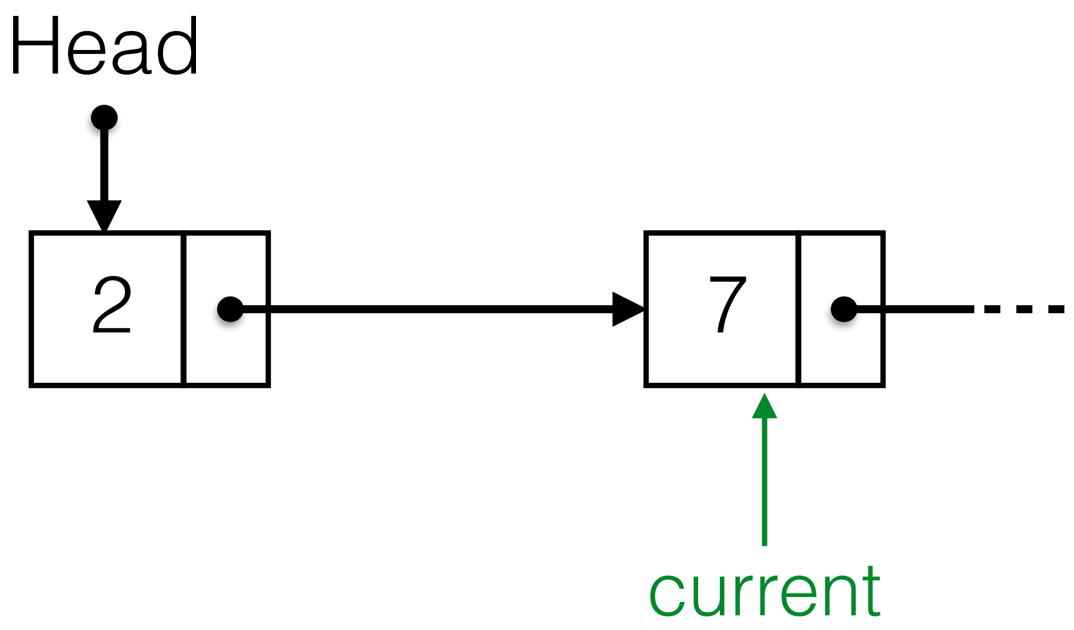
```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



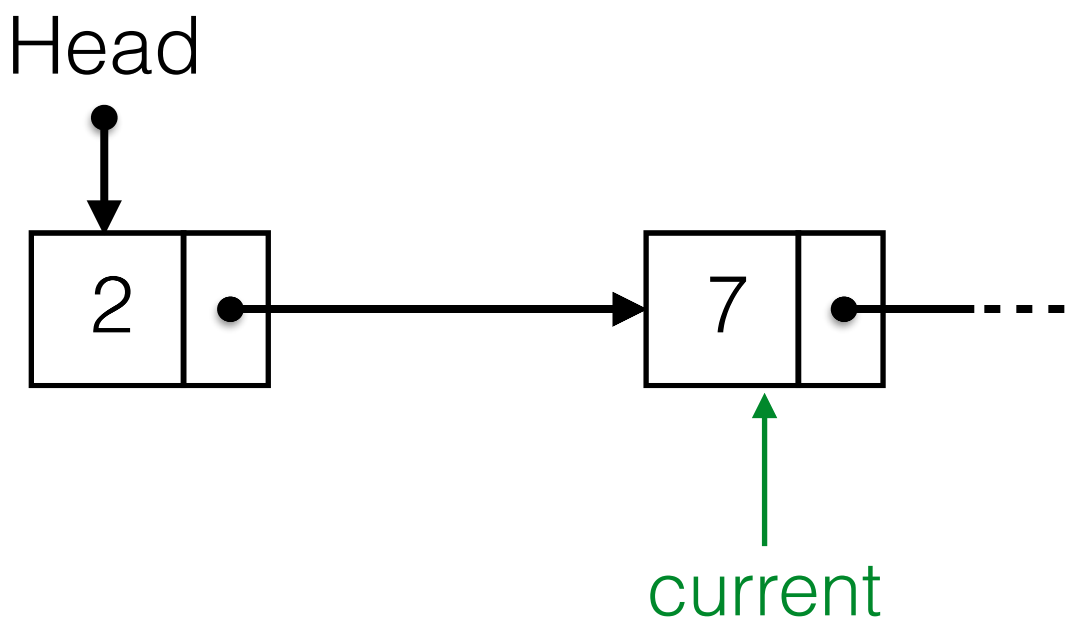
Contains(7)

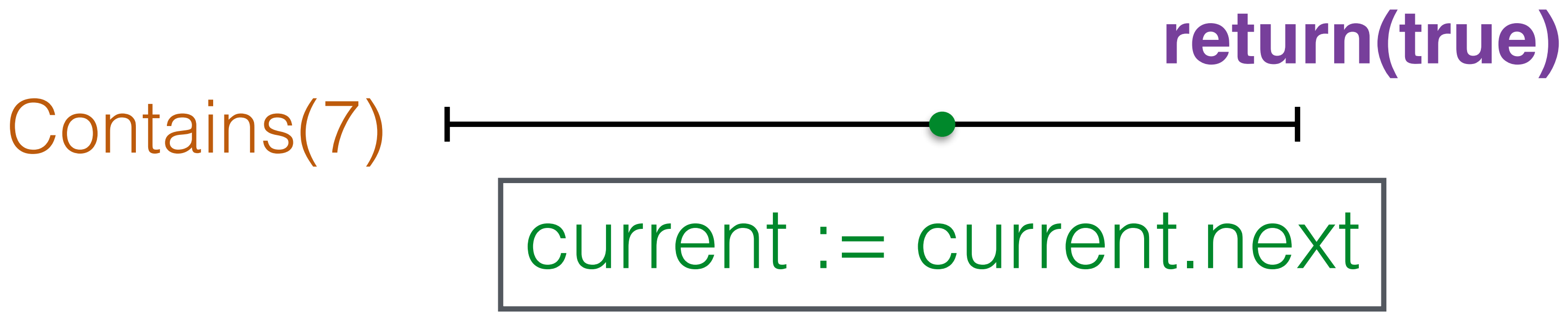
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```

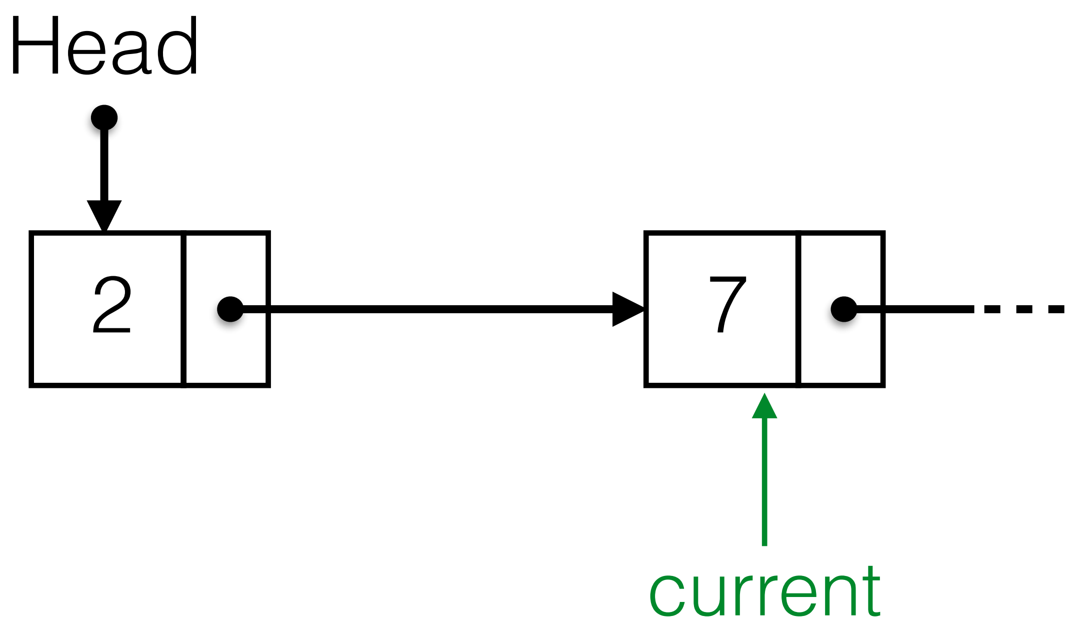


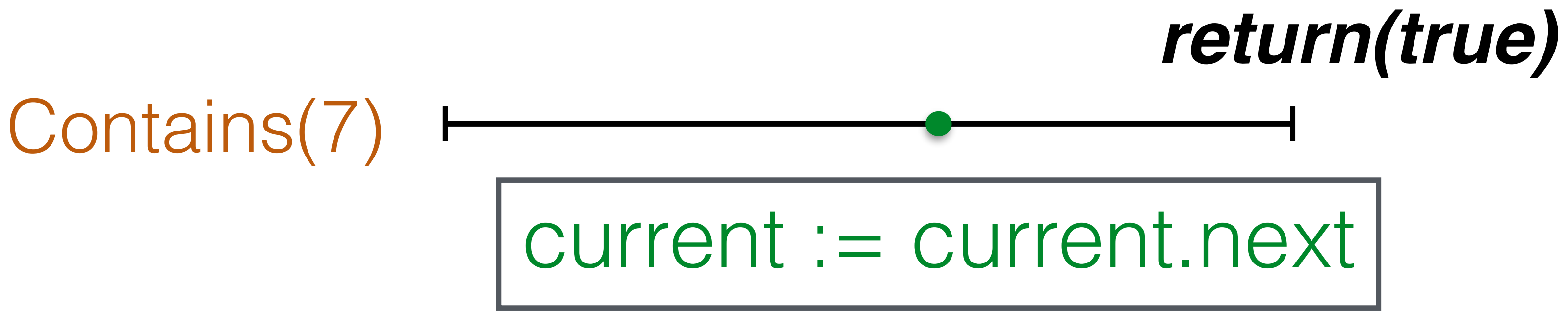


Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```




return(true)

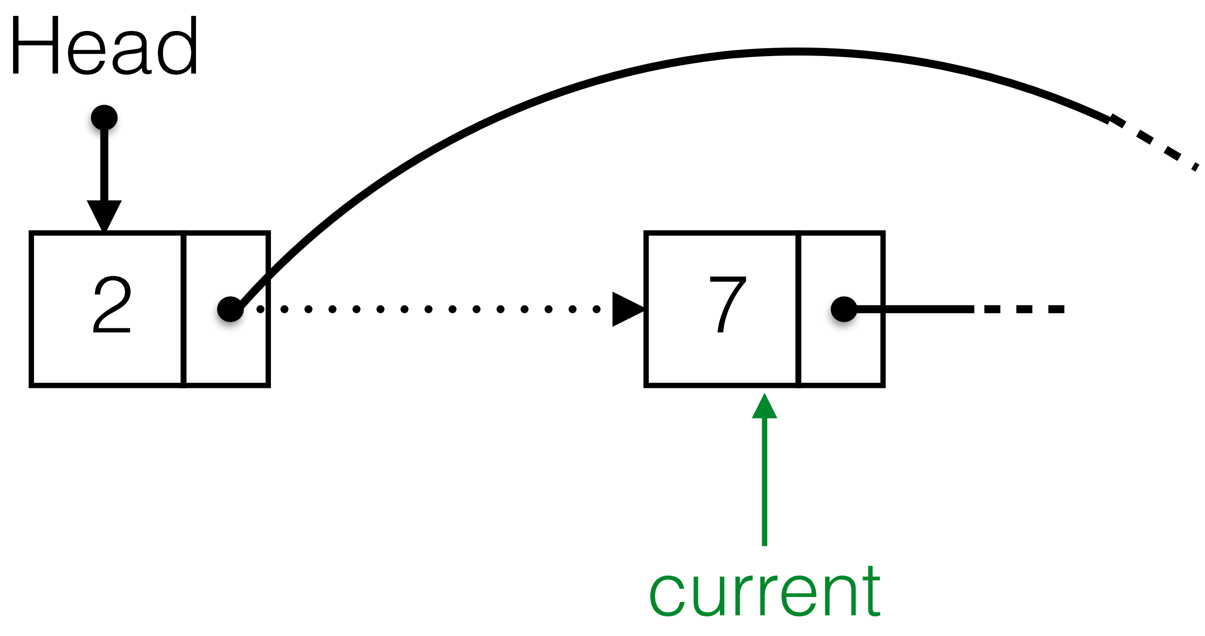
Contains(7)

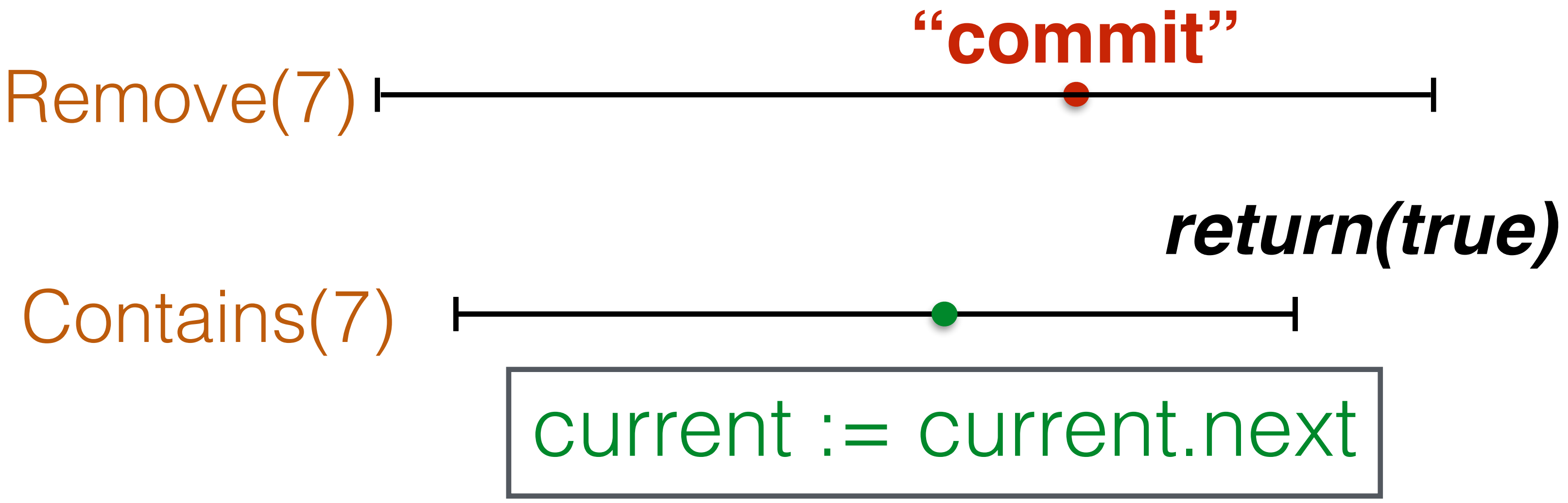
