Parallele Programmierung FS25

Exercise Session 11

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Assignment 10
- Theory
- Intro Assignment 11
- Exam questions
- Kahoot

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too
- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9

Organisation

- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9
- Falls ihr Feedback möchtet kommt bitte zu mir

Organisation

- Wo sind wir jetzt?

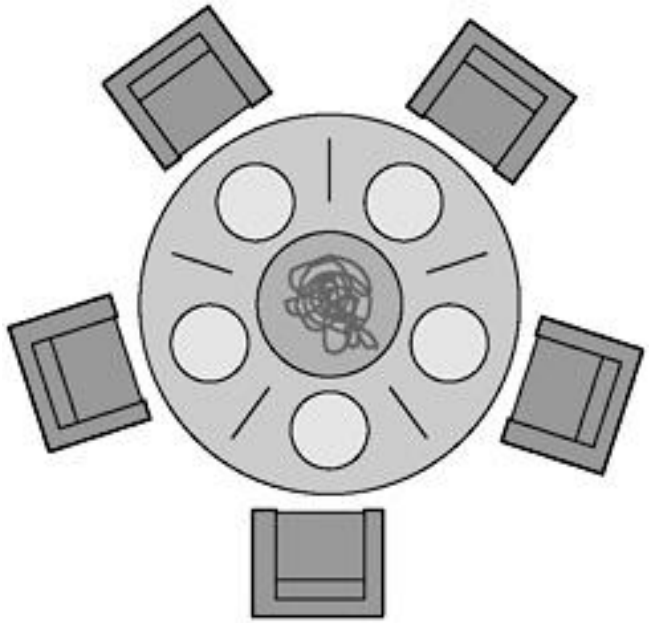
Reader Writer Lock
Lock granularity
Coarse-grained, Fine-grained, optimistic locking, lazy locking
Concurrent LinkedList

To come: lock free synchronization, lock free data structures, Linearizability, Consensus

Plan für heute

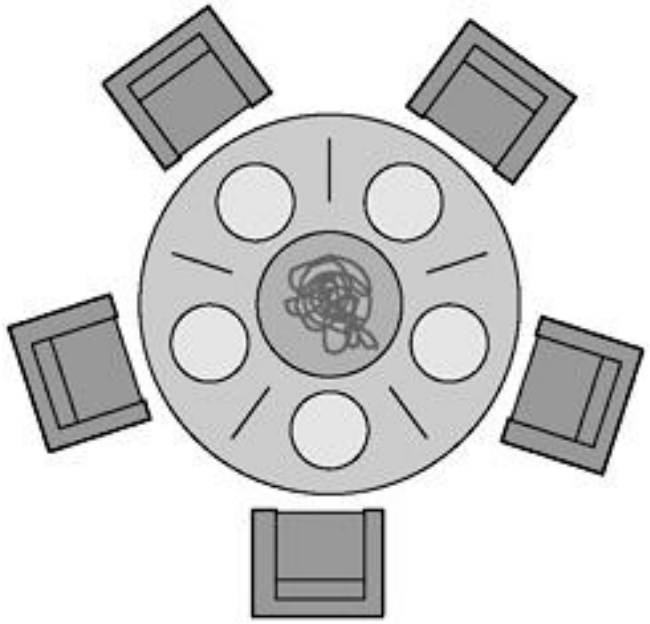
- Organisation
- **Nachbesprechung Assignment 10**
- Theory
- Intro Assignment 11
- Exam questions
- Kahoot

Task 1 - Dining Philosophers



Originally proposed by E. W. Dijkstra
Imagine five philosophers who spend their lives thinking and eating.
They sit around a circular table with five chairs with a big plate of spaghetti.
However, there are only five chopsticks available.

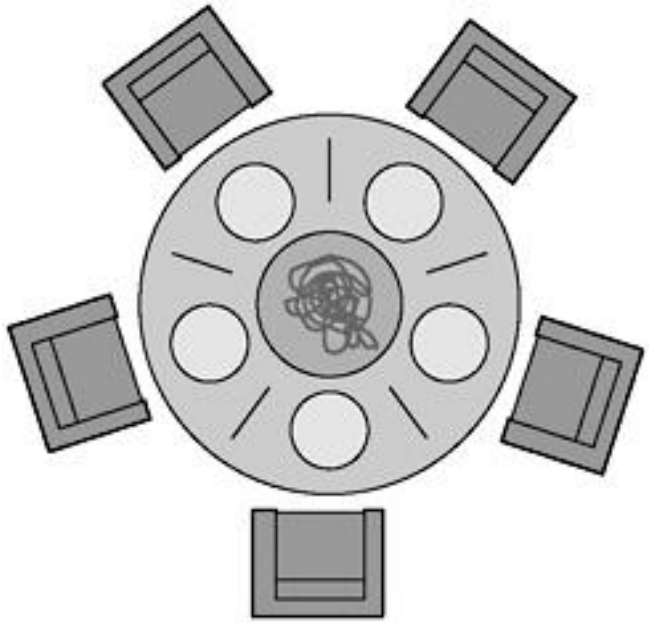
Task 1 - Dining Philosophers



Each philosopher thinks and when he gets hungry picks up the two chopsticks closest to him.

- If a philosopher can pick up BOTH chopsticks, he eats for a while.
- After a philosopher finishes eating, he puts down the chopsticks and starts to think again.

Find a solution that…



- Makes deadlocks impossible
- Has no starvation
- More than one parallel eating philosopher is possible

Assignment 10

Dining Philosophers:

```
while(true) {
    think();
    acquire_fork_on_left_side();
    acquire_fork_on_right_side();
    eat();
    release_fork_on_right_side();
    release_fork_on_left_side();
}
```

One philosophers left side is anothers right side!

But we take left first, then right. So we hold one fork, then wait – leads to cycle in dependency graph.

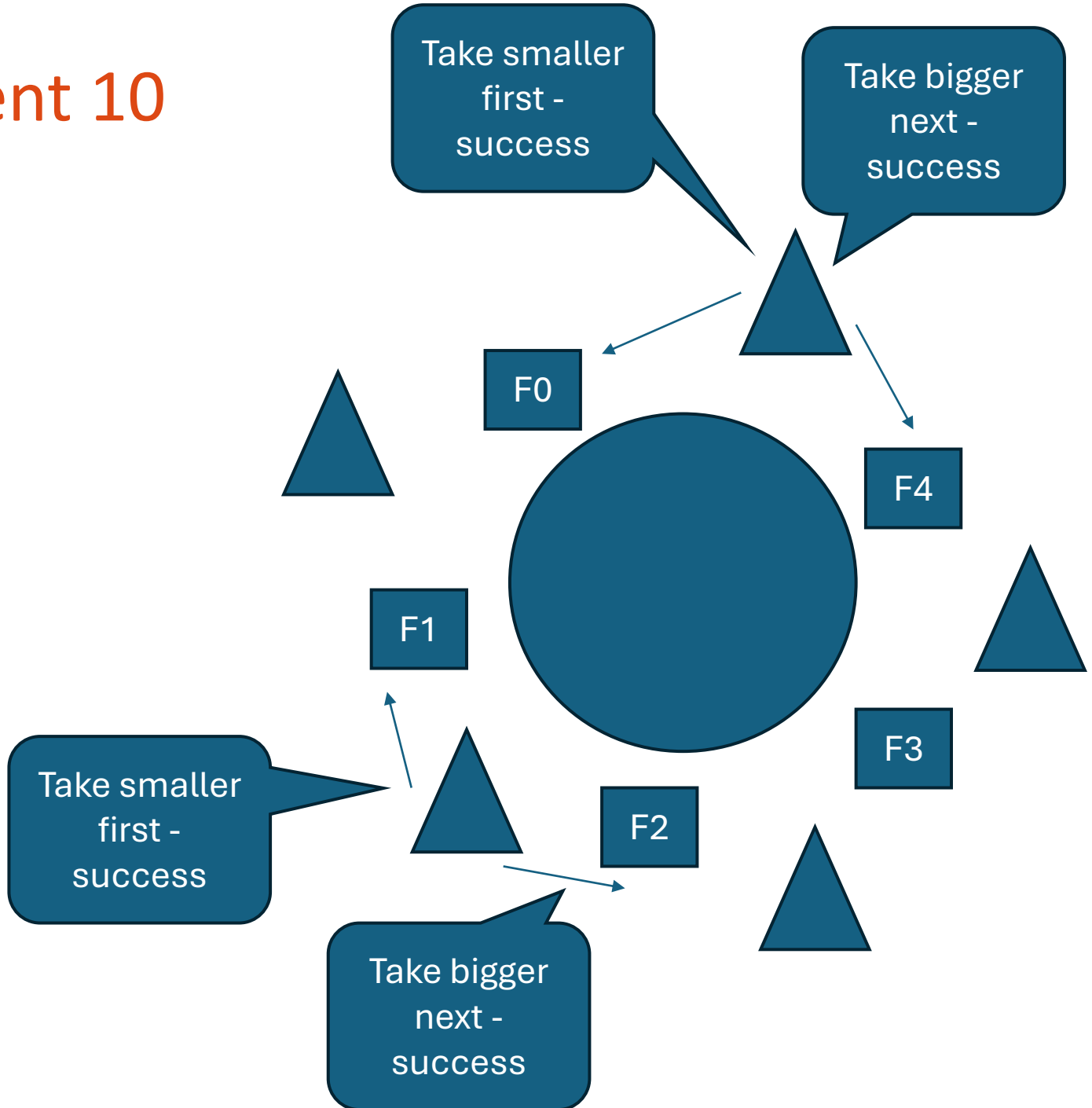
Assignment 10

Dining Philosophers:

- To avoid cyclic dependencies: Lock-ordering!

- Number all forks, take the one with smaller number first.
- Same principle we saw with bank-account already!

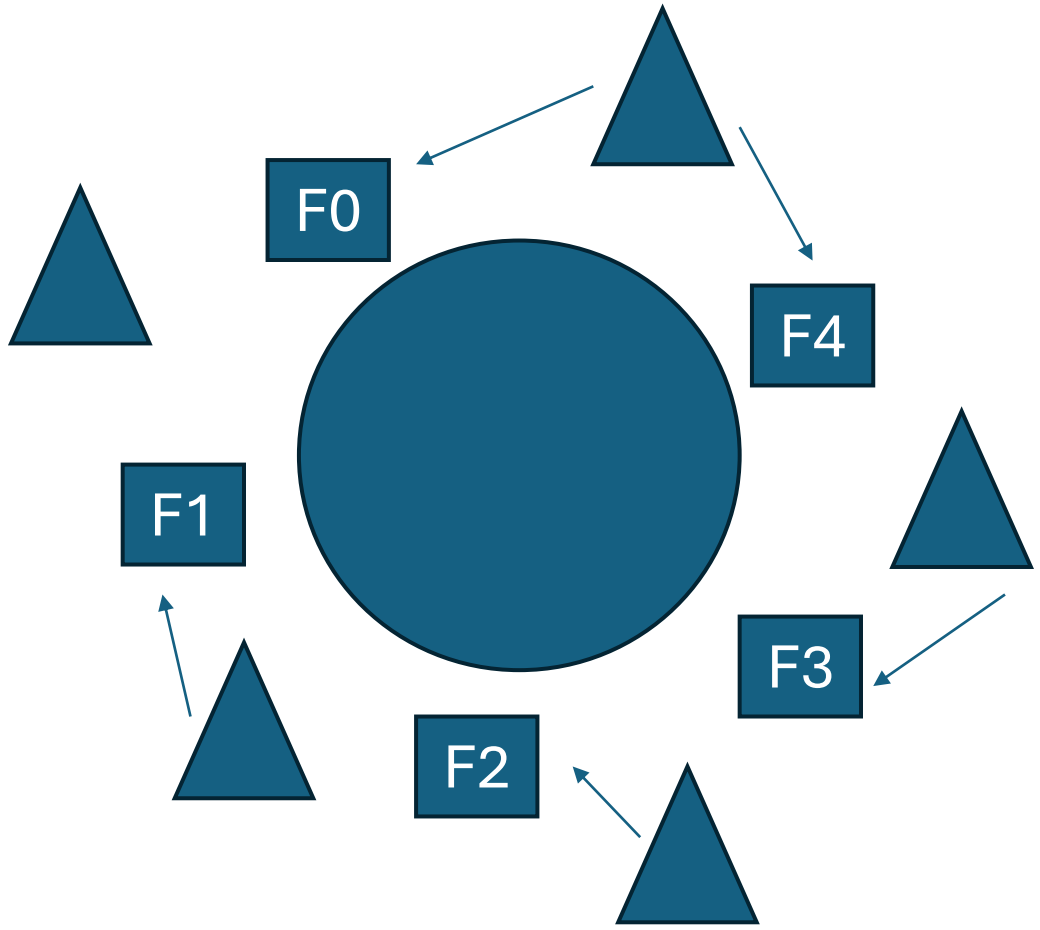
Assignment 10



Two can eat at the same time.

Three is impossible (would need six forks).

Assignment 10



Now only one is eating.

All others have to wait -
Not great, not terrible (no deadlock!)

Assignment 10

Dining Philosophers:

- The only way to ensure that two can always eat at the same time is to introduce additional elements (communication, a waiter, etc.)

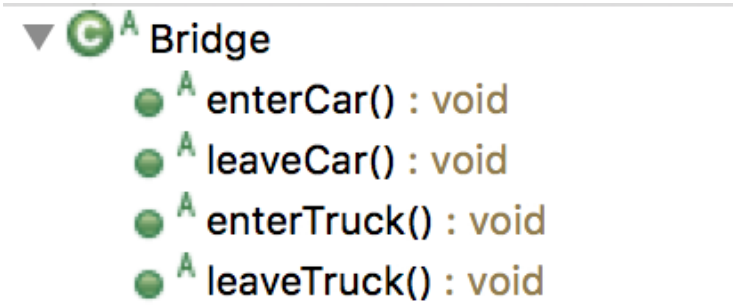
Now only one is eating.

All others have to wait -
Not great, not terrible (no deadlock!)

Task 2 – Monitors, Conditions and Bridges

Only either 3 cars or one truck may be on the bridge at each moment.

Implement Classes `BridgeMonitor` and `BridgeCondition`



How to Test my Implementation?
Implement method `invariant()` to check if the state is valid: at the end of a method there are never too many cars or trucks on the bridge

Assignment 10 – Bridge with monitor

```
public class BridgeMonitor extends Bridge {

    private int carCount = 0;
    private int truckCount = 0;
    private final Object monitor = new Object();

    public void enterCar() throws InterruptedException {
        synchronized(monitor)
        {
            while (carCount >= 3 || truckCount >= 1) {
                    monitor.wait();
            }
            carCount++;
        }
    }

    public void leaveCar() {
        synchronized (monitor) {
            carCount--;
            monitor.notifyAll();
        }
    }
```

Is this really needed?

Why notifyAll()?
We only want to wake up one car or maybe a truck (if carCount ==0)

Assignment 10 – Bridge with condition

```
public class BridgeCondition extends Bridge {

    final Lock bridgeLock = new ReentrantLock();
    Condition truckCanEnter = bridgeLock.newCondition();
    Condition carCanEnter = bridgeLock.newCondition();

    volatile private int carCount = 0;
    volatile private int truckCount = 0;
```

Make two separate groups of “waiters”

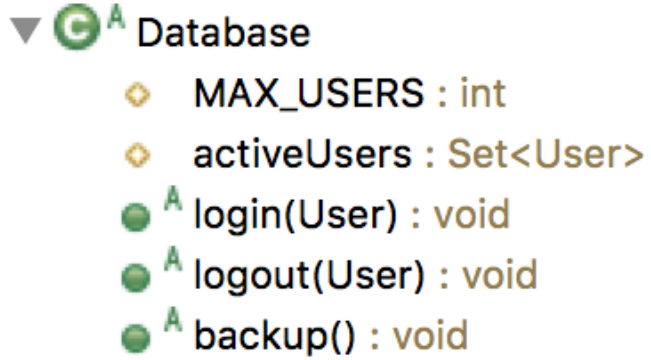
Assignment 10 – Bridge with condition

```
public void enterCar() throws InterruptedException {
    bridgeLock.lock();
    while (carCount >= 3 || truckCount >= 1) {
        carCanEnter.await();
    }
    carCount++;
    bridgeLock.unlock();
}

public void leaveCar() {
    bridgeLock.lock();
    carCount--;
    if (carCount == 0)
        truckCanEnter.signalAll();
    if (carCount < 3)
        carCanEnter.signalAll();
    bridgeLock.unlock();
}
```

Choose who to wake up based on conditions.

Task 3 – Semaphores and Databases

Use semaphores to implement login and logout database functionality that supports up to 10 concurrent users



Use barrier to implement 2-phase backup functionality.

