

Parallel Programming

Exercise Session 11

Outline

- Post-Disc. Assignment 10
- Pre-Disc. Assignment 11
- Theory
- Kahoot

Feedback: Assignment 10

Assignment 10

Dining Philosophers (naive attempt):

```
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 philosopher's left side
is another's right side!

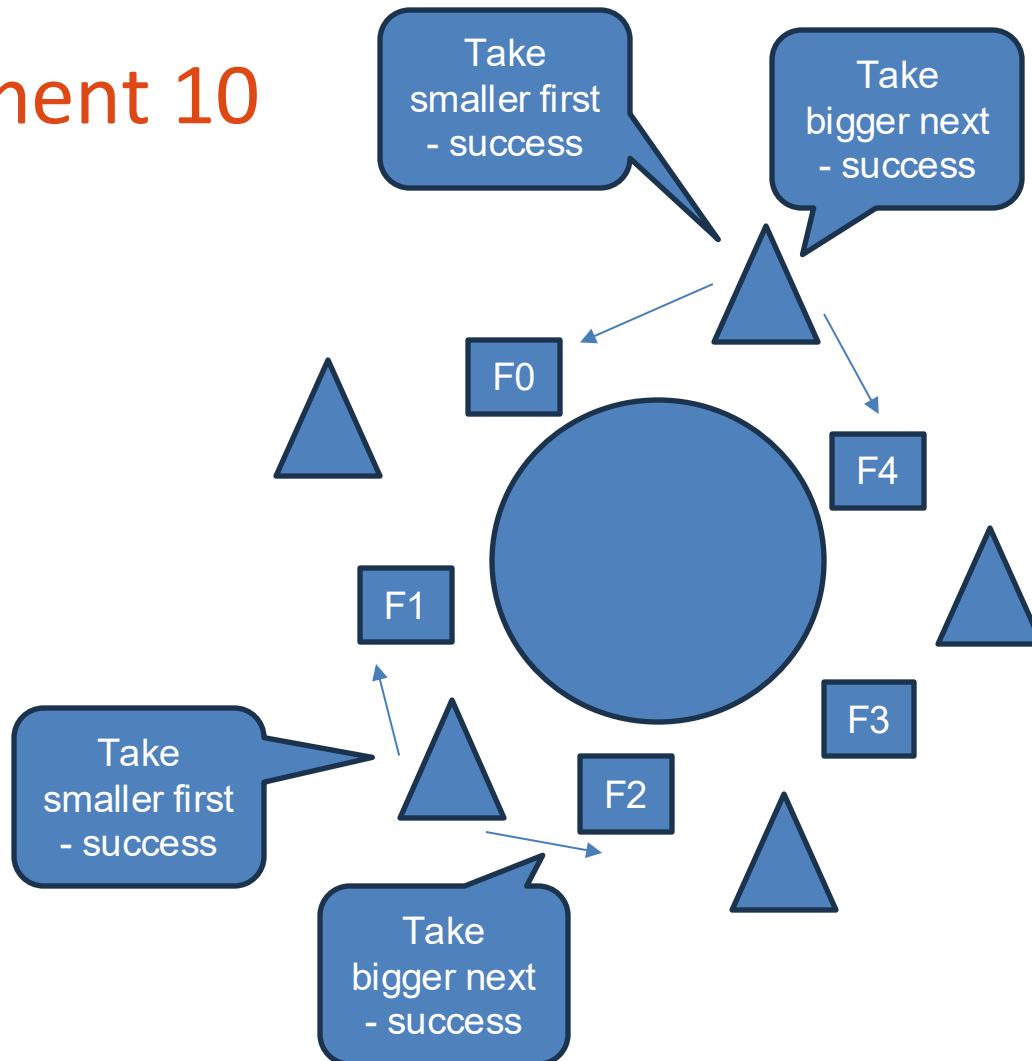
But we take left first, then
right. So we hold one fork,
then wait – leads to cycle
in dependency graph.

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!

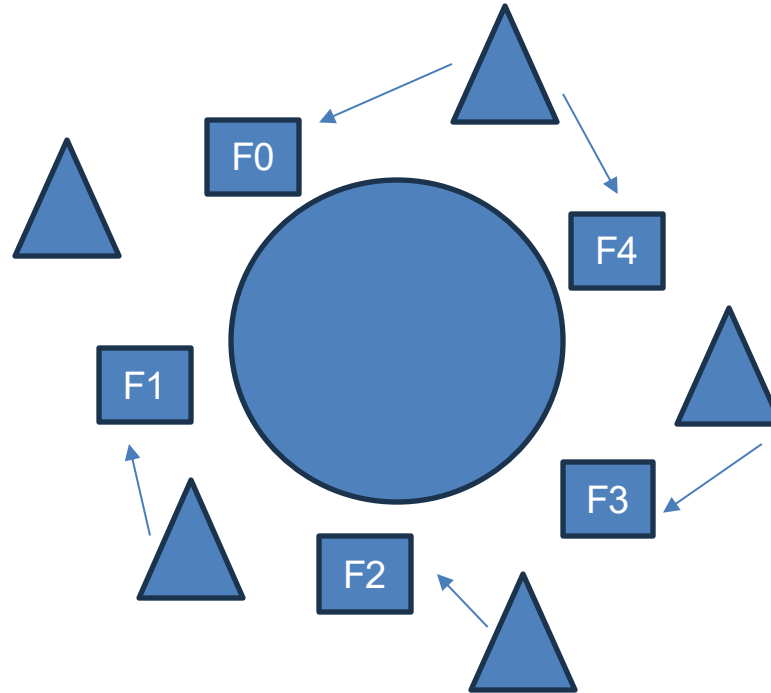
Assignment 10



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!)

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.

Assignment 10 – Semaphore implementation