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```



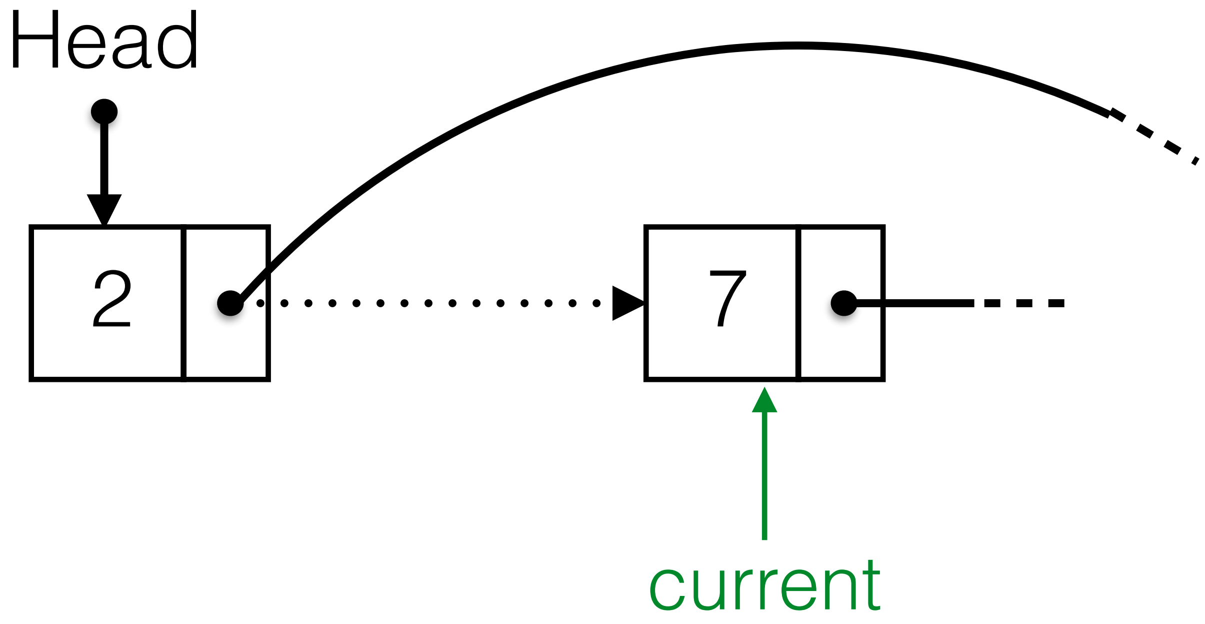


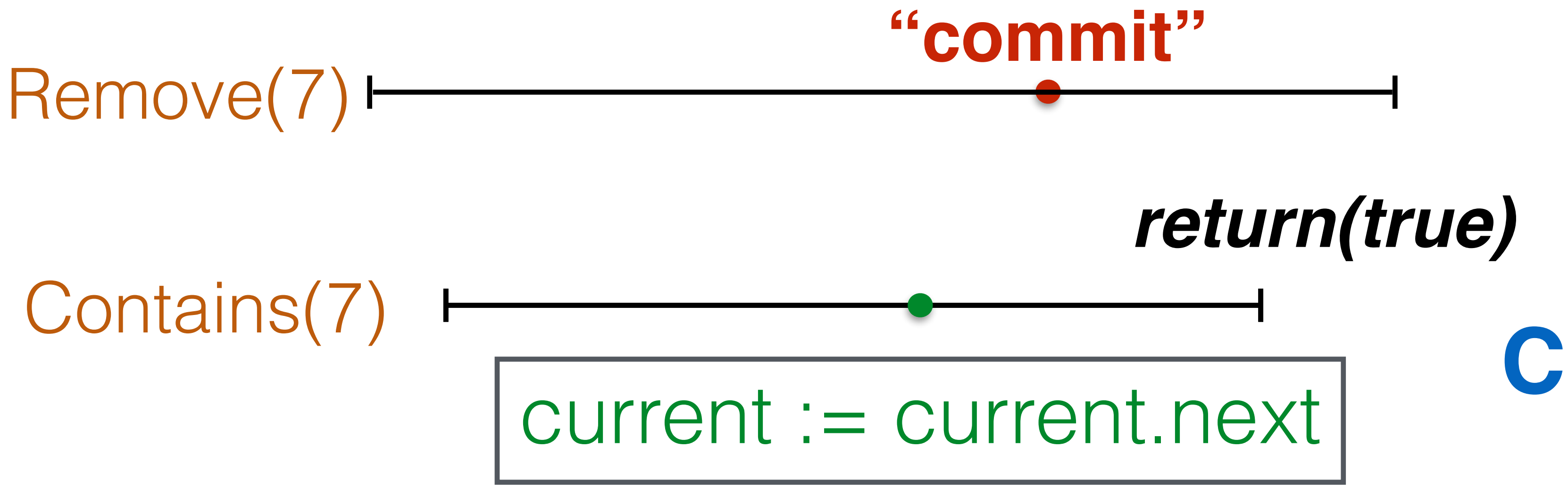
current := current.next

Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```




“commit”

Remove(7)

return(true)

Contains(7)

current := current.next

Linearize before Commit of Remove(7)

Fixed Linearisation Points + Read-Only Methods

- No need to guess linearisation points
- => Monitor keeps track of only ***one*** linearisation
- Linearisation = state of the spec. + set of pending op.