Task 3 – Semaphores and Databases

Implement Classes MySemaphore and MyBarrier

Use monitors for both to avoid busy loop

- Put processes to sleep, when there is no entry into semaphore
- Wake up a waiting process when releasing a semaphore

```
acquire(S) {
        wait until S > 0
        dec(S)
}

release(S) {
        inc(S)
}
```

Assignment 10 – Semaphore implementation

```
public class MySemaphore {
    private volatile int count;

    public MySemaphore(int maxCount) {
        this.count = maxCount;
    }

    public void acquire() throws Inte
        synchronized (this) {
            while (count == 0) {this.wait();}
            count--;
        }
    }

    public void release() {
        synchronized (this) {
            count++;
            this.notifyAll();
        }
    }
}
```

Why a while loop here and not an if?

Does this have to be notifyAll()?

Note that we use “this” here – could also have created new object (see Bridge monitor)

Assignment 10 – Barrier implementation

```
synchronized void await() throws InterruptedException {
  while (draining) {
    wait();
  }
  ++i;
  while (i < n && !draining) {
    wait();
  }
  if (i-- == n) {
    draining = true;
    notifyAll();
  }
  if (i == 0) {
    draining = false;
    notifyAll();
  }
}
```

Why do we distinguish between draining and non-draining?

Two-Phase Barrier

init

barrier

```
mutex=1; barrier1=0; barrier2=1; count=0

acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
signal(mutex)

acquire(barrier2); release(barrier2)
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

Plan für heute

- Organisation
- Nachbesprechung Assignment 10
- **Theory & Intro Assignment 11**
- Exam questions
- Kahoot

Assignment 11

- Implement SortedList with different lock strategies
- Exercise about effective use of locks
  - Coarse grained vs. fine grained locks
  - Tricks to avoid locking altogether for certain operations
- Measure the performance impact of your implementation choice

SortedListInterface

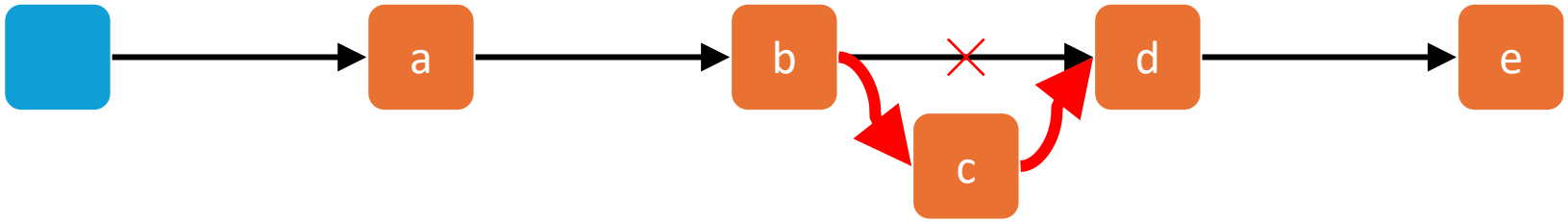
Add, Remove and Find unique elements in a **sorted linked list**.

```
add(c)
```

SortedListInterface

Add, Remove and Find unique elements in a **sorted linked list**.

`add(c)`



find b and d

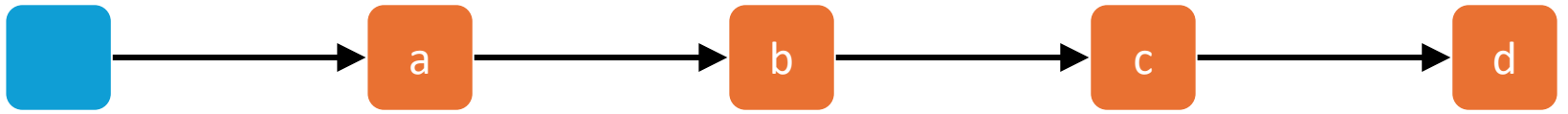
b.next=c

c.next=d

SortedListInterface

Add, Remove and Find unique elements in a **sorted linked list**.

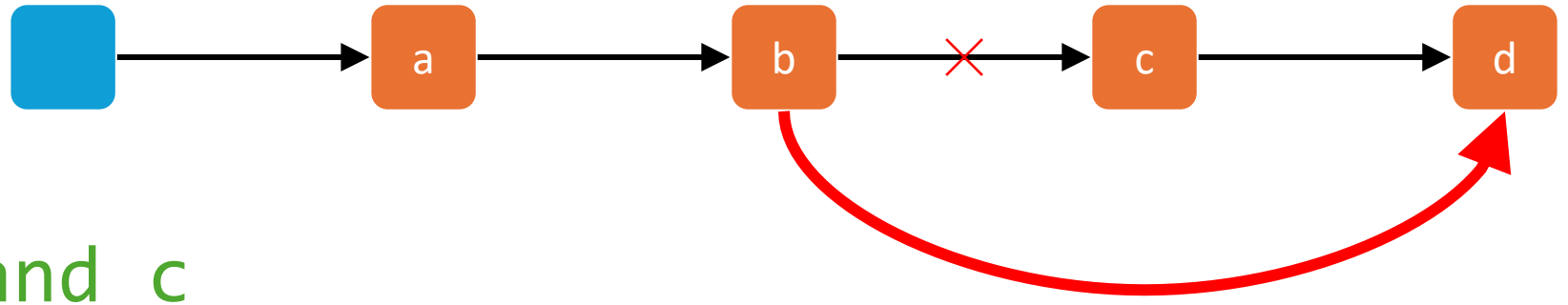
`remove(c)`



SortedListInterface

Add, Remove and Find unique elements in a **sorted linked list**.

`remove(c)`



`find b and c`

`b.next=c.next`

List and Node

```
public interface SortedListInterface<T extends Comparable<T>> {

      public boolean add (T item);
      public boolean remove (T item);
      public boolean contains (T item);

}
```

Make sure we can sort the entries in the list!

Implement those methods in a thread-safe way

Implementation tipps

- Keep an abstract Node to store list element:

```
private class Node {
        public Node next ;
        public T item;
}
```

- Code is simpler if we always have two **sentinel nodes** in the list:

```
public SequentialList() {
        first = new Node(Integer.MIN_VALUE);
        first.next = new Node(Integer.MAX_VALUE);
}
```

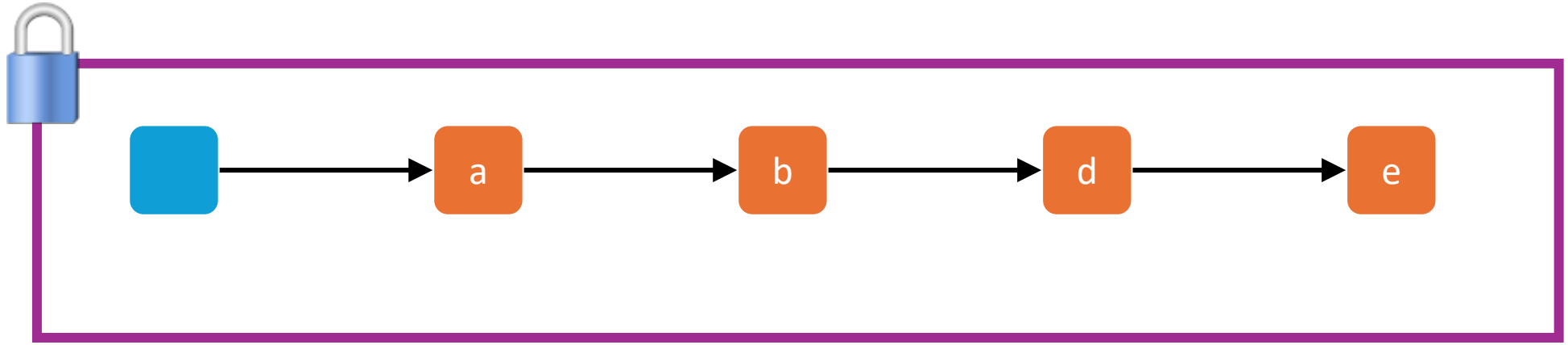
Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```

add(c)

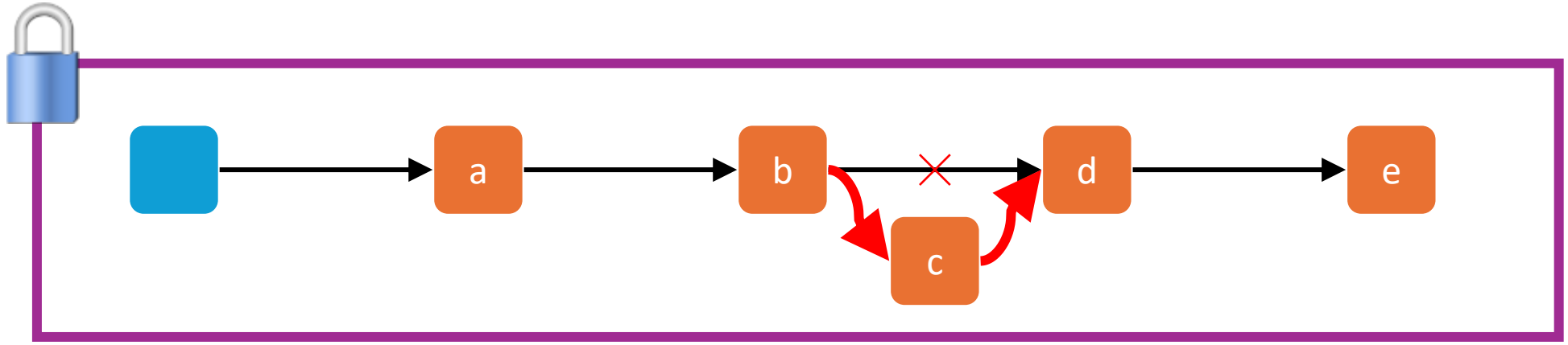
Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```



Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```



Simple, but a bottleneck for many threads, why?

Fine grained Locking

Often more intricate than visible at a first sight

- requires careful consideration of special cases

Idea: split object into pieces with separate locks

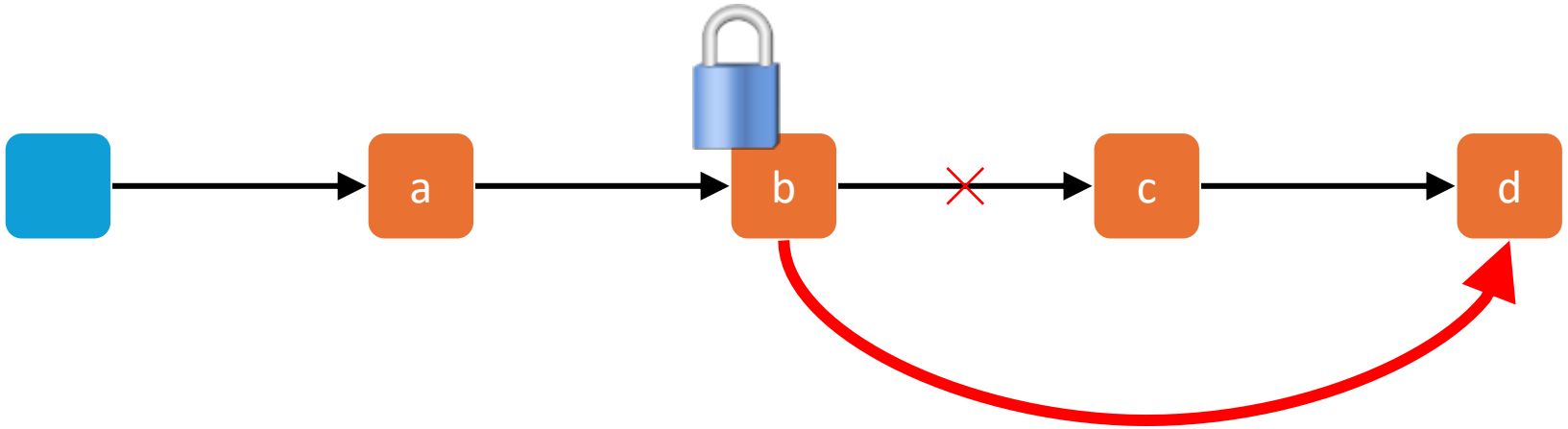
- no mutual exclusion for algorithms on disjoint pieces

Let's try this

`remove(c)`

Let's try this

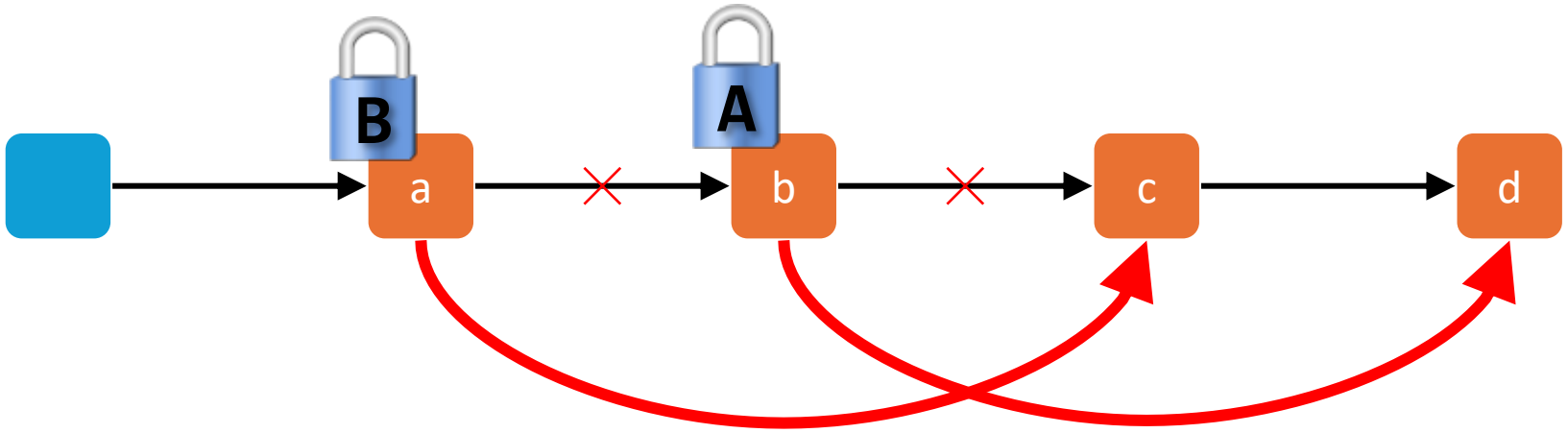
`remove(c)`



`Locking the predecessor is ok?`

Let's try this

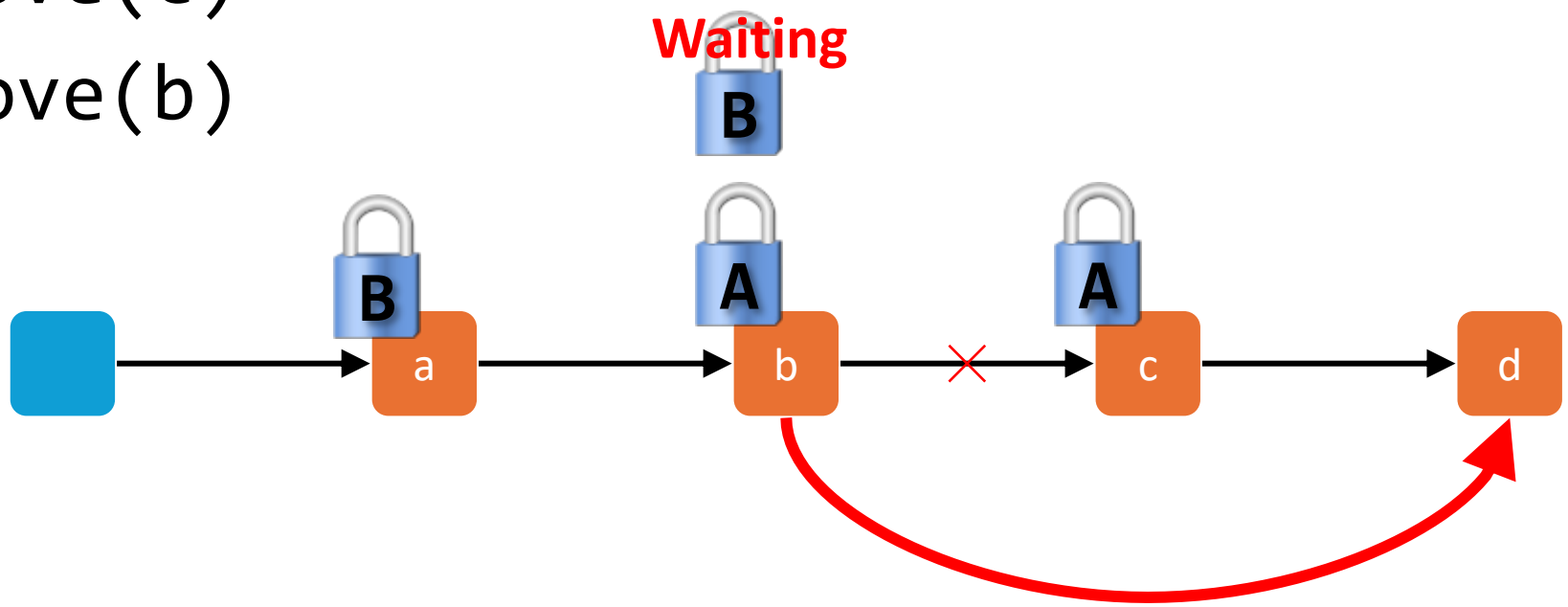
```
A: remove(c)
B: remove(b)
```



c not deleted!

Let's try this

```
A: remove(c)
B: remove(b)
```

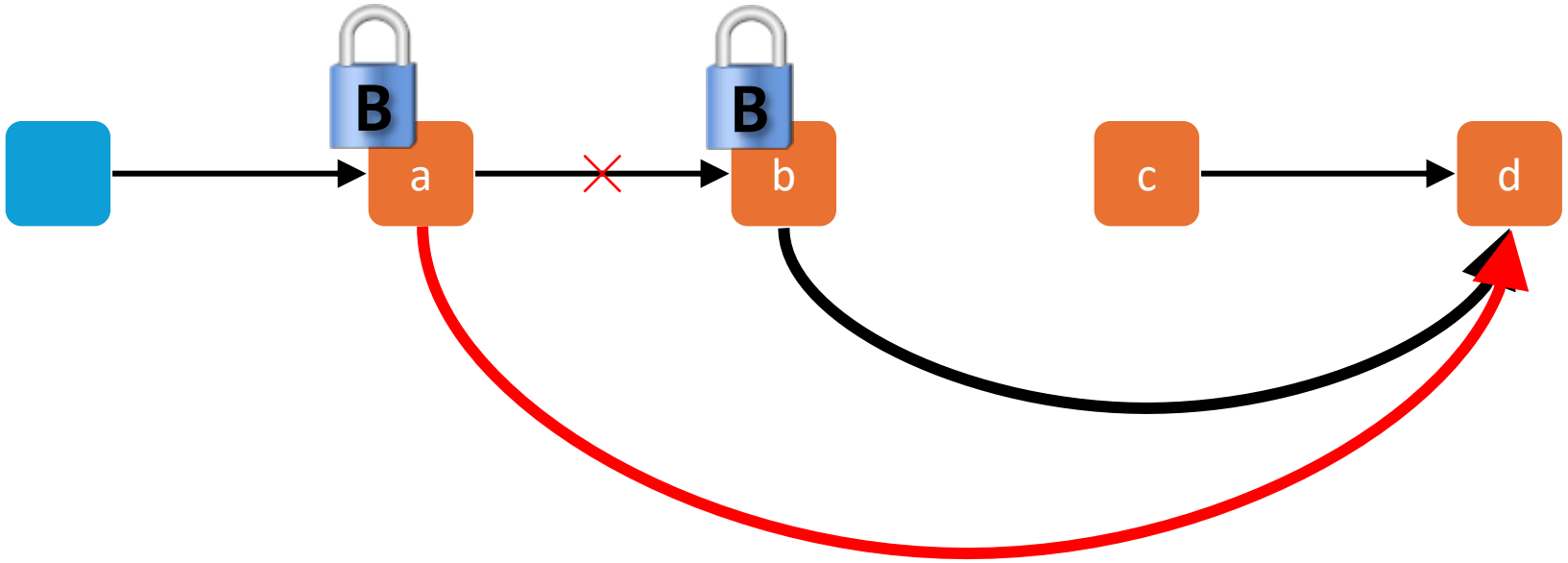


- c not deleted!

When removing, lock the **successor** defensively.

Let's try this

```
A: remove(c)
B: remove(b)
```



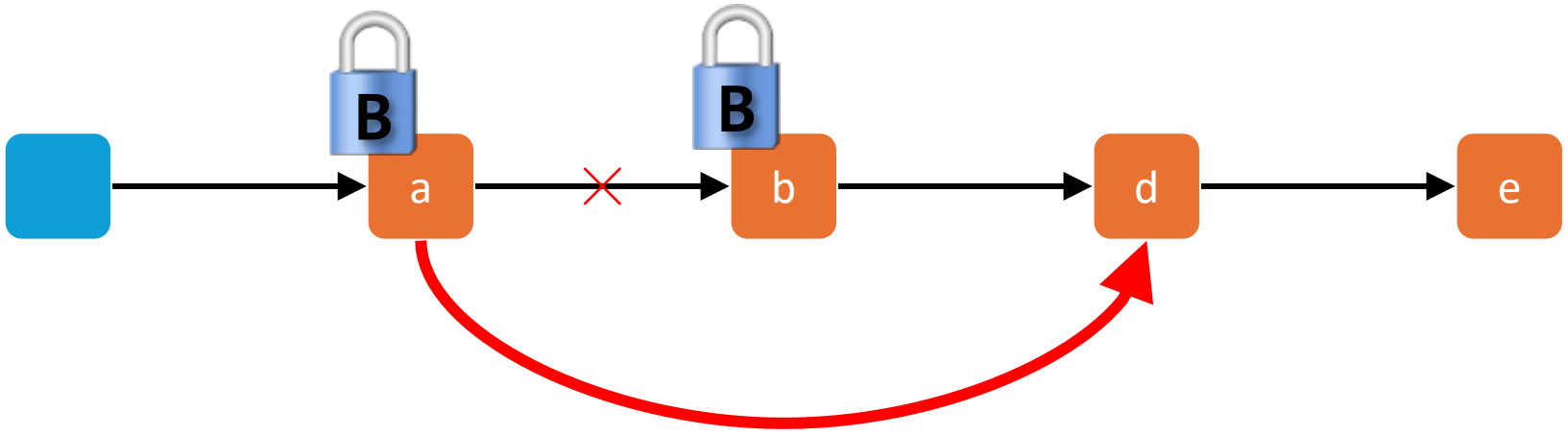
- c not deleted!

When removing, lock the **successor** defensively.

What's the problem?

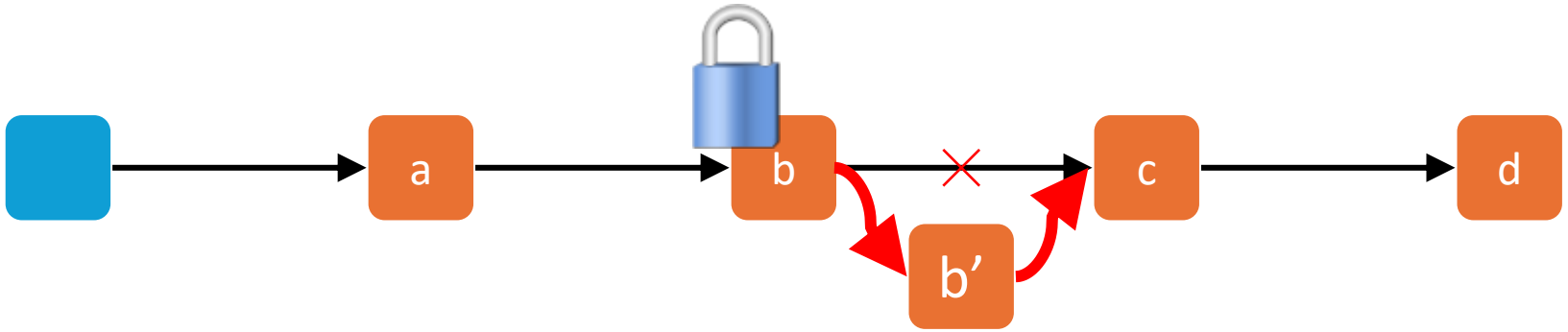
When deleting, the next field of next is read, i.e. next also has to be protected.

```
find a and b
a.next=b.next
```



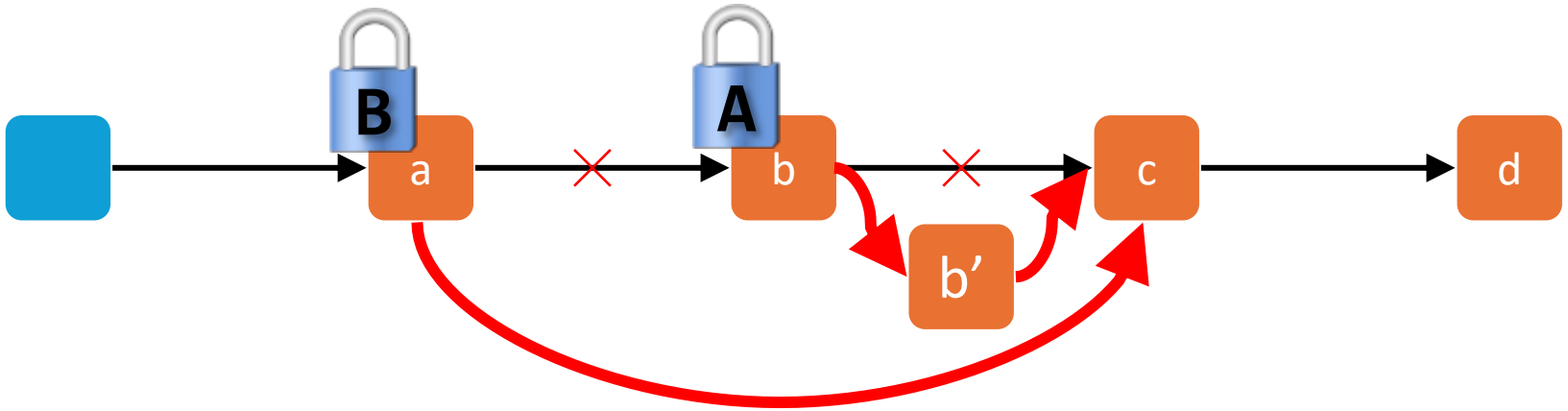
What about add?

```
add(b’)
```



What about add?

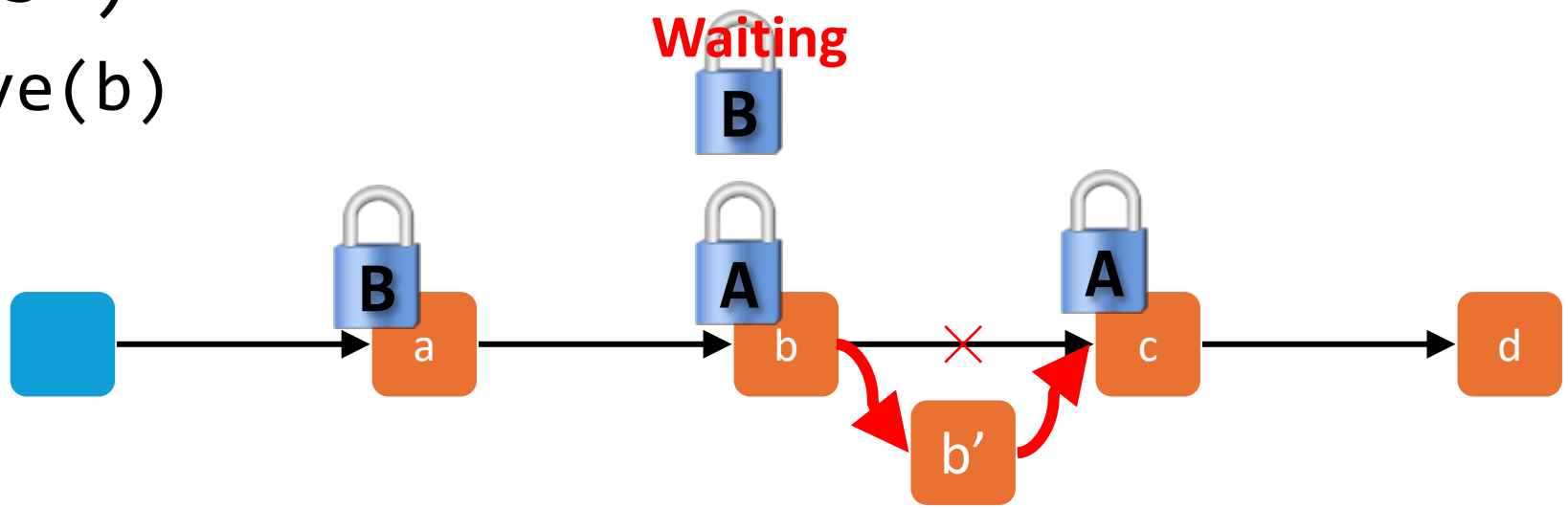
```
A: add(b’)
B: remove(b)
```



- b’ not added!

What about add?

```
A: add(b’)
B: remove(b)
```

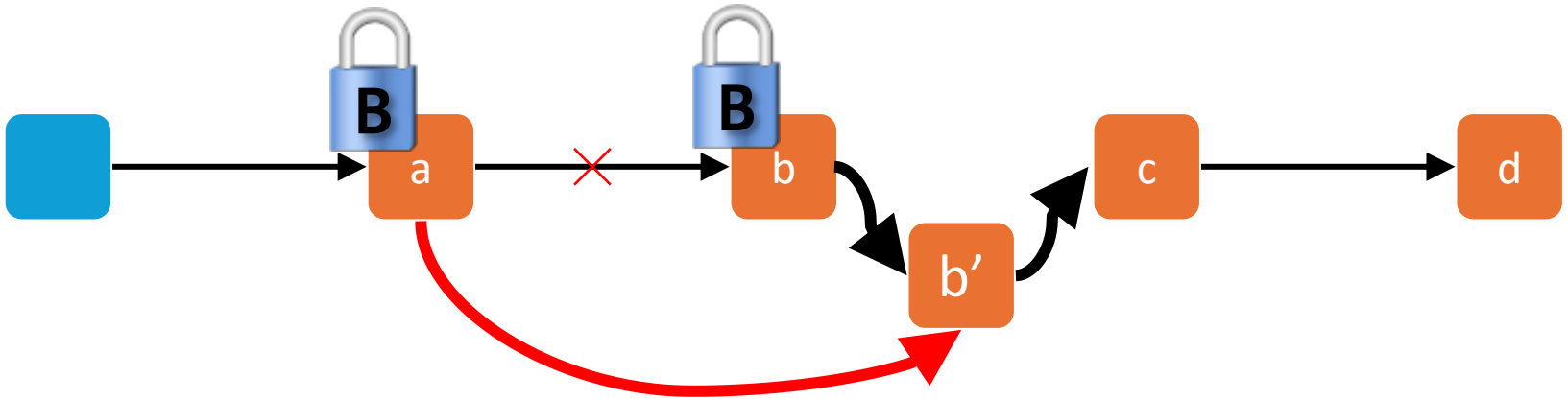


- b’ not added!

- To fix this lock the **successor** defensively as in the remove case

What about add?