- Linearisation point => Move the state of the spec. + record the expected return value
- + Linearize all read-only methods returning false Before
- + Linearize all read-only methods returning true After
- Return => check the value is conform to the spec.

Summary

- Correctness of a concurrent library = Linearizability

Summary

- Correctness of a concurrent library = Linearizability
- Checking linearizability is a complex problem

	Linearizability	Fixed Lin. Points Linearizability + Read-Only	State Reachability
Fixed Nb Threads	EXPSPACE-C (1)	PSPACE-C (2)	PSPACE-C
Unbounded Nb of Threads	Undecidable (2)	EXPSPACE-C (2)	EXPSPACE-C

(1) Upper Bound: Alur, McMillan, Peled 1996 — Lower Bound: Hamza 2015
(2) B., Emmi, Enea, Hamza, 2013

Summary

- Correctness of a concurrent library = Linearizability
- Checking linearizability is a complex problem

	Linearizability	Fixed Lin. Points Linearizability + Read-Only	State Reachability
Fixed Nb Threads	EXPSPACE-C (1)	PSPACE-C (2)	PSPACE-C
Unbounded Nb of Threads	Undecidable (2)	EXPSPACE-C (2)	EXPSPACE-C

(1) Upper Bound: Alur, McMillan, Peled 1996 — Lower Bound: Hamza 2015
(2) B., Emmi, Enea, Hamza, 2013

- Tractable reductions to state state reachability?
- Avoid reasoning about linearisation points?

Tractable Linearizability Checking?

Tractable Linearizability Checking?

Special classes of implementations

- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

Tractable Linearizability Checking?

Special classes of implementations

- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

Special classes of specifications (abstract structures)

Common structures: stacks, queues, registers, …

Tractable Linearizability Checking?

Special classes of implementations

- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

Special classes of specifications (abstract structures)

Common structures: stacks, queues, registers, …

Special classes of behaviours

- Suitable bounding concepts
- Parametrised under-approximation schemas (bugs detection)
- Good coverage, scalability?

Tractable Linearizability Checking?

Special classes of implementations

- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

Special classes of specifications (abstract structures)

Common structures: stacks, queues, registers, …

Special classes of behaviours

- Suitable bounding concepts
- Parametrised under-approximation schemas (bugs detection)
- Good coverage, scalability?

Focusing on a Class of Concurrent Objects

[B, Emmi, Enea, Hamza, ICALP'15]

- Consider a class of specifications including: stack, queue, register, mutex.
- Characterizing the set of concurrent violations: **A finite number of "bad patterns"** (ordered sets of operations that should not be embedded in any correct execution)
- Defining **finite-state automata** recognising the set executions that include one of the "bad patterns" (using a *data independence* assumption)
- **Linear reduction** of *linearizability checking* to state reachability problem (using these automata as monitors.)
- Decidability for an unbounded number of FS threads.

Specifying queues and stacks

Queue

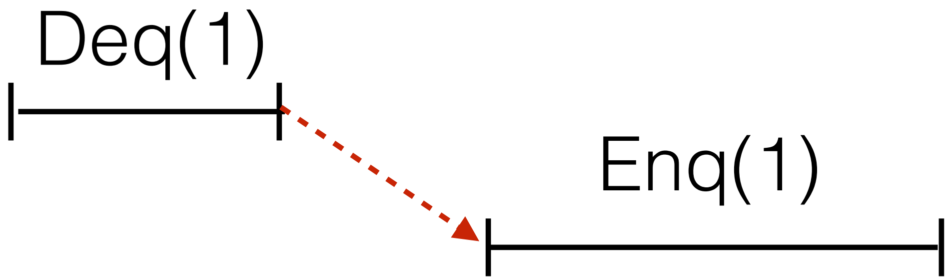
- u . v : Q & u : ENQ* —> **Enq(x)** . u . **Deq(x)** . v : Q
- u . v : Q & no unmatched *Enq* in u —> u . **Emp** . v : Q

Stack

- u . v : S & no unmatched *Push* in u —> **Push(x)** . u . **Pop(x)** . v : S
- u . v : S & no unmatched *Push* in u —> u . **Emp** . v : S

Order Violation

FIFO violations:



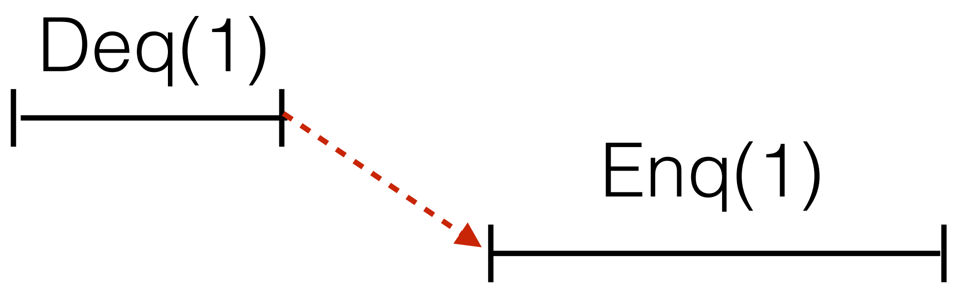
Deq(1)

Enq(1)

ret(Deq(1)) < call(Enq(1))

Order Violation

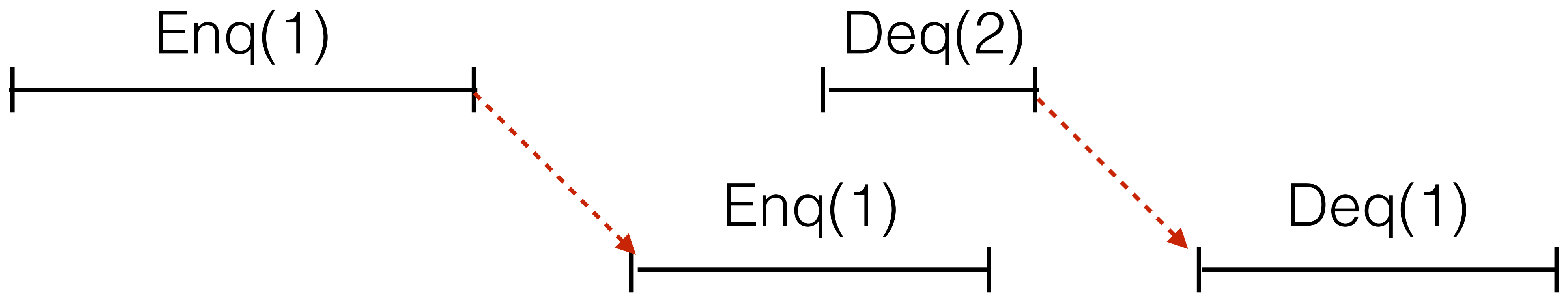
FIFO violations:



Deq(1)

Enq(1)

ret(Deq(1)) < call(Enq(1))



Enq(1)

Deq(2)

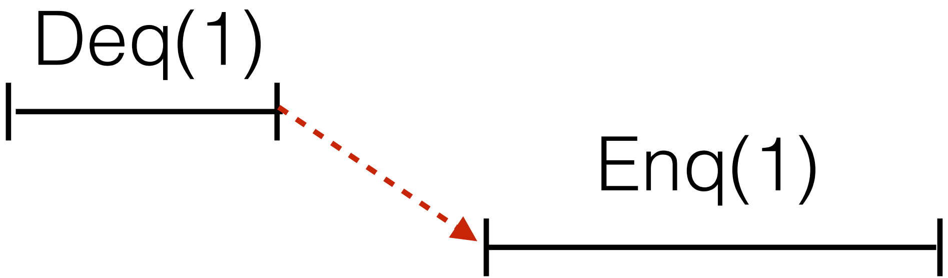
Enq(1)

Deq(1)

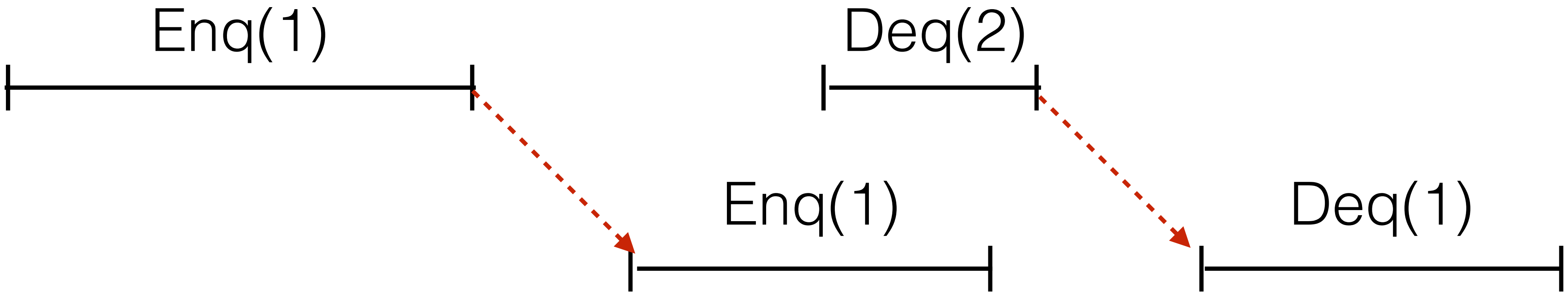
ret(Enq(1)) < call(Enq(2)) & ret(Deq(2)) < call(Deq(1))

Order Violation

FIFO violations:



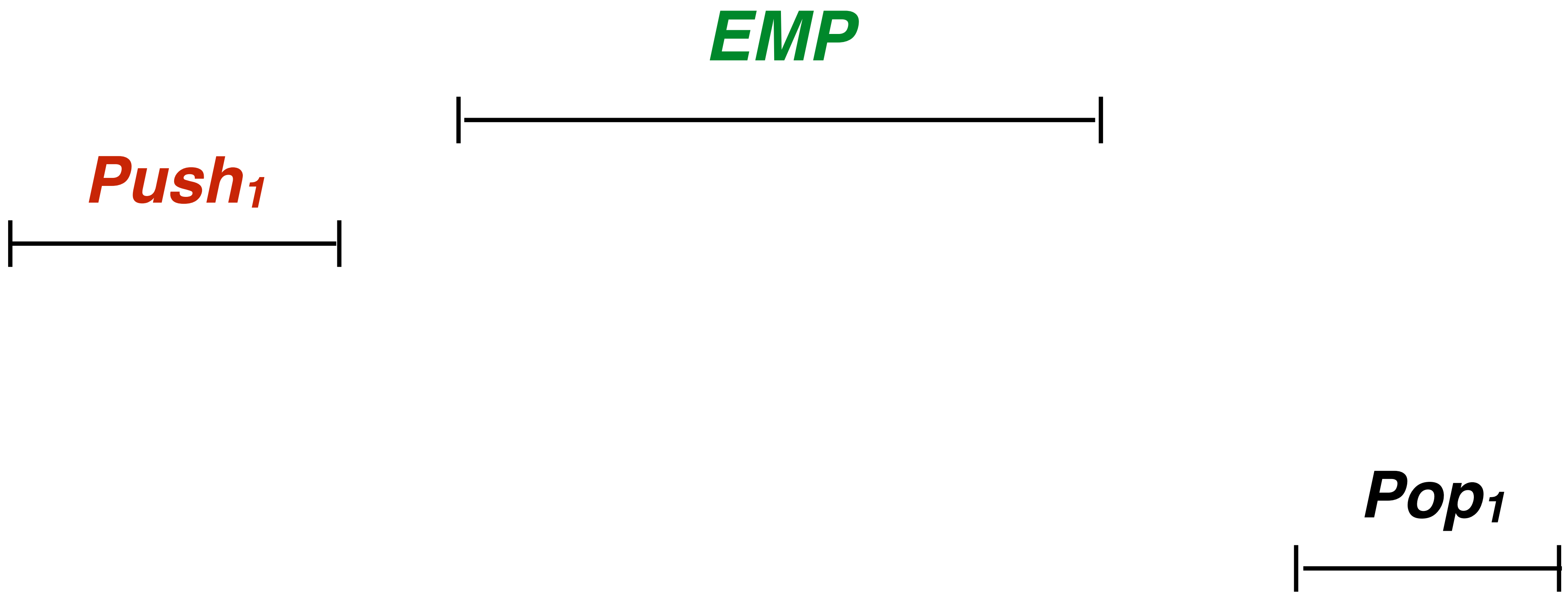