```
A: add(b’)
B: remove(b)
```

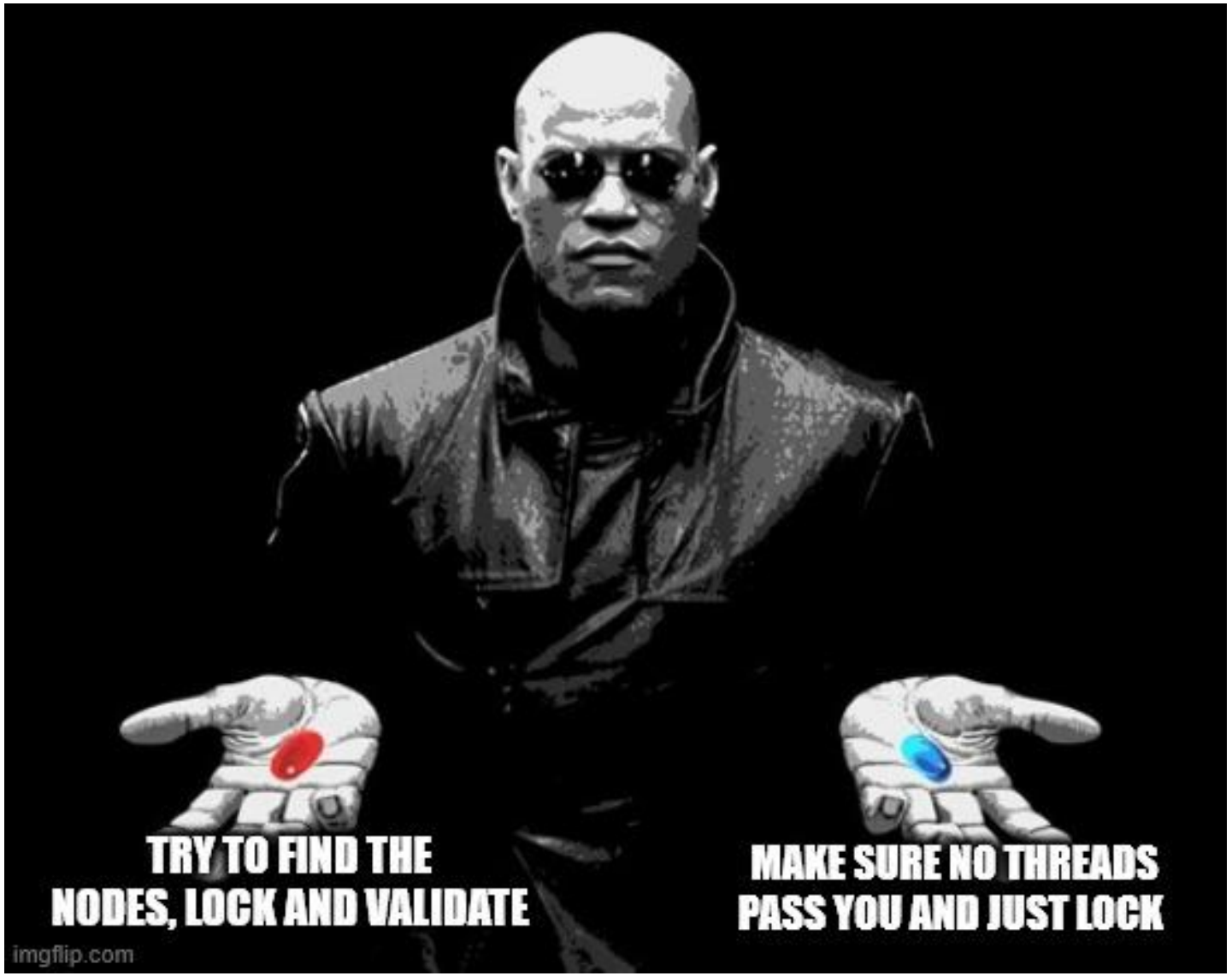


- b’ not added!

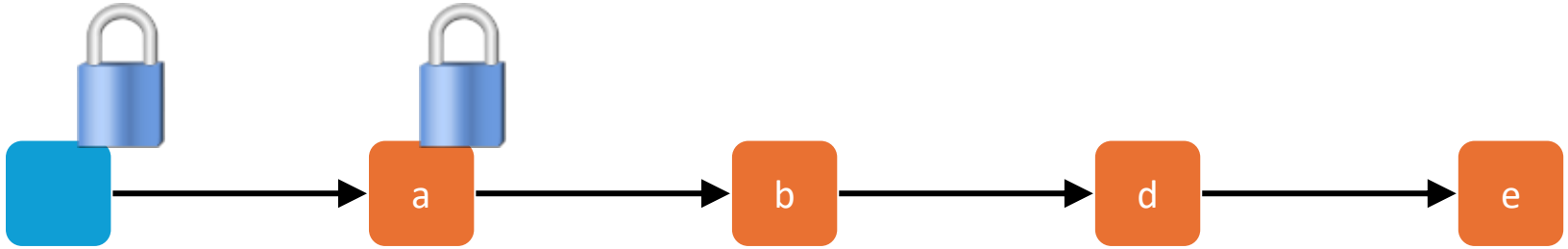
- Solution: lock the **successor** defensively.

The choice

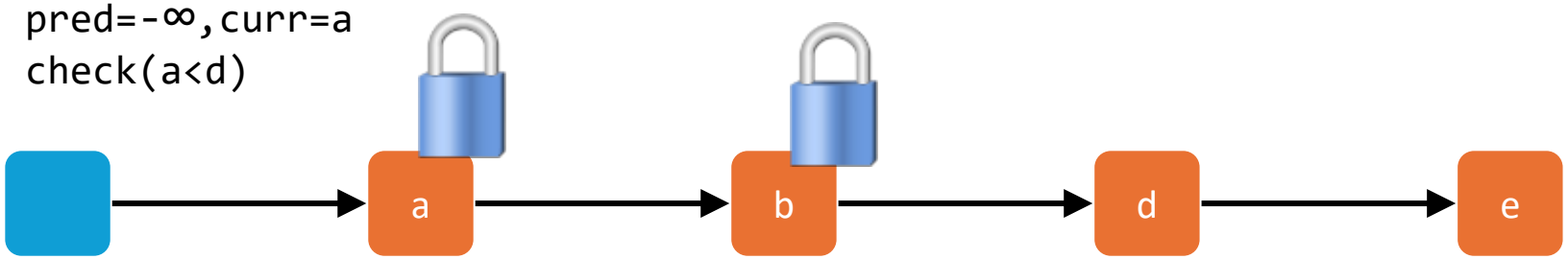
- How do we get to the data we need to work on?



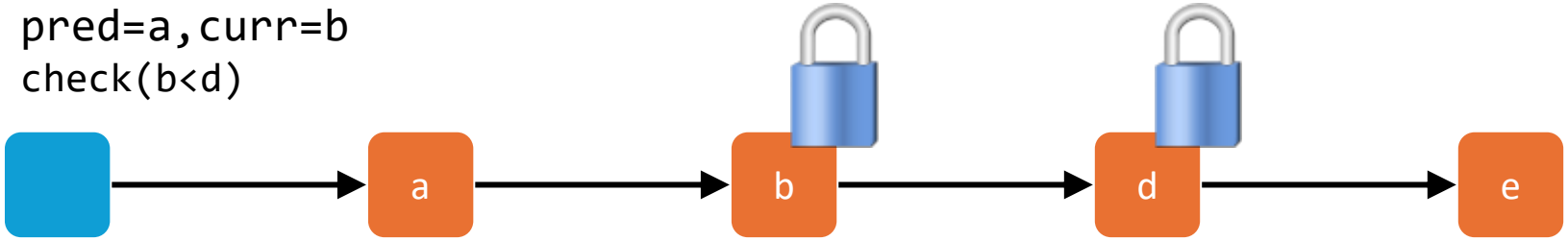
Hand-over-hand locking (remove d)



pred=-∞,curr=a
check(a<d)

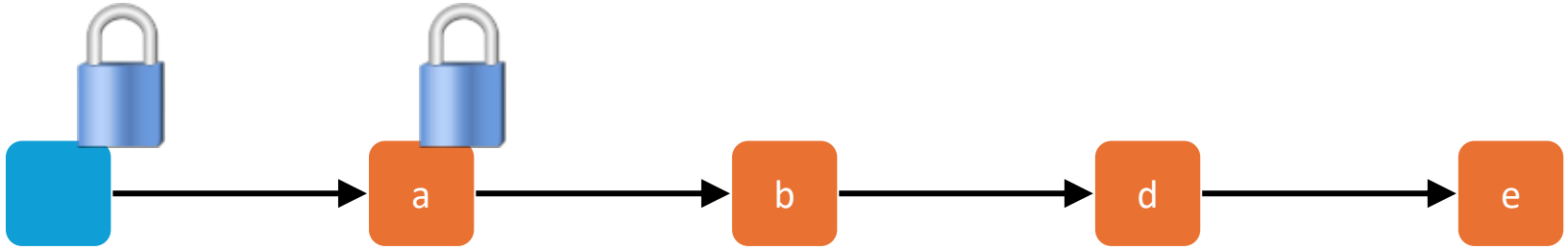


pred=a,curr=b
check(b<d)

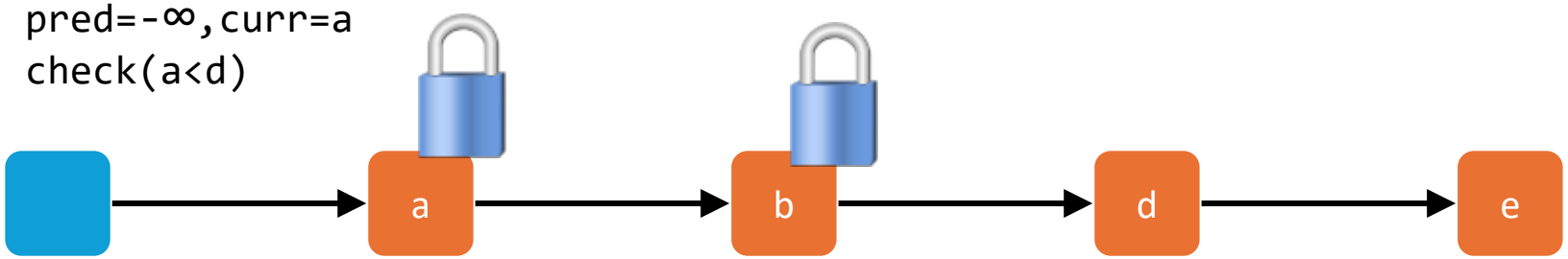


pred=b,curr=d
check(d<d)
if(d==d)
  remove(d)

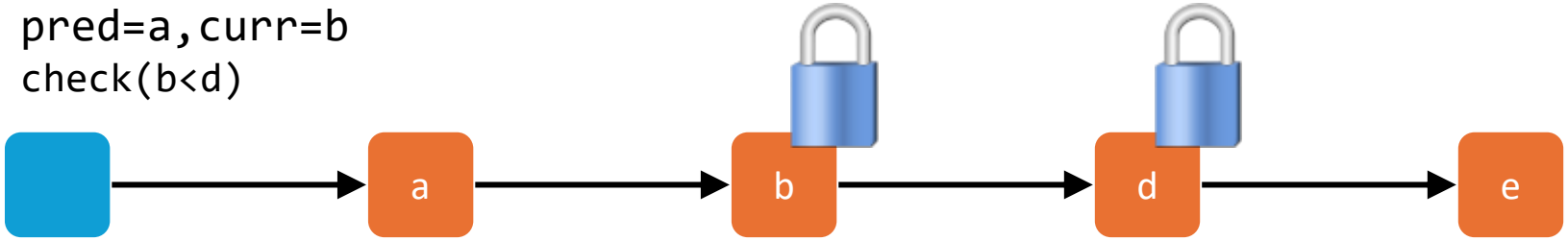
Hand-over-hand locking (remove d)



pred=-∞,curr=a
check(a<d)



pred=a,curr=b
check(b<d)



pred=b,curr=d
check(d<d)
if(d==d)
  remove(d)

What about add(c) and contains(e)?

Hand-over-hand locking

Benefits:

- Multiple readers and writers can be actively doing work in the same list.
- Readers and writers that are traversing the list in the same order will not pass each other.
- The locks taken on parts of the list won't deadlock with each other, because multiple locks are acquired **in the same order**.

Hand-over-hand locking

But what's bad?

- We can have "traffic jam", Threads can't overtake each other
- O(n) locks acquired/released => Big Overhead!

Optimistic Synchronization

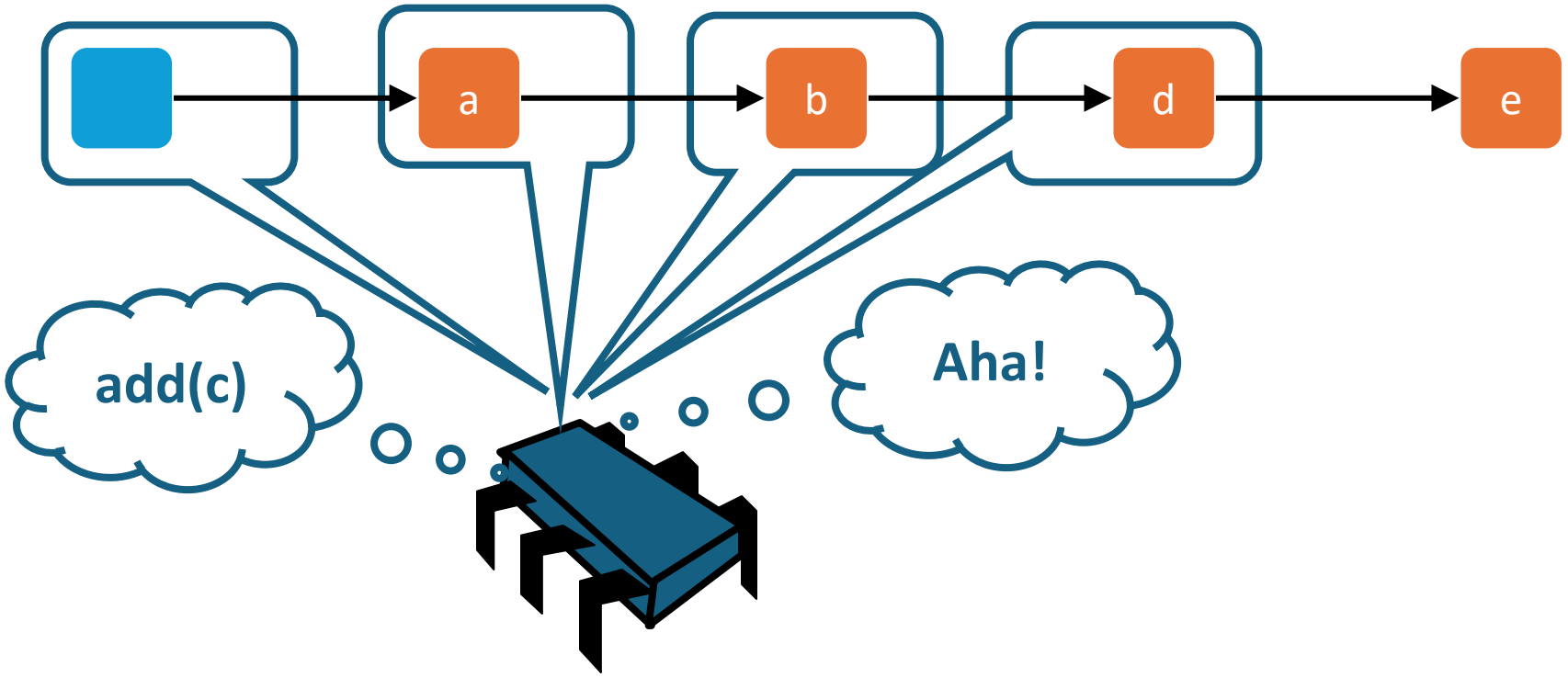
Idea

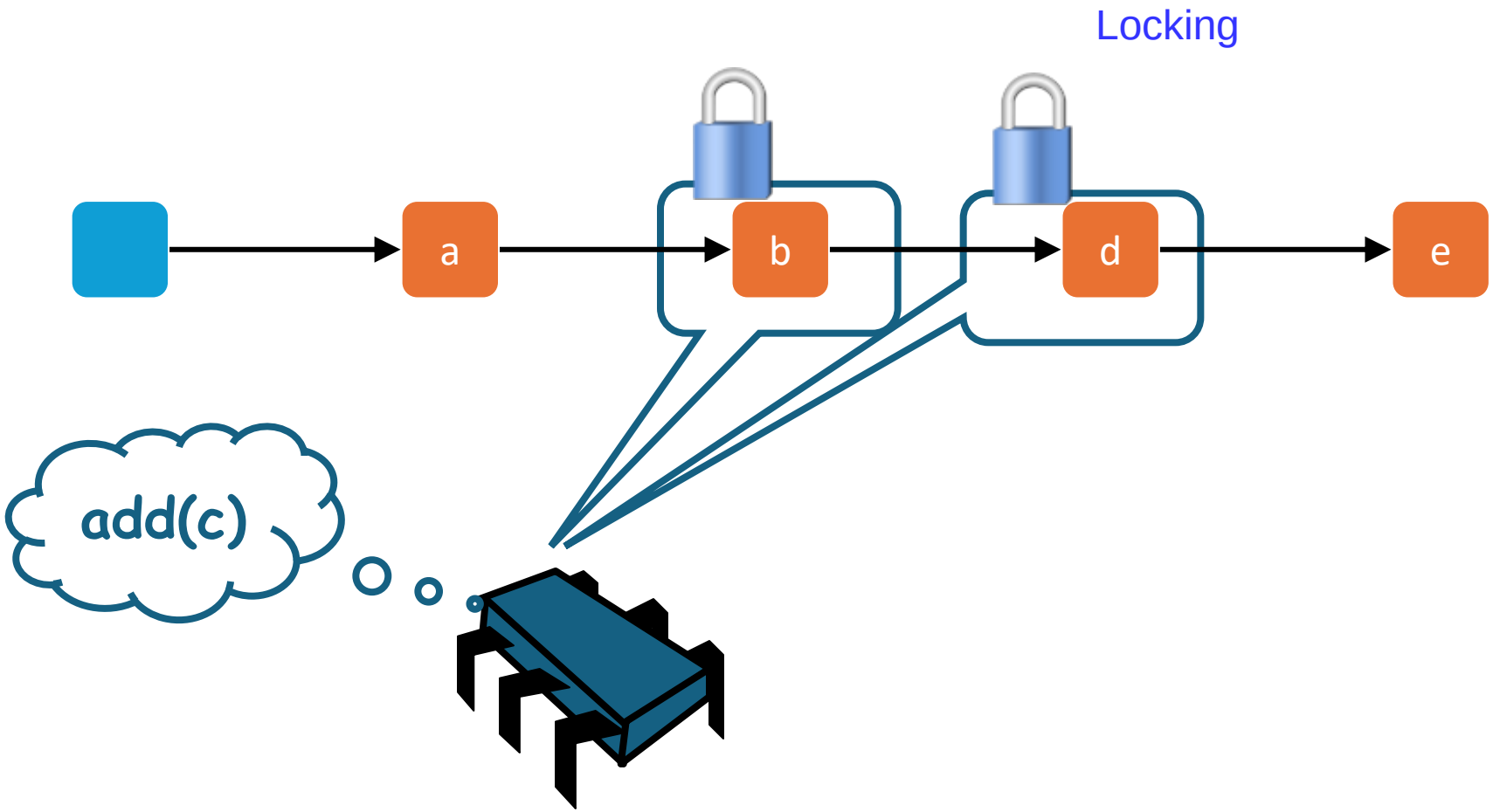
Algorithm:

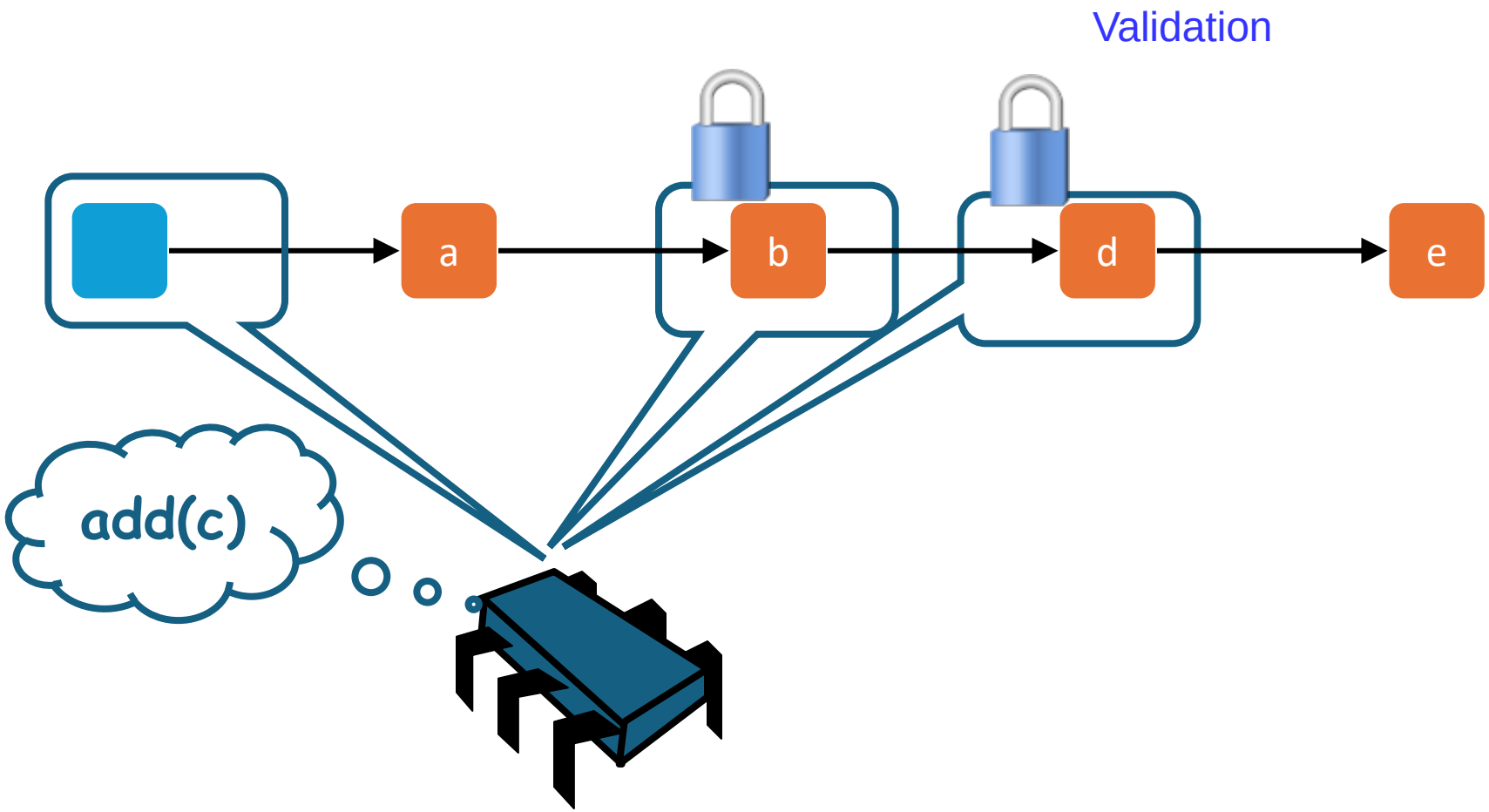
- find nodes without locking,
- then lock the two nodes and
- check that everything is ok (**validation**)
    - if so perform the operation (add, remove or contains) and return true
    - if not return false
- finally release the two locks

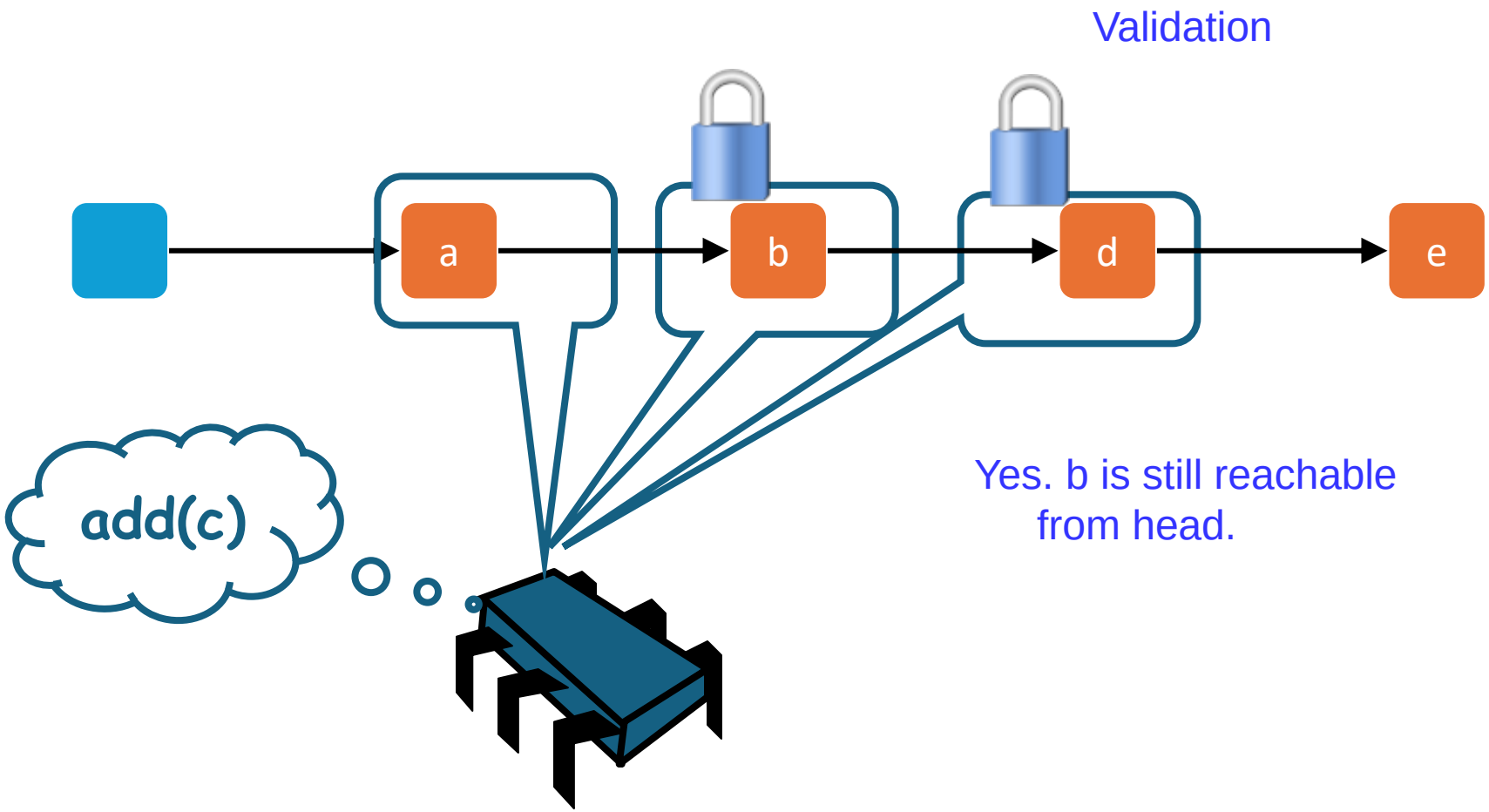
`e.g. add(c)`

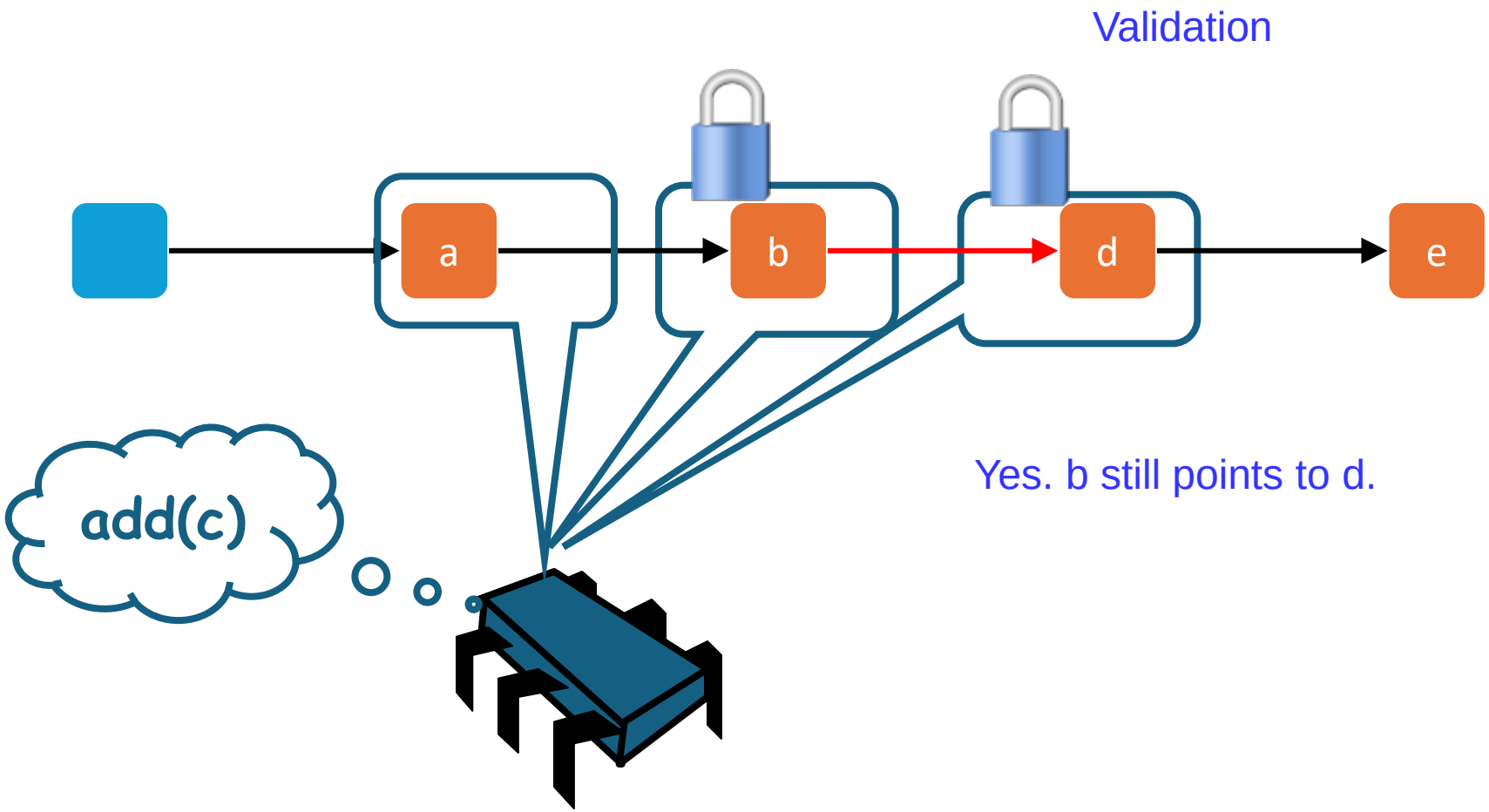
Finding without locking

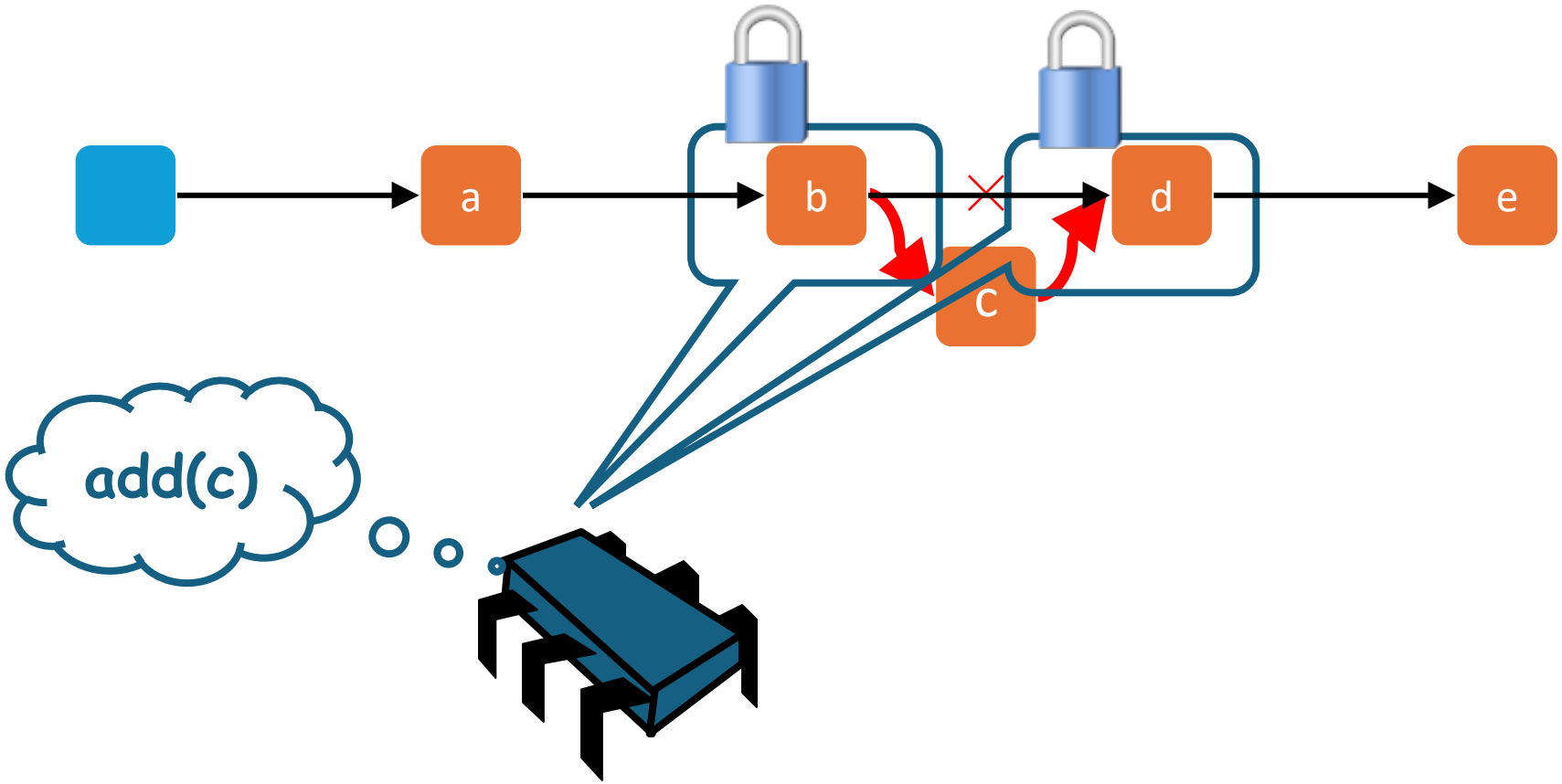


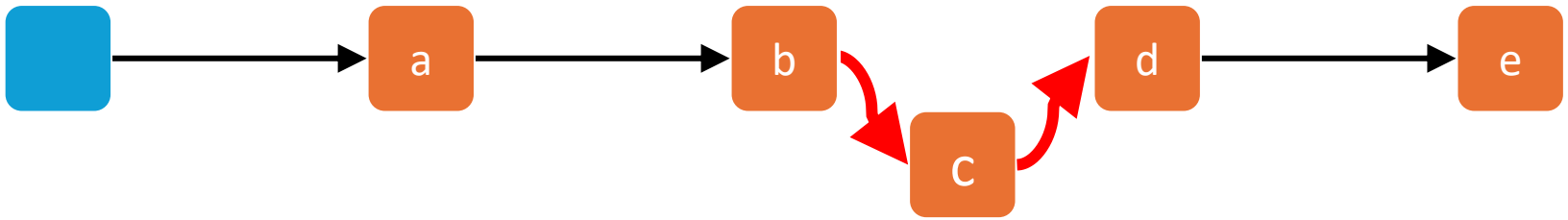

Locking
a
b
d
e
add(c)

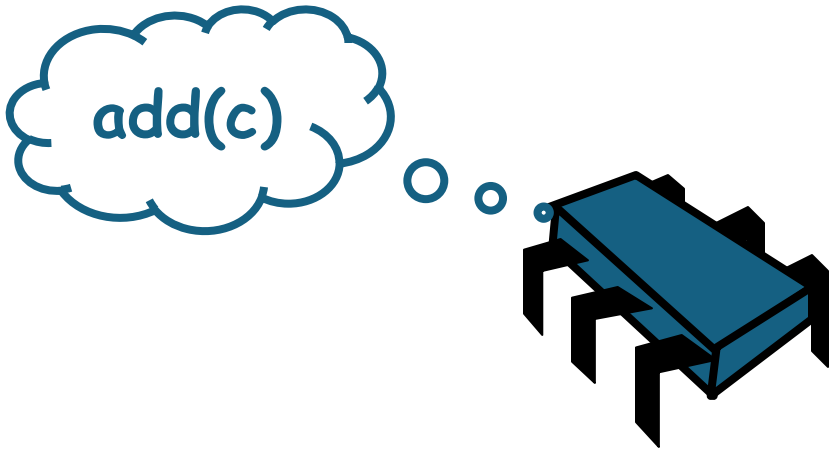

Validation
a
b
d
e
add(c)


Validation
a
b
d
e
add(c)
Yes. b is still reachable
from head.


Validation
a
b
d
e
add(c)
Yes. b still points to d.


a
b
c
d
e
add(c)


a
b
c
d
e

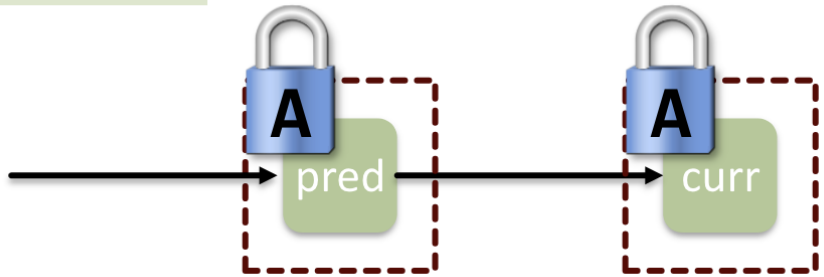

add(c)

Validation: what can go wrong?

Why do we even need validation?

Validate - summary