ret(Deq(1)) < call(Enq(1))



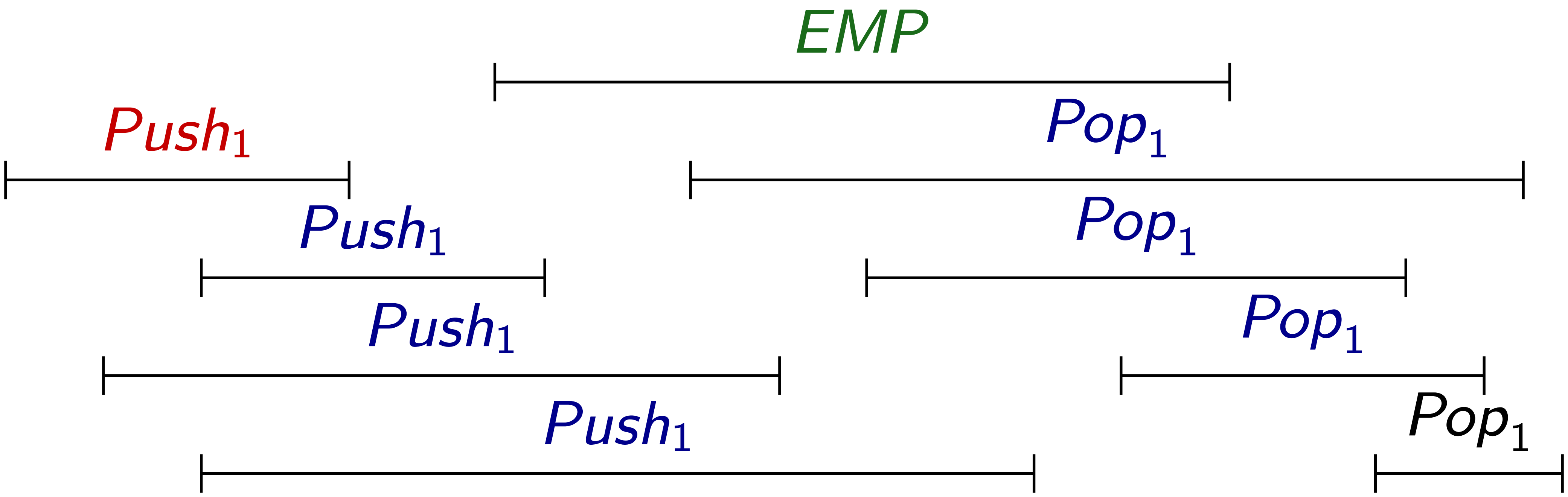
ret(Enq(1)) < call(Enq(2)) & ret(Deq(2)) < call(Deq(1))

- Regular Language over Call and Return events
- Only 3 different data values are needed

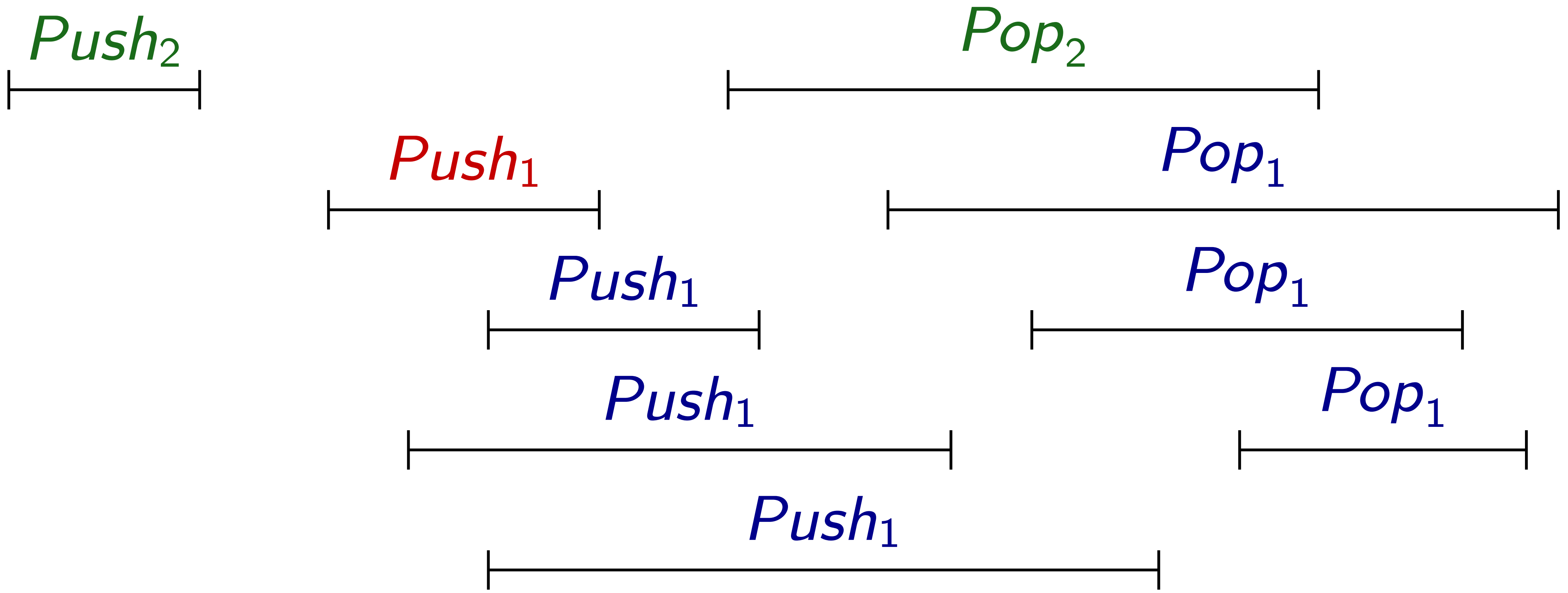
Empty Violation



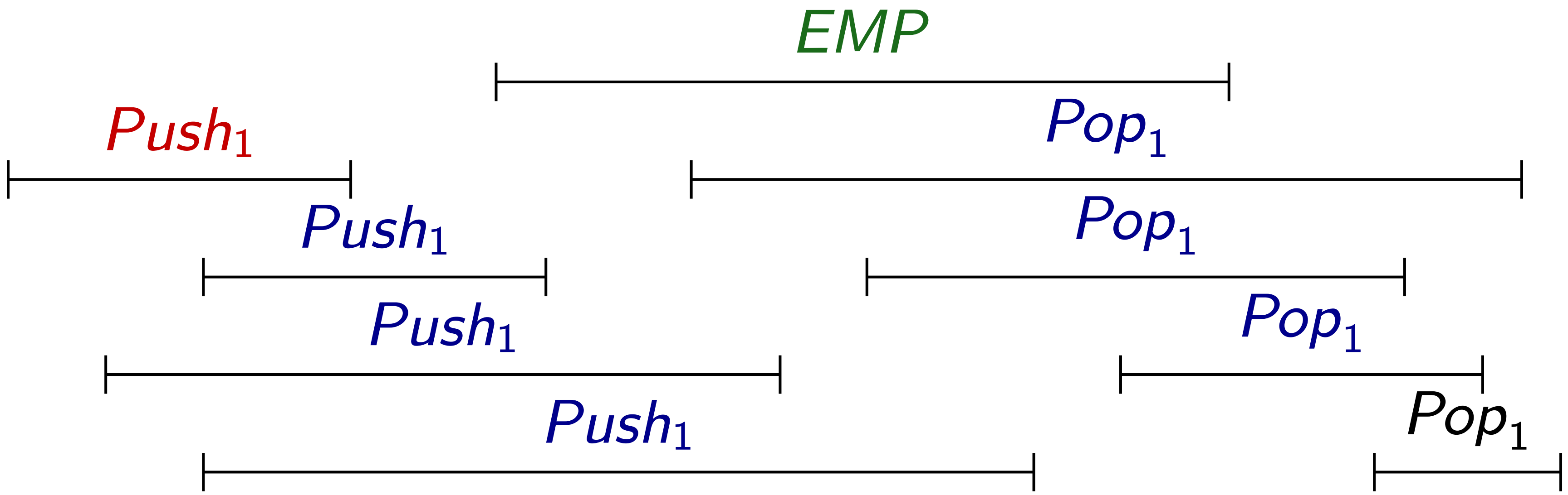
Empty Violation



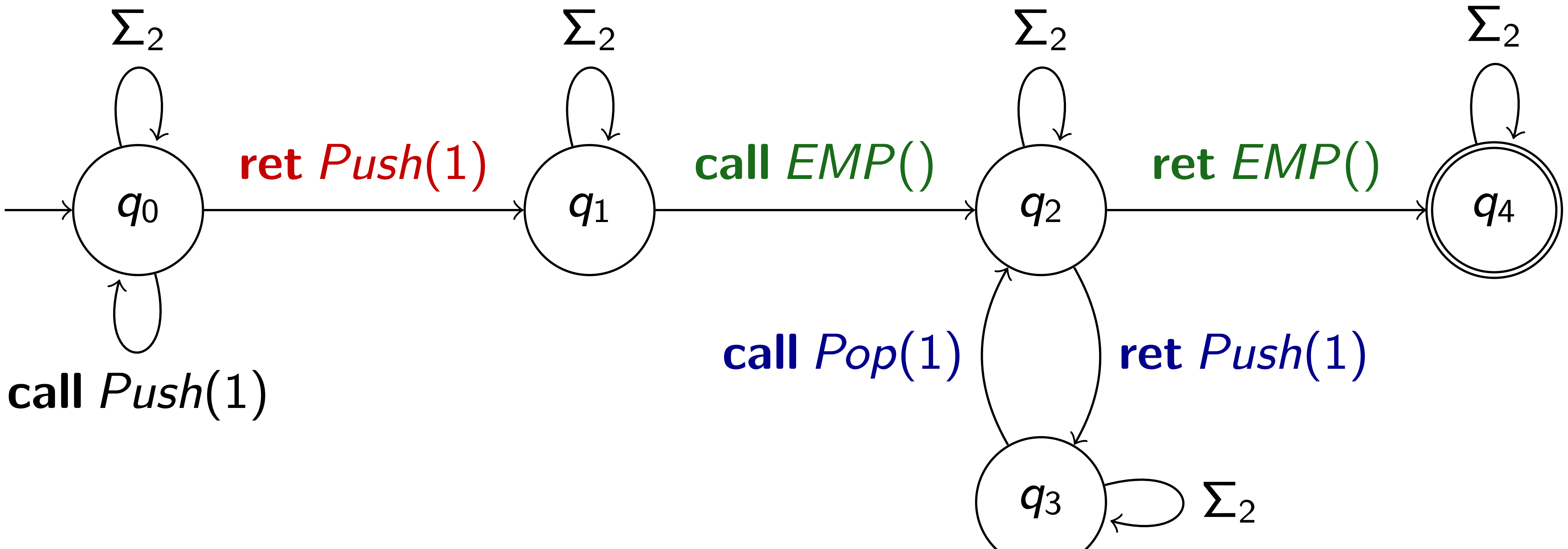
Order Violation cont. (stack)



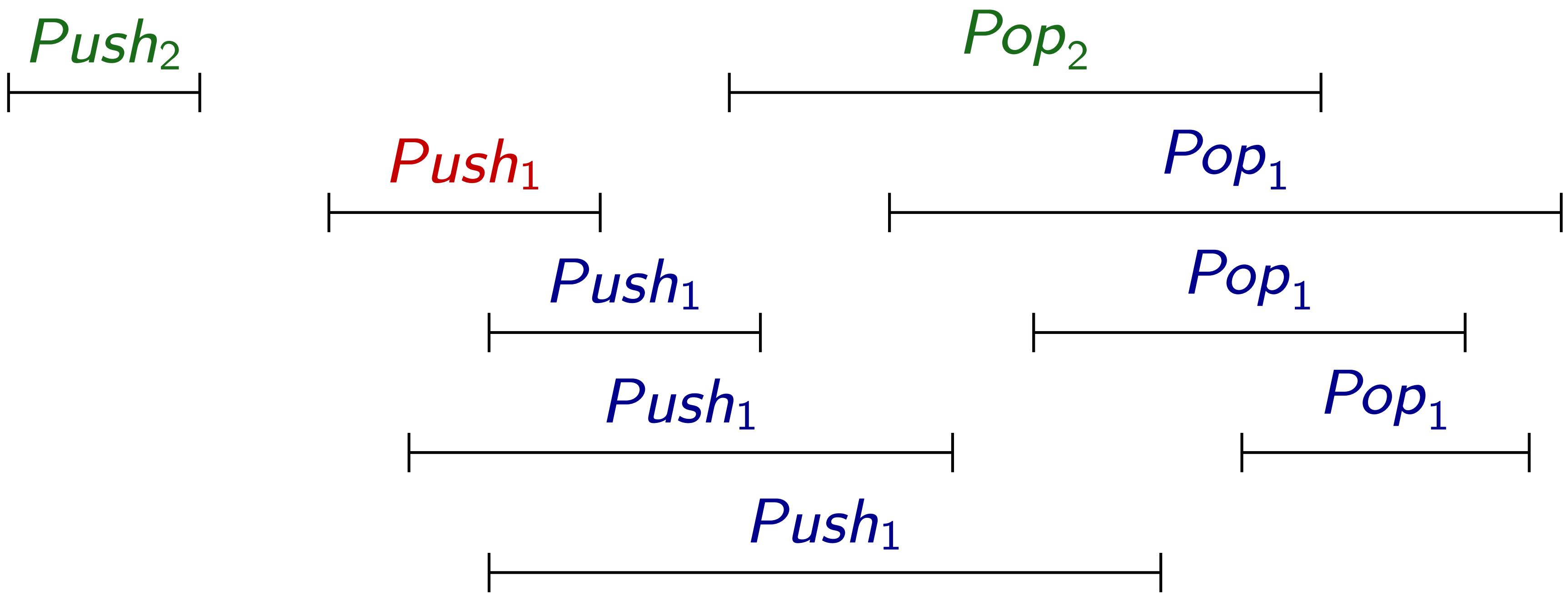
Automaton for Empty Violation



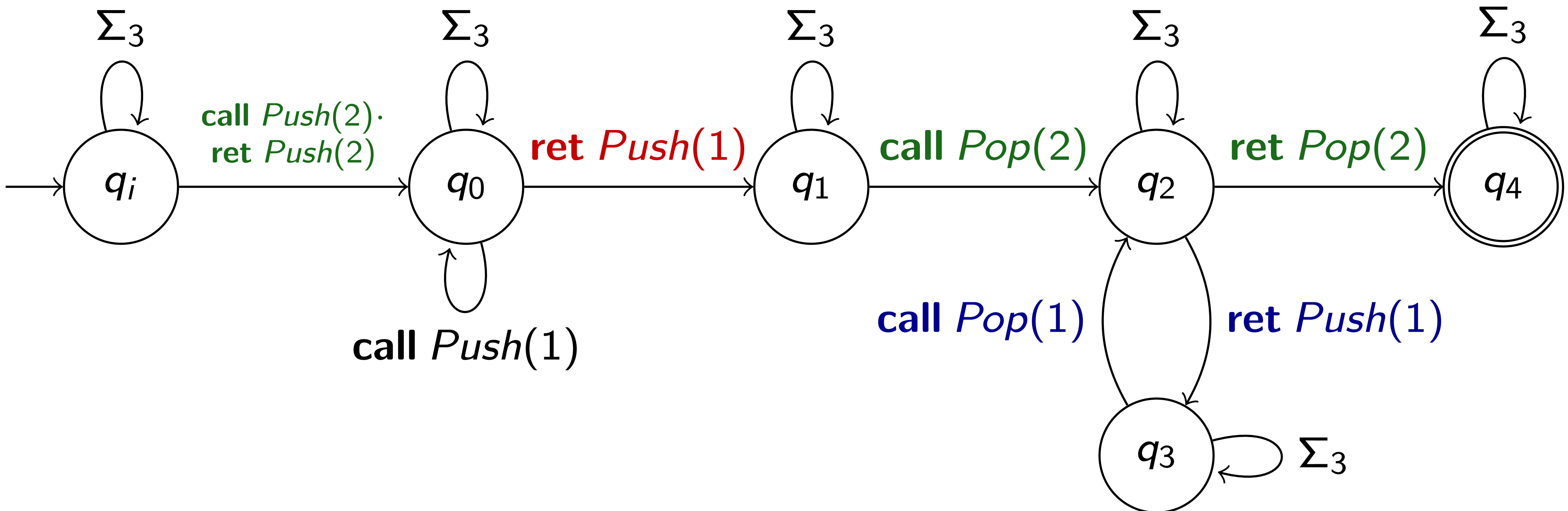
Recognized by:



Automaton for Push-Pop Order Violation



Recognized by:



Linearizability to State Reachability

Thm:

For each **S** in {Stack, Queue, Mutex, Register},

there is an automaton **A(S)** s.t.

for every data independent concurrent implementation **L**,

L is linearisable wrt S iff L intersected with A(S) is empty

Same complexity as state reachability

Under-approximate Analysis

[B, Emmi, Enea, Hamza, POPL'15]

- **Bounded information** about computations
- Useful for **efficient bug detection**

- **Bounding concept** for detecting linearizability violations?
- Should offer **good coverage**, and **scalability**

- **Interval-length** bounded analysis
- Based on characterising *linearizability as history inclusion*
- Monitor uses **counters**
- Allows for symbolic encodings
- Efficient static and dynamic analysis

Linearizability as a History Inclusion (Recall)

Consider an **abstract data structure**,
let **S** be its **sequential specification**,
and let **L_S** be a **sequential implementation** of S,
i.e., ***L_S* *satisfies S***

L_C reference concurrent implementation =
L_S + lock/unlock at beginning/end of each method

Lemma:
H(L_C) is the set histories that are linearised to a sequence in S

Thm: **L is linearisable wrt S iff H(L) is included in H(L_C)**

Abstracting Histories

Weakening relation

$h_1 \leq h_2$ (h_1 is **weaker than** h_2)

iff

h_1 has less constraints than h_2

Abstracting Histories

Weakening relation

$h_1 \leq h_2$ (h_1 is **weaker than** h_2)

iff

h_1 has less constraints than h_2

Lemma:

($h_1 \leq h_2$ and h_2 is in $H(L)$) ==> h_1 is in $H(L)$

Approximation Schema

Weakening function A_k, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a k s.t. $h = A_k(h)$

Approximation Schema

Weakening function A_k, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a k s.t. $h = A_k(h)$

Approximate History Inclusion Checking, for fixed $k \geq 0$

- Given a library L and a specification S
- Check: **Is there an h in H(L) s.t. A_k(h) is not in H(S)?**
- A_k(h) is not in H(S) => h is not in H(S) — Violation!

Histories are Interval Orders

Interval Orders = partial order (O, <) such that

(o1 < o1' and o2 < o2') implies (o1 < o2' or o2 < o1')

Prop: For every execution *e*, H(*e*) is an interval order

Notion of Length

Let h = (O,<) be an Interval Order (history in our case)

- Past of an operation: past(o) = {o’ : o’ < o}
- Lemma [Rabinovitch’78]:

  The set {past(o) : o in O} is linearly ordered

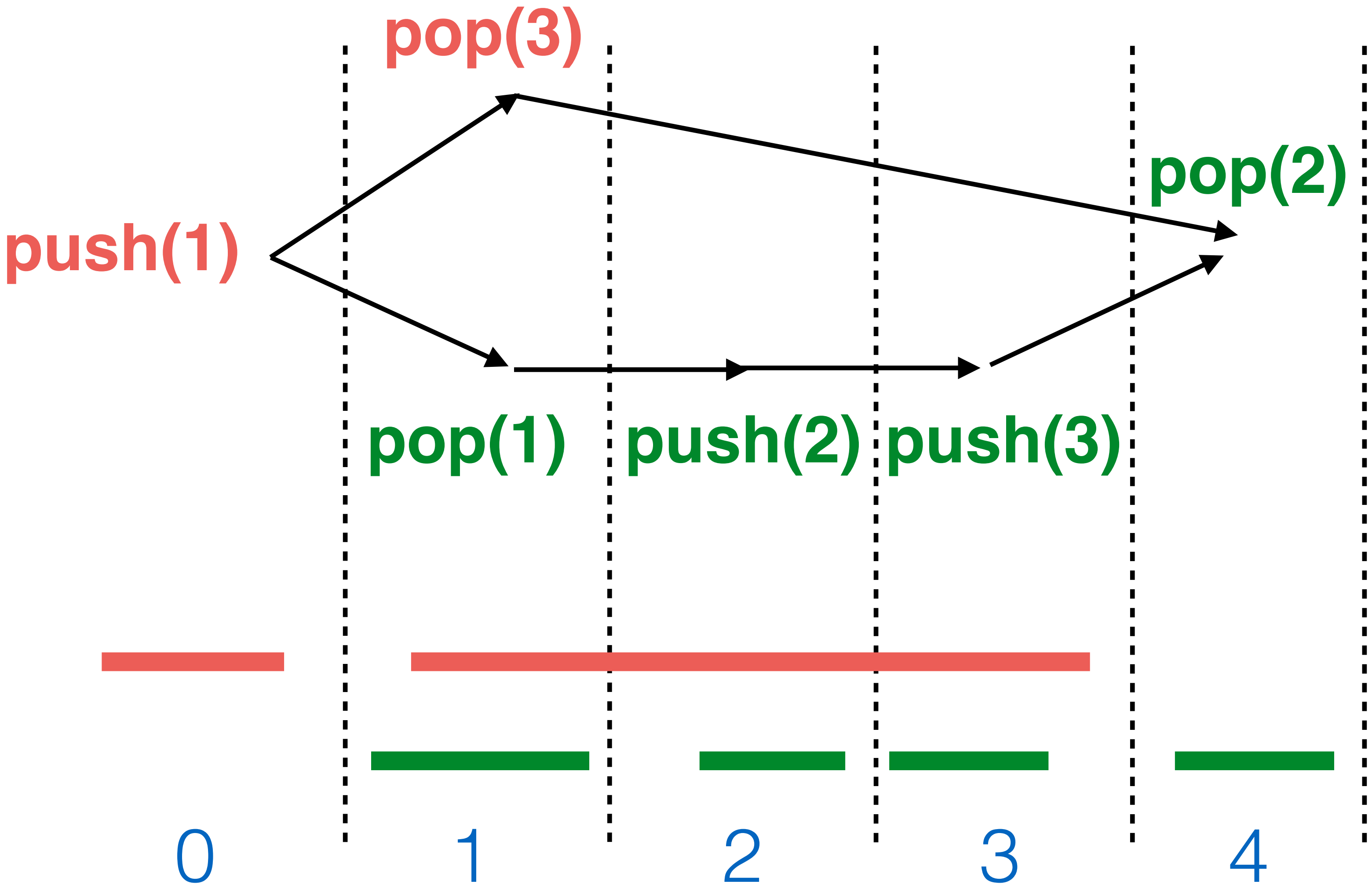
- The *length* of the order = number of pasts - 1

Canonical Representation of Interval Orders

- Mapping I : O —> $[n]^2$ where n = length(h) [Greenough '76]
- I(o) = [i, j], with i, j $\leq$ n, such that

i = |{past(o') : o' < o}| and

j = |{past(o') : not (o < o')}| - 1



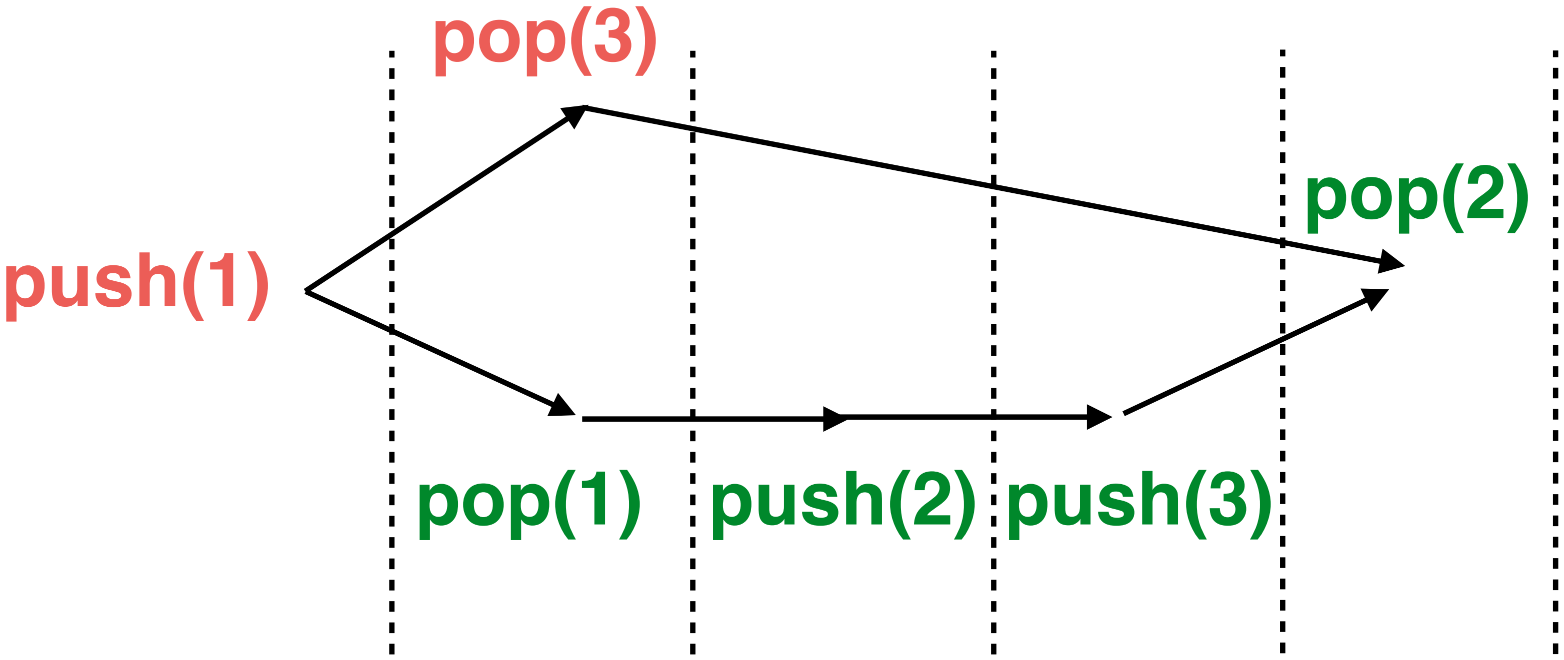
I(push(1)) = [0, 0]
I(pop(1)) = [1, 1]
I(push(2)) = [2, 2]
I(push(3) = [3, 3]
I(pop(3)) = [1, 3]
I(pop(2)) = [4, 4]

length = 4

Bounded Interval-length Approximation

Let A_k maps each h to some $h' \leq h$ of length k

=> Keep precise the information about the k last intervals



pop(3)

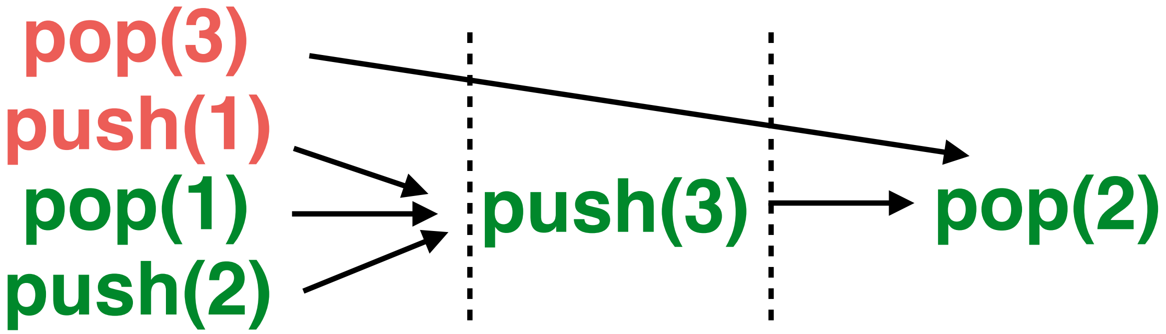
push(1)

pop(2)

pop(1) push(2) push(3)

I(push(1)) = [0, 0]
I(pop(1)) = [0, 0]
I(push(2)) = [0, 0]
I(push(3) = [1, 1]
I(pop(3)) = [0, 1]
I(pop(2)) = [2, 2]

K=2



pop(3)
push(1)
pop(1)
push(2)

push(3)

pop(2)