```
private Boolean validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) { // reachable?
       if (node == pred)
             return pred.next == curr; // connected?
       node = node.next;
   }
   return false;
}
```



Validation: what can go wrong?

Remove case

A: add(c)

A: find insertion point

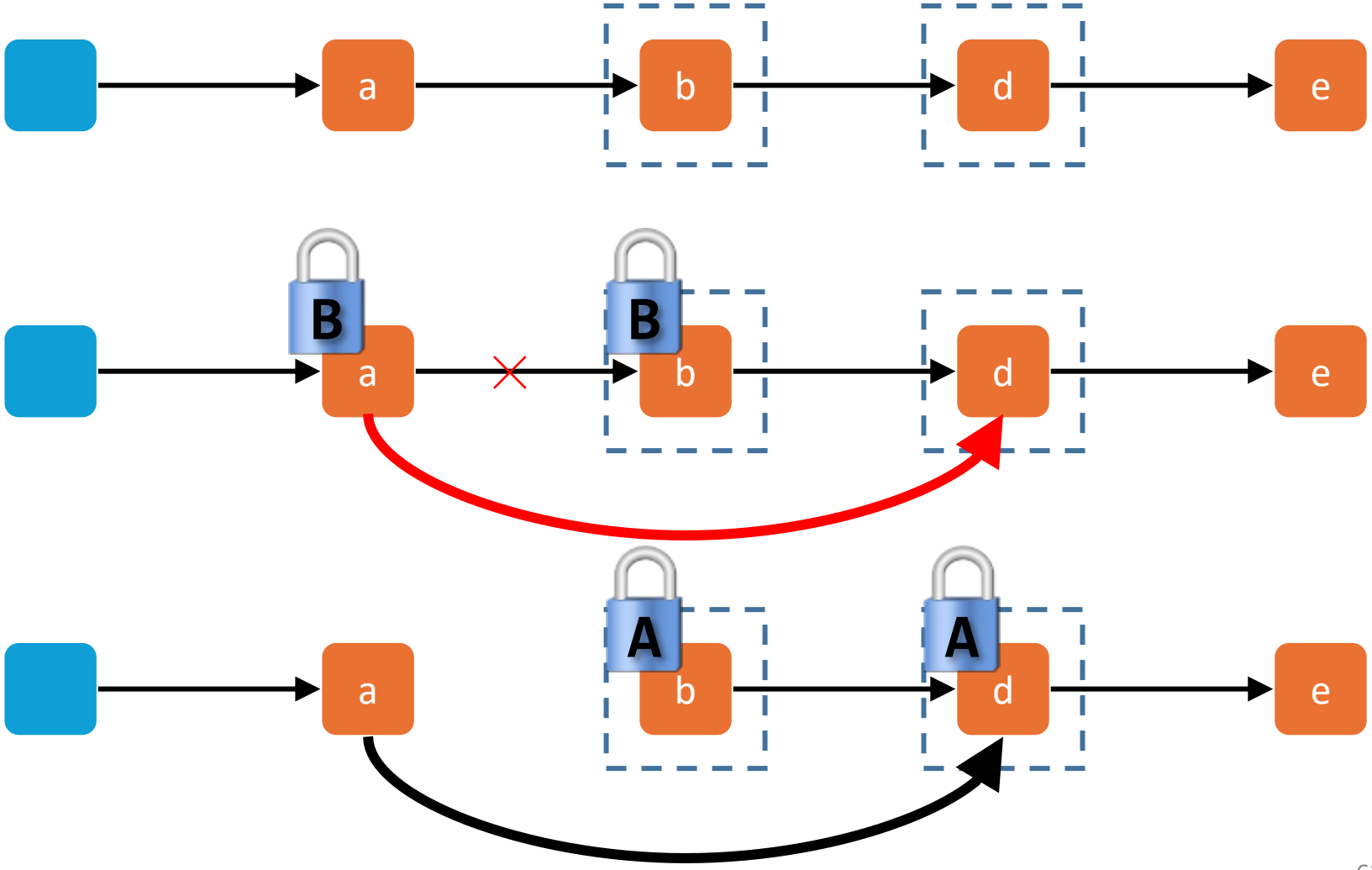
B: remove(b)

A: lock

A: validate: rescan

A: b not reachable

→return false

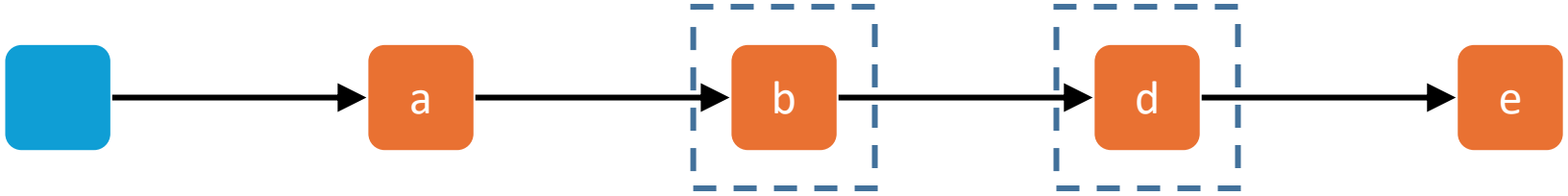


Validation: what can go wrong?

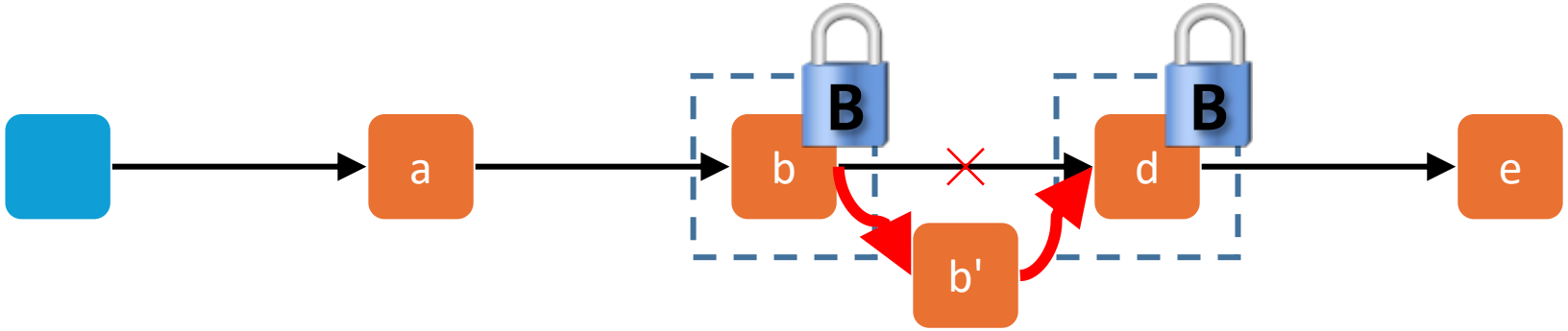
Insert case

A: add(c)

A: find insertion point



B: insert(b')

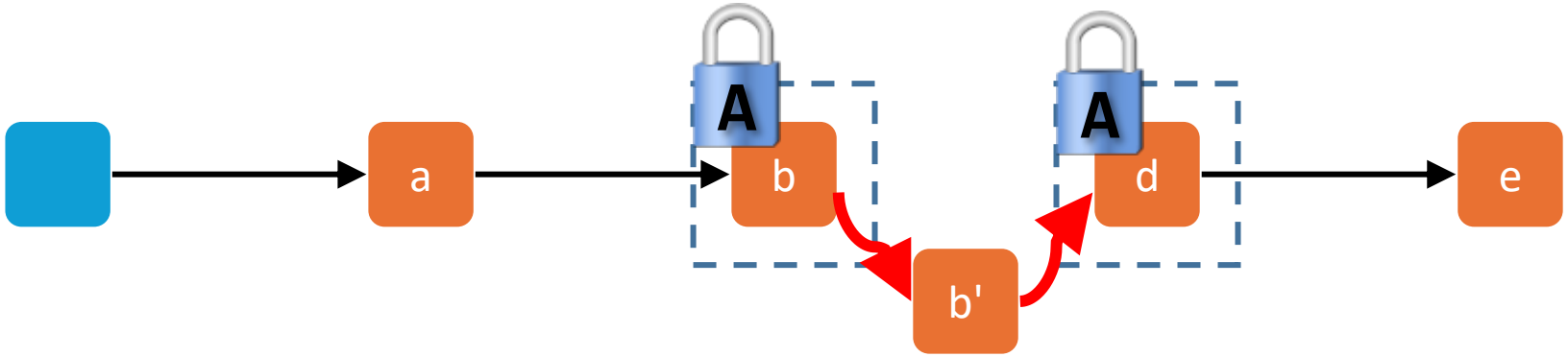


A: lock

A: validate: rescan

A: d != succ(b)

→return false



Optimistic Locking

```
@Override
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) { // present
                    return false;
                } else { // not present
                    Node entry = new Node(item);
                    entry.next = curr;
                    pred.next = entry;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

```
@Override
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

```
@Override
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        try {
            pred.lock();
            //curr.lock();
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally {
            pred.unlock();
            //curr.unlock();
        }
    }
}
```

```
private boolean validate(Node pred, Node curr) {
    Node entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred)
            return pred.next == curr;
        entry = entry.next;
    }
    return false;
}
```

Optimistic List

Good:

No contention on traversals.

Traversals are wait-free.

Less lock acquisitions.

Bad:

Need to traverse list twice (find + validate)

contains() method needs to acquire locks

Teaching Awards

- Ich wäre dankbar, wenn ihr für mich abstimmen könntet!

Lazy Synchronisation

Lazy List

Like optimistic list but

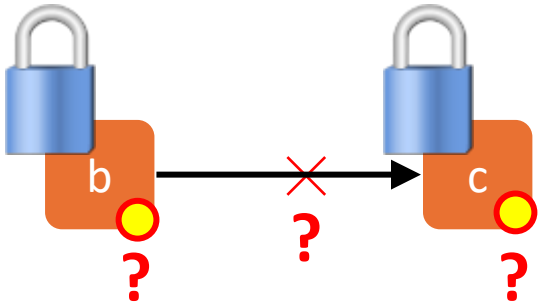
- scan only once
- contains() never locks

How?

- Removing nodes causes trouble
- do it "lazily”
- add a special **”removed?” flag** to the nodes

New Validate

- Given two locked nodes
- Pred is not marked
- Curr is not marked
- Pred points to Curr



Lazy List: Remove

Find nodes to remove (as before)

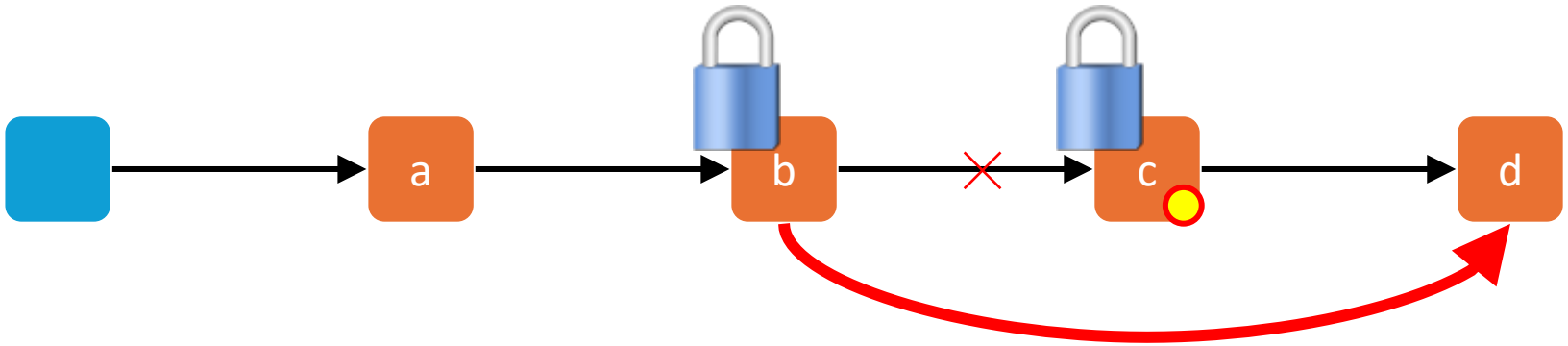
Lock predecessor and current (as before)

Validate (new validation)

Logical delete: mark current node as removed

Physical delete: redirect predecessor's next

e.g. remove(c)



Lazy List: Remove

Find nodes to remove (as before)

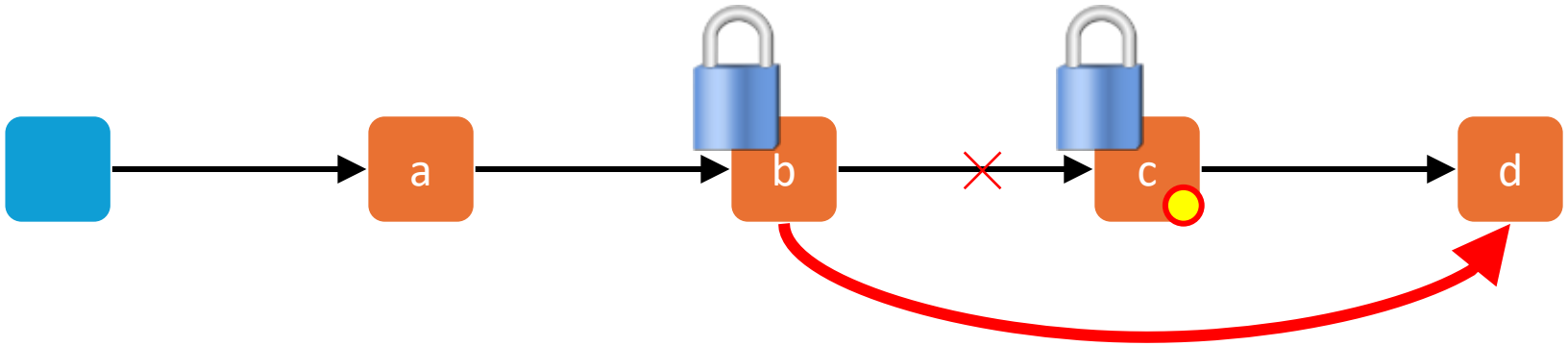
Lock predecessor and current (as before)

Validate (new validation)

Logical delete: mark current node as removed volatile?

Physical delete: redirect predecessor's next

e.g. remove(c)



Invariant

If a node is not marked then

- it is reachable from head
- and reachable from its predecessor

Only check if nodes are adjacent. Why?

A: remove(c)

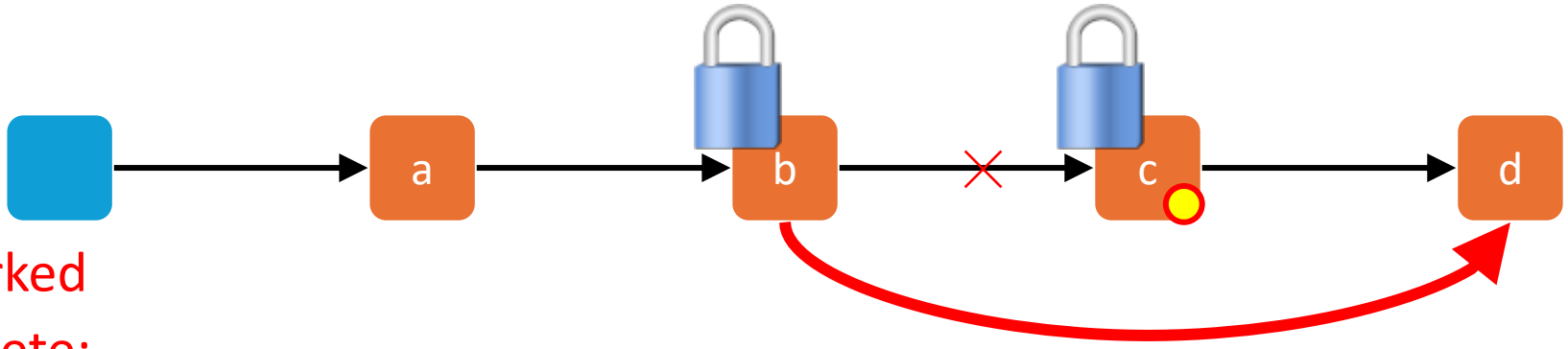
lock

check if b or c are marked

not marked? ok to delete:

mark c

delete c



Remove method

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) { // optimistic, retry
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        // remove or not
      } finally { curr.unlock(); }
    } finally { pred.unlock(); }
  }
}
```