Counting Representation of Interval Orders

Count the number of occurrences of each operation type in each interval

- h = (O, <) an IO with canonical representation I:O—>$[k]^2$
- Associate a **counter** with each operation type and interval
- **∏(h) is the Parikh image of h**
- It represents the multi-set **{** [label(o), I(o)] : o in O **}**

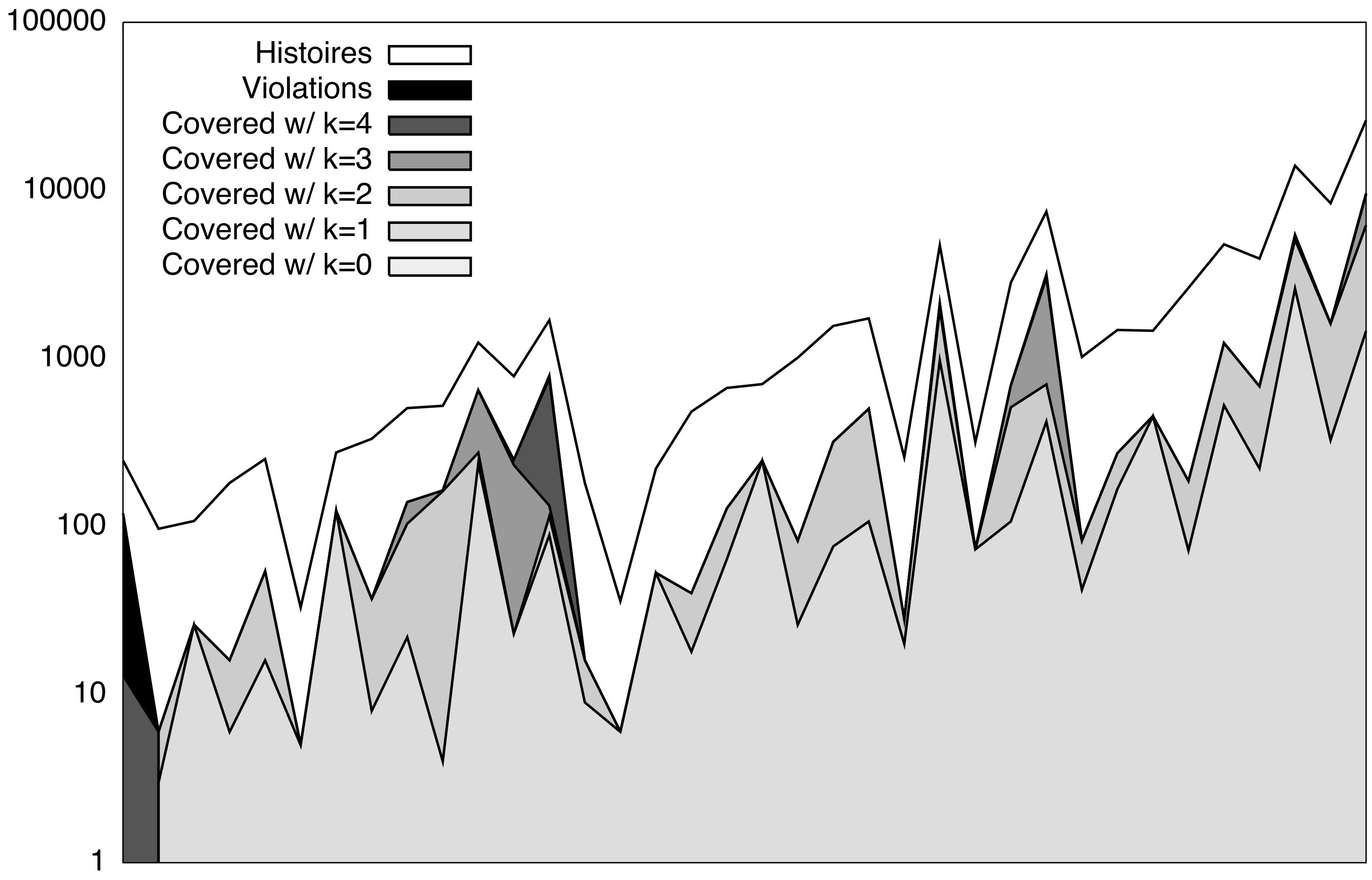
Prop: H_k(e) is in H_k(L) iff ∏(H_k(e)) is in ∏(H_k(L))

Reduction to Reachability with Counters

$H_k(L)$ subset of $H_k(S)$
iff
$\Pi(H_k(L))$ subset of $\Pi(H_k(S))$

- Consider **k-bounded-length abstract histories**
- Track histories of L using a **finite number of counters**
- Use an **arithmetic-based representation of $\Pi(H_k(S))$**
- Π(Hk(S)) can be either computed, or given manually
- Check that **$\Pi(H_k(S))$ is an invariant**

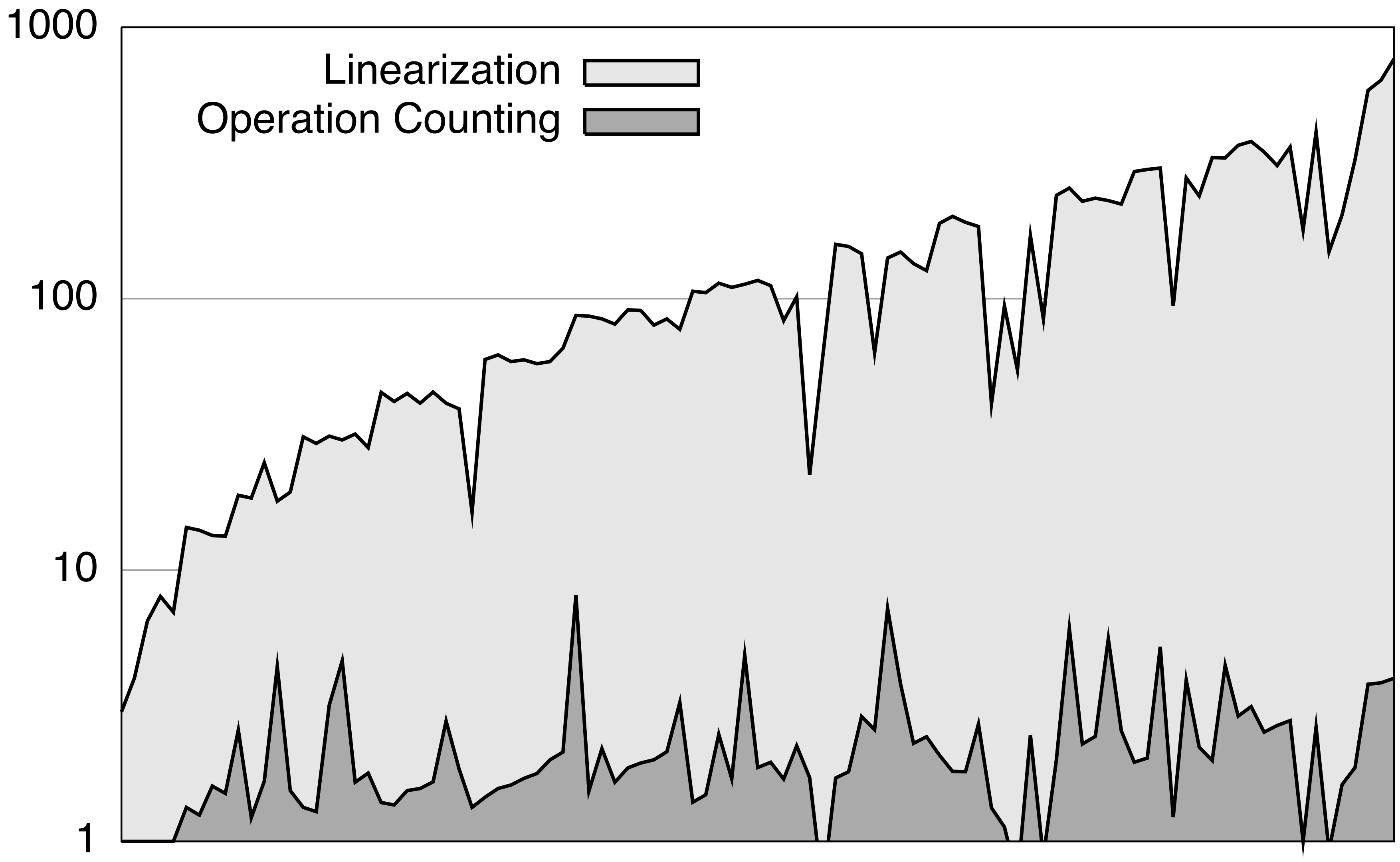
Experimental Results: Coverage



Comparison of violations covered with k ≤ 4

- Data point: Counts in logarithmic scale over all executions (up to 5 preemptions) on Scal’s nonblocking bounded-reordering queue with ≤4 enqueue and ≤4 dequeue
- x-axis: increasing number of executions (1023-2359292)
- White: total number of unique histories over a given set of executions
- Black: violations detected by traditional linearizability checker (e.g., Line-up)

Experimental Results: Runtime Monitoring



Comparison of runtime overhead
between Linearization-based monitoring and Operation counting

- Data point: runtime on logarithmic scale, normalised on unmonitored execution time
- Scal's nonblocking Michael-Scott queue, 10 enqueue and 10 dequeue operations.
- x-axis is ordered by increasing number of operations

Experimental Results: Static Analysis

Library	**Bug**	**P**	**k**	**m**	**n**	**Time**
Michael-Scott Queue	B1 (head)	2x2	1	2	2	24.76s
Michael-Scott Queue	B1 (tail)	3x1	1	2	3	45.44s
Treiber Stack	B2	3x4	1	1	2	52.59s
Treiber Stack	B3 (push)	2x2	1	1	2	24.46s
Treiber Stack	B3 (pop)	2x2	1	1	2	15.16s