What is validate() now?

Remove method

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) { // optimistic, retry
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        // remove or not
      } finally { curr.unlock(); }
    } finally { pred.unlock(); }
  }
}
```

```
if (!pred.marked && !curr.marked &&
      pred.next == curr) {
    if (curr.key != key)
       return false;
    else {
       curr.marked = true;    // logically remove
       pred.next = curr.next; // physically remove
       return true;
    }
}
```

Lazy List: Add

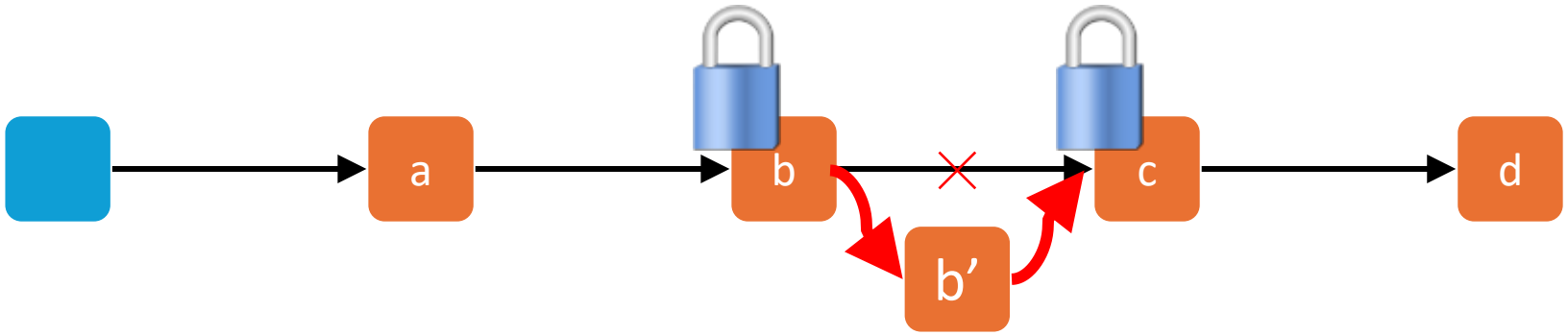
- Find nodes to where to add (as before)

Lock predecessor and current (as before)

Validate (new validation)

Physical add: change predecessor's next

e.g. add(b’)

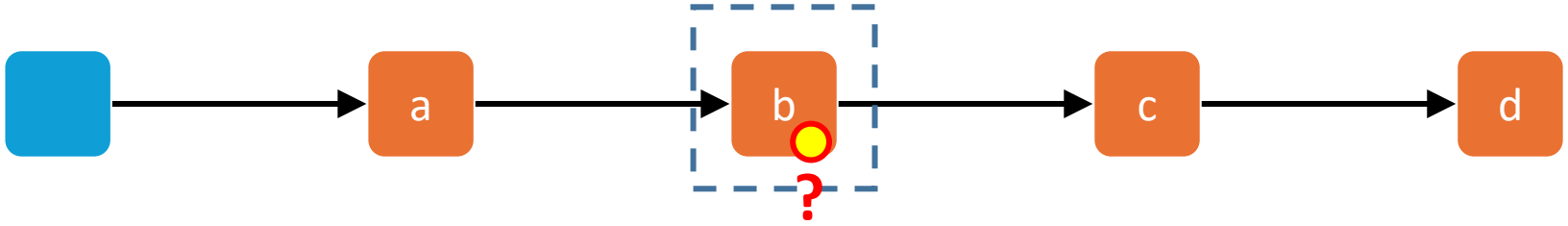


Lazy List: Contains

- Find nodes to return without locking

Return true if node is not marked

e.g. contains(b)



New Validation: What can go wrong?

A: add(c)

A: find insertion point

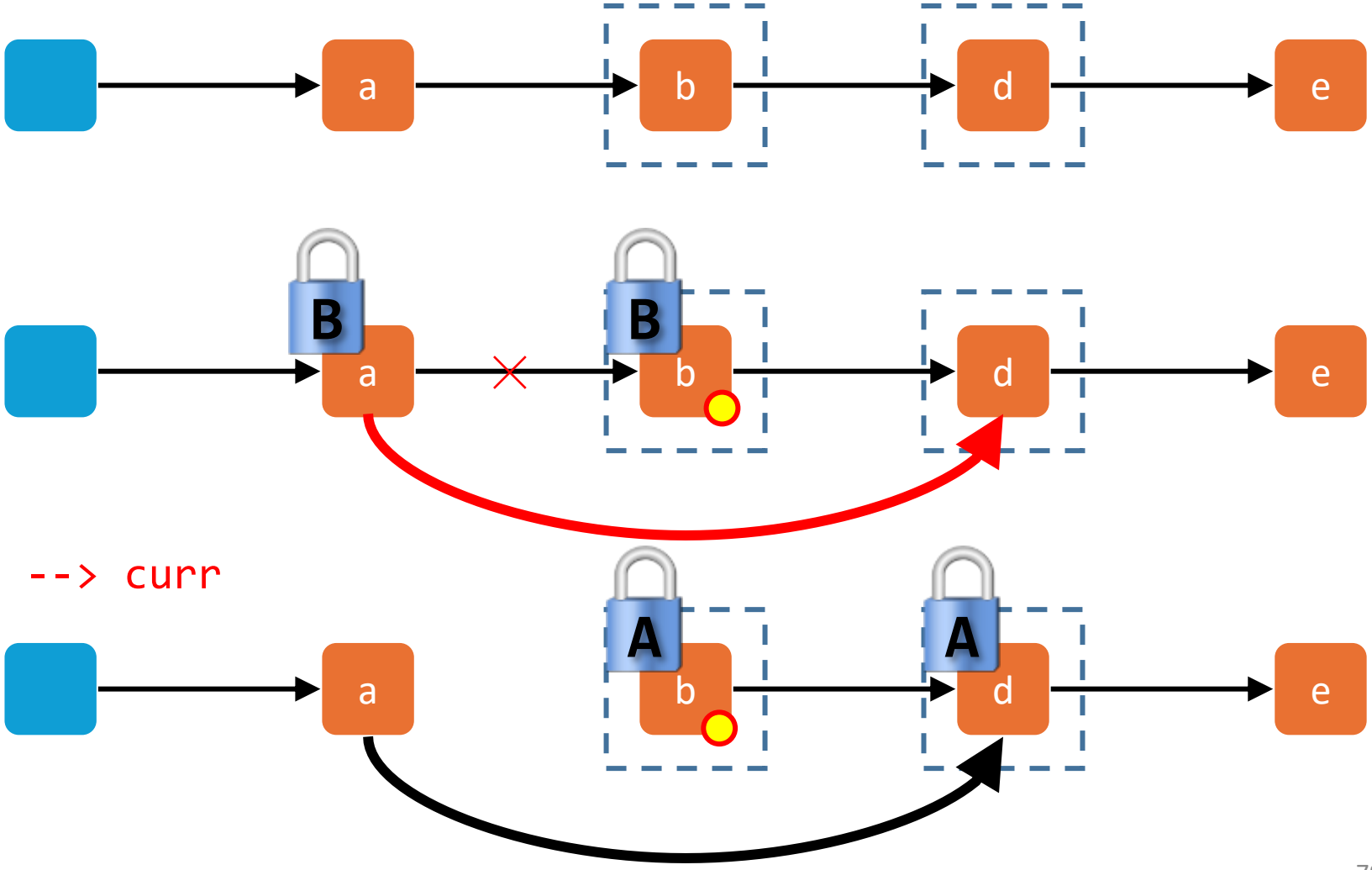
B: remove(b)

A: lock

A: validate: marks + pred --> curr

A: b marked

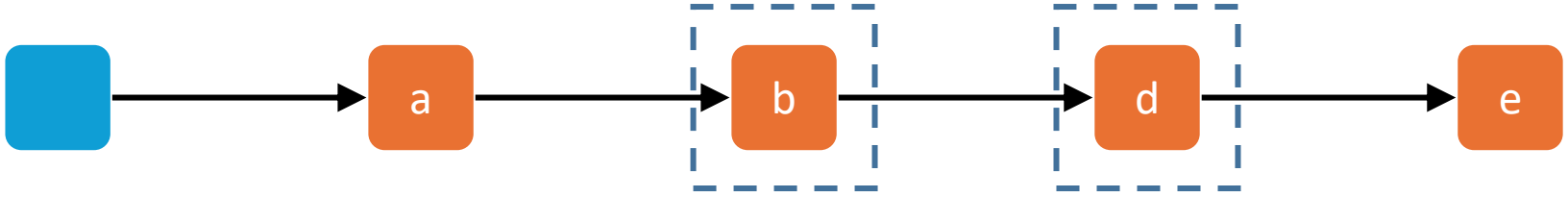
→return false



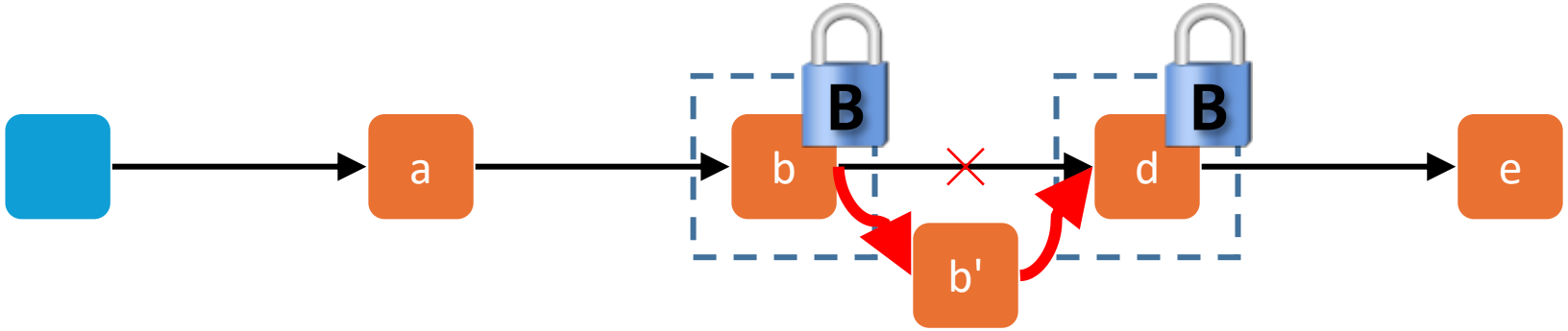
New Validation: What can go wrong?

A: add(c)

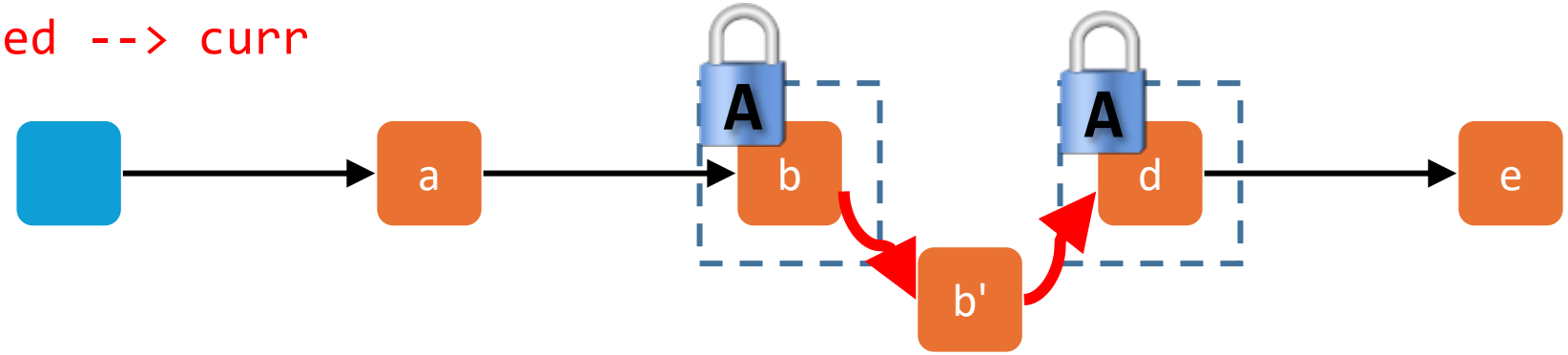
A: find insertion point



B: insert(b')

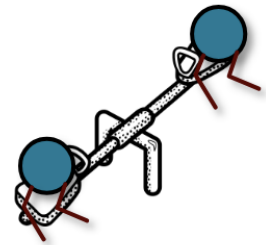


- A: lock
- A: validate: marks + pred --> curr
- A: pred –x-> curr
- →return false

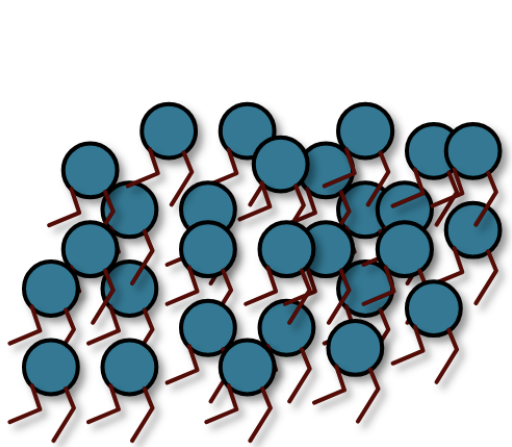


Lock free data structures

Locks performance



- **Uncontended case**
  - when threads do not compete for the lock
  - lock implementations try to have minimal overhead
  - typically "just" the cost of an atomic operation



- **Contended case**
  - when threads do compete for the lock
  - can lead to significant performance degradation
  - also, starvation
  - there exist lock implementations that try to address these issues

Disadvantages of locking

Locks are pessimistic by design

- Assume the worst and enforce mutual exclusion

Performance issues

- Overhead for each lock taken even in uncontended case
- Contended case leads to significant performance degradation
- Amdahl's law!

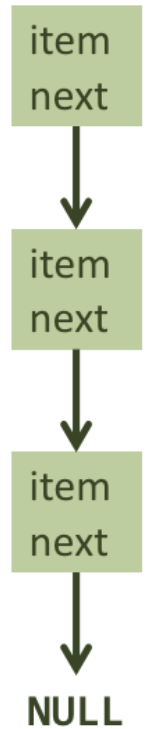
Blocking semantics (wait until acquire lock)

- If a thread is delayed (e.g., scheduler) when in a critical section → all threads suffer
- What if a thread dies in the critical section
- Prone to deadlocks (and also livelocks)
- Without precautions, locks cannot be used in interrupt handlers

So how do we build lock free data structures?

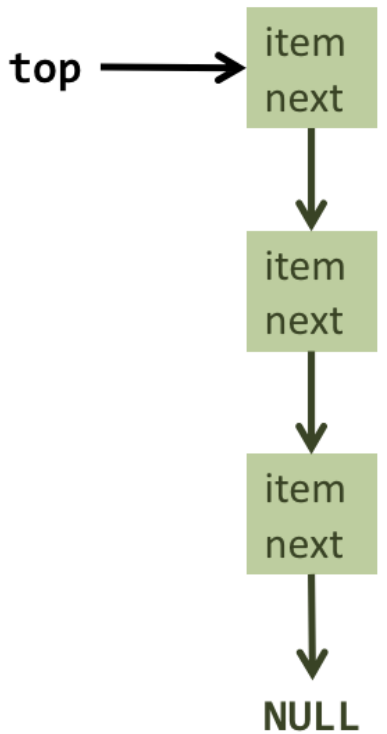
Stack Node