Elimination Stack	B4	4x1	0	1	4	317.79s
Elimination Stack	B5	3x1	1	1	4	222.04s
Elimination Stack	B2	3x4	0	1	2	434.84s
Lock-coupling Set	B6	1x2	0	2	2	11.27s
LFDS Queue	B7	2x2	1	1	2	77.00s

- Static detection of injected refinement violations with CSeq & CBMC.
- Program Pij with i and j invocations to the push and pop methods, explore n-round round-robin schedules with m loop iterations unrolled, with monitor for Ak.
- Bugs: (B1) non-atomic lock, (B2) ABA bug, (B3) non-atomic CAS operation, (B4) misplaced brace, (B5) forgotten assignment, (B6) misplaced

Conclusion

- **Linearizability** checking is **hard/undecidable** in general
- But **tractable reductions to state reachability** are possible
- Consider **relevant** classes of **concurrent objects**:
  - Covers common structures such as stacks and queues
  - Finite-state monitor: **Linear reduction to state reachability**
  - Decidability for unbounded number of threads
- Consider **relevant types executions**:
  - Bounding principle based on an abstraction of histories
  - Monitor: Counter machine
  - Use symbolic techniques => Static and dynamic analysis
  - Good coverage, scalable monitoring

Some future work

- Extend the first approach to other structures, e.g., sets.
- Specification language+systematic construction of monitors.
- Combine our approach with providing linearisation policies

[Abdulla et al., TACAS'13, SAS'16

Some future work

- Extend the first approach to other structures, e.g., sets.
- Specification language+systematic construction of monitors.
- Combine our approach with providing linearisation policies

  [Abdulla et al., TACAS'13, SAS'16

- Extend it to distributed (replicated) data structures

  Weaker consistency notions are needed:
  Eventual consistency, causal consistency, etc.

  - Eventual consistency —> Reachability, Model-checking

    [B., Enea, Hamza, POPL'14]

  - Causal consistency ?

    [Recent work for the Read-Write memory/Key-value store]