```
public static class Node {
   public final Long item;
   public Node next;

   public Node(Long item) {
       this.item = item;
   }

   public Node(Long item, Node n) {
       this.item = item;
       next = n;
   }

}
```



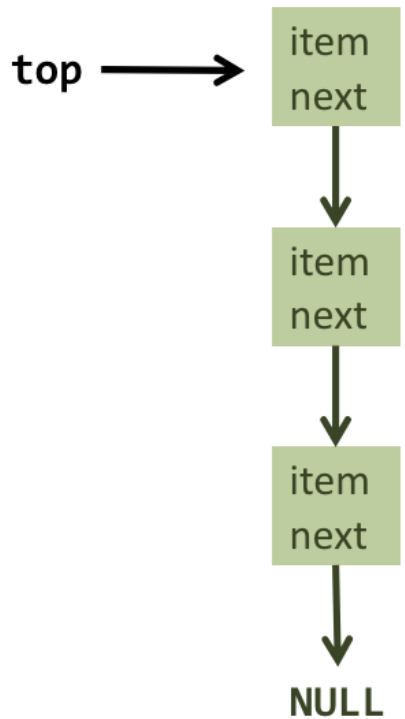
Blocking Stack

```
public class BlockingStack {
   Node top = null;

   synchronized public void push(Long item) {
      top = new Node(item, top);
   }

   synchronized public Long pop() {
      if (top == null)
         return null;
      Long item = top.item;
      top = top.next;
      return item;
   }
}
```



Non-blocking Stack

```
public class ConcurrentStack {
   AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```

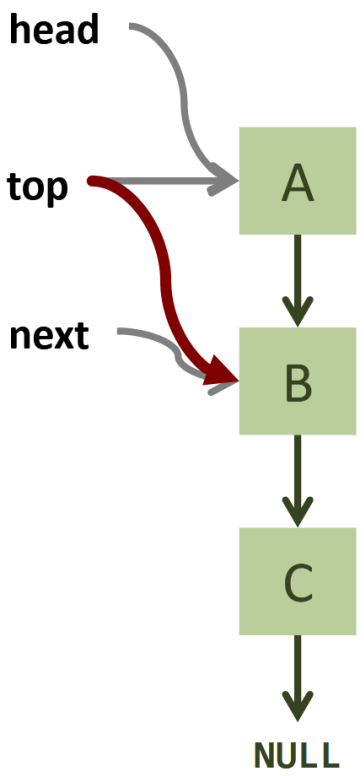


pop

```
public Long pop() {
  Node head, next;

  do {
    head = top.get();
    if (head == null) return null;
    next = head.next;
  } while (!top.compareAndSet(head, next));

  return head.item;
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change

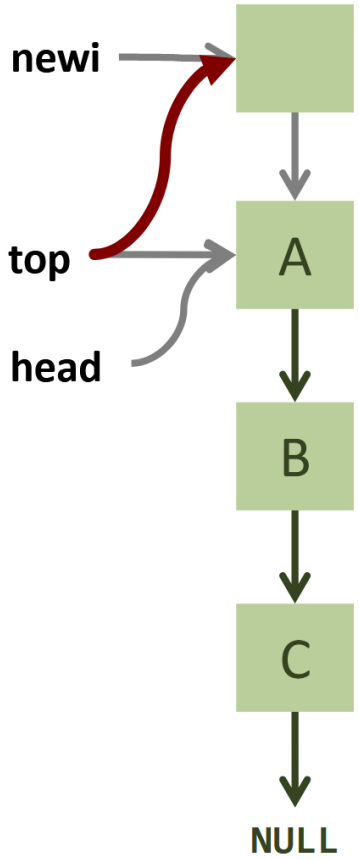


push

```
public void push(Long item) {
      Node newi = new Node(item);
      Node head;

      do {
            head = top.get();
            newi.next = head;
      } while (!top.compareAndSet(head, newi));
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change



What's the benefit?

Lock-free programs are deadlock-free by design.

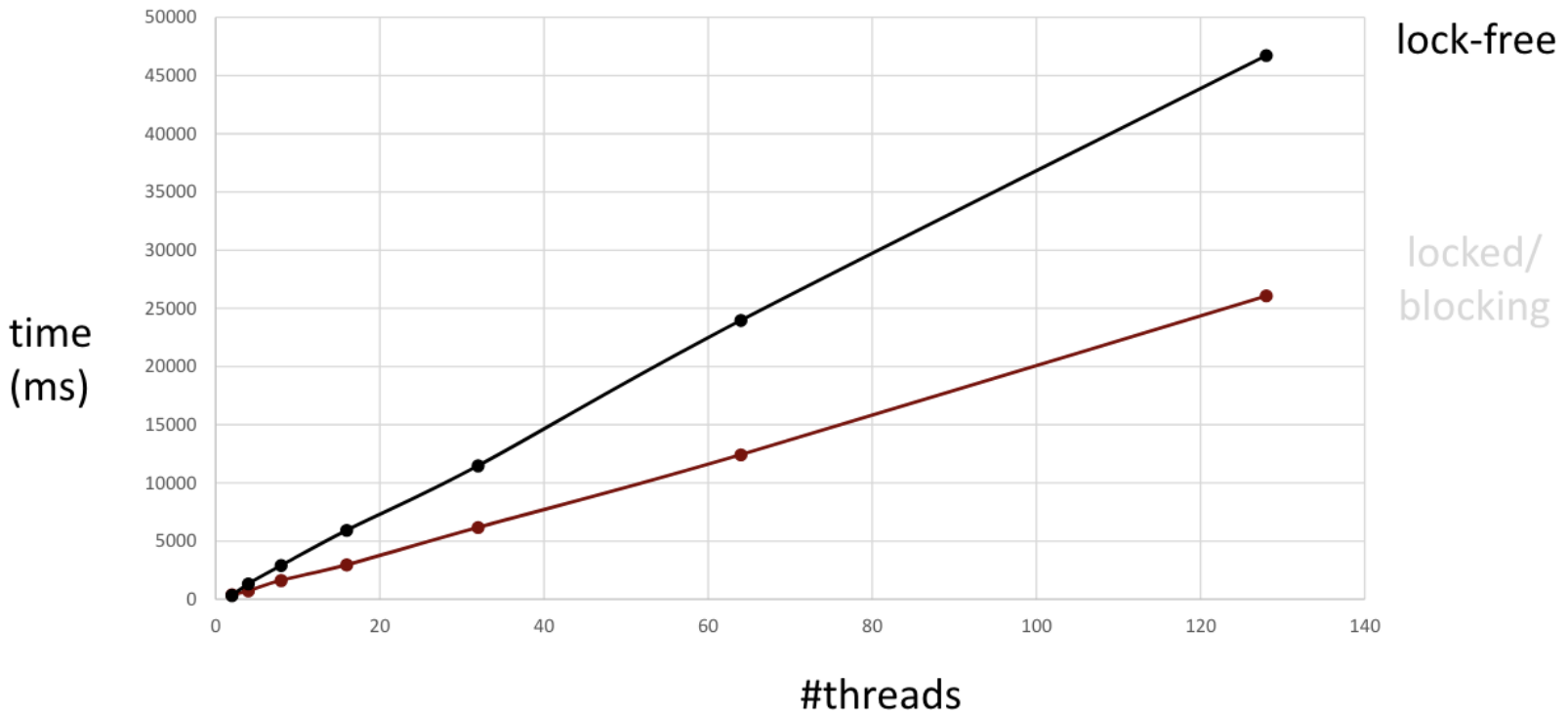
How about performance?

```
public void push(Long item) {
        Node newi = new Node(item);
        Node head;
        do {
                head = top.get();
                newi.next = head;
        } while (!top.compareAndSet(head, newi));
}
```

n threads

100,000 push/pop operations

10 times

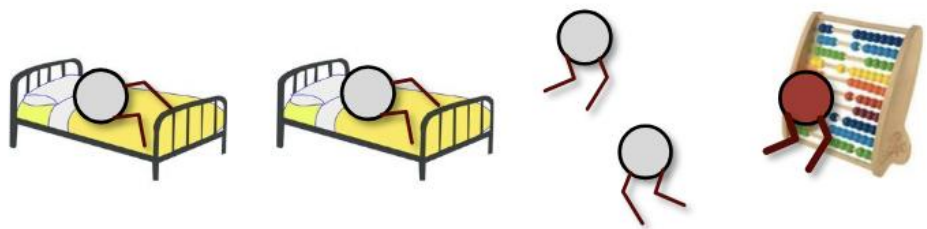


Performance

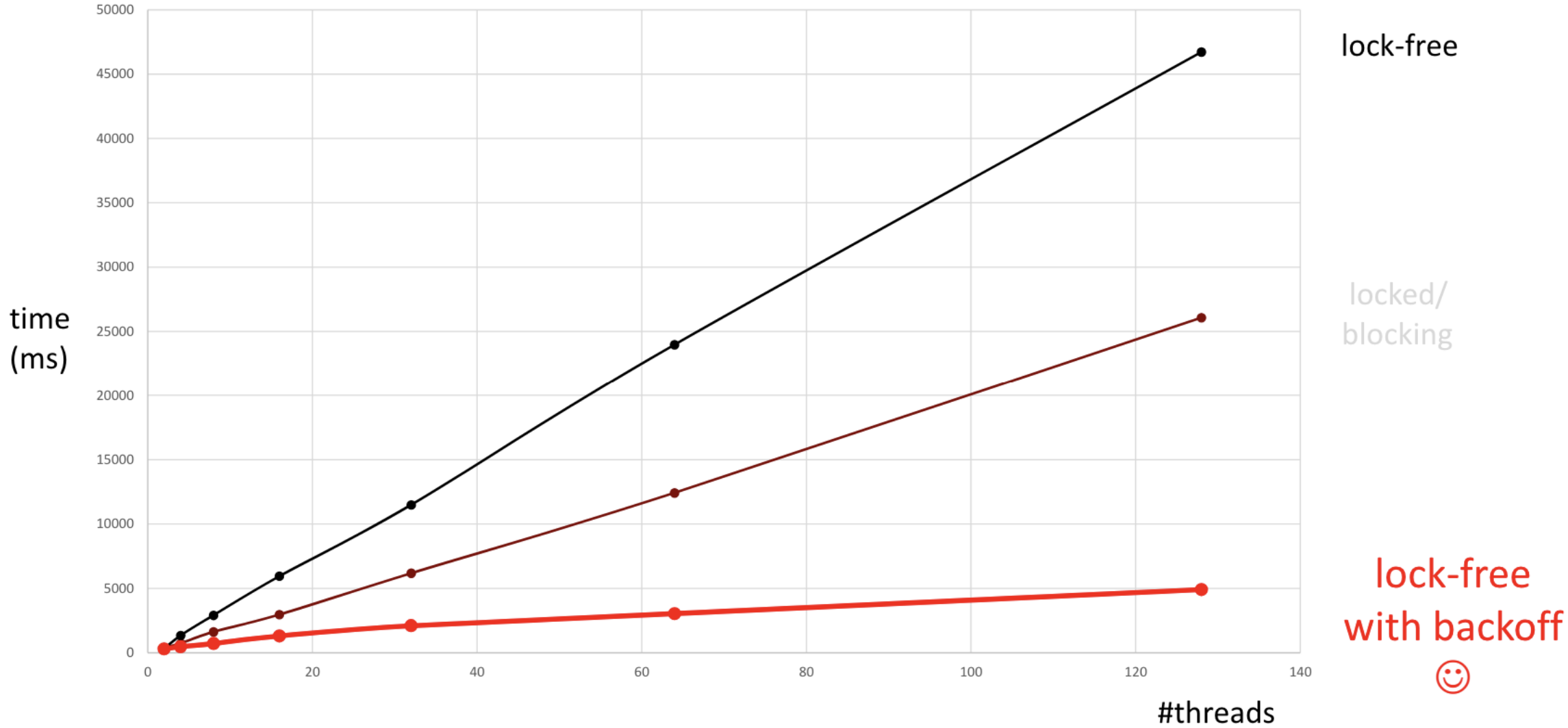
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

Atomic operations are expensive and contention can still be a problem.

→ Backoff, again.



With backoff



Problems with this implementation?

- Say we want to use a node pool instead of always creating new nodes (i.e. not always use new Node() but instead take it out of a list)
- -> ABA Problem (exam relevant)

Plan für heute

- Organisation
- Nachbesprechung Assignment 10
- Theory
- Intro Assignment 11
- **Kahoot**
- Exam questions

Kahoot!

Plan für heute

- Organisation
- Nachbesprechung Assignment 10
- Theory
- Intro Assignment 11
- Kahoot
- **Exam questions**

Types of exercises that might come in the exam

Disclaimer: This list is not guaranteed to be complete and is only meant to give you an idea of what has been asked on previous exams.

Locks

- Usually there are not too many question on this topic. true/false questions of which lock has which properties (fairness, starvation free)
- find bug in lock code (violation of mutual exclusion or deadlock freedom)
- draw state space diagram and/or read off correctness properties
- reproduce Peterson/Filter/Bakery lock
- prove correctness of Peterson lock or similar (but not Filter or Bakery)

Monitors, semaphores, barriers

- semaphore implementation (mostly with monitors)
- (never seen rendezvous with semaphores in an exam)
- barrier implementation (mostly with monitors)
- (only seen a task on implementing a barrier with semaphores *once* in FS21, 8b)
- fill out some program using monitors (similar to wait/notify exercises, maybe with lock conditions)

Credits @aellison PProg23

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.
```
1   class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5           int cur_threads = i.incrementAndGet();
6           if(cur_threads < threads) {
7             while (i.get() < threads) {}
8           }
9       }
10  }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.

(4)

i.
```
1  class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5               int cur_threads = i.incrementAndGet();
6               if(cur_threads < threads) {
7                 while (i.get() < threads) {}
8               }
9       }
10 }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

True, there is no data race since `incrementAndGet` increases `i` atomically.

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5          int cur_threads = i.incrementAndGet();
6          if(cur_threads < threads) {
7            while (i.get() < threads) {}
8          }
9      }
10 }
```

◯ Der Code beendet sich immer. *Code will always complete.*

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.

(4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5              int cur_threads = i.incrementAndGet();
6              if(cur_threads < threads) {
7                while (i.get() < threads) {}
8              }
9      }
10 }
```

◯ Der Code beendet sich immer. *Code will always complete.*

True, it is a correct barrier implementation.

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.

(4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5          int cur_threads = i.incrementAndGet();
6          if(cur_threads < threads) {
7            while (i.get() < threads) {}
8          }
9      }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.

(4)

i.
```
1  class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5               int cur_threads = i.incrementAndGet();
6               if(cur_threads < threads) {
7                 while (i.get() < threads) {}
8               }
9       }
10 }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

True, the waiting threads are busy waiting.

ii.

```
1  class Barrier {
2      int i = 0;
3      final int threads = N;
4      public synchronized void await() throws InterruptedException {
5          ++i;
6          while (i < threads) { wait(); }
7          notify();
8      }
9  }
```

◯ Der gezeigte Code hat die gewünschte Semantik.

Code has the desired semantics.

ii.

```
1  class Barrier {
2      int i = 0;
3      final int threads = N;
4      public synchronized void await() throws InterruptedException {
5          ++i;
6          while (i < threads) { wait(); }
7          notify();
8      }
9  }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

Yes

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code beendet sich immer. *Code will always complete.*

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code beendet sich immer. *Code will always complete.*

True

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

False, the code makes use of wait/notify and thus does not waste compute resources.

Feedback

- Falls ihr Feedback möchtet sagt mir bitte Bescheid!
- Schreibt mir eine Mail oder auf Discord

Teaching Awards

- Ich wäre dankbar, wenn ihr für mich abstimmen könntet!

Danke

- Bis nächste Woche!