```
public class MySemaphore {  
    private volatile int count;  
  
    public MySemaphore(int maxCount) {  
        this.count = maxCount;  
    }  
  
    public void acquire() throws InterruptedException {  
        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?

Can we also use notify here?

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();  
    }  
}
```

```
private int i = 0;  
private final int n;  
private boolean draining = false;
```

```
MyBarrier(int n) {  
    this.n = n;  
}
```

Why do we distinguish between draining and non-draining?

Assignment 11

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**.

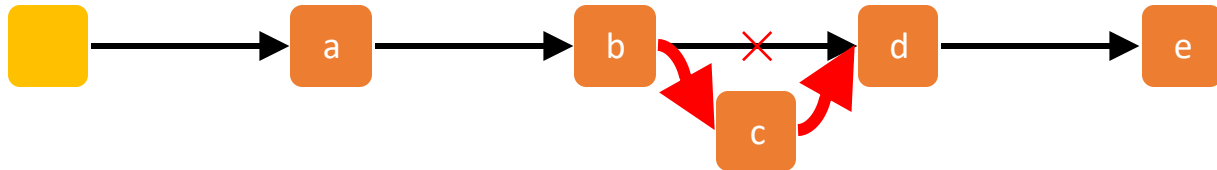
`add(c)`



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**.

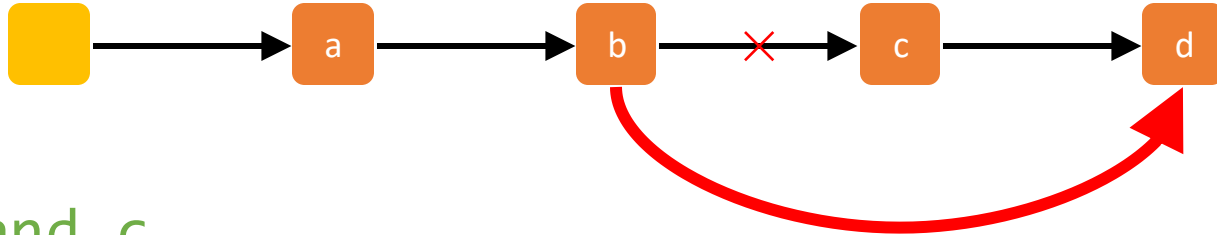
`remove(c)`



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 tips

- 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

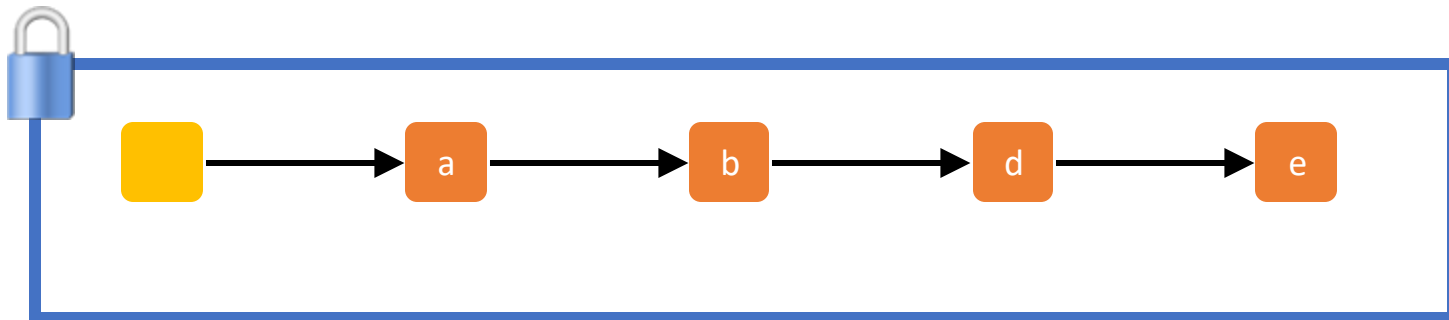
```
public synchronized boolean add(T x) {...};  
public synchronized boolean remove(T x) {...};  
public synchronized boolean contains(T x) {...};
```

add(c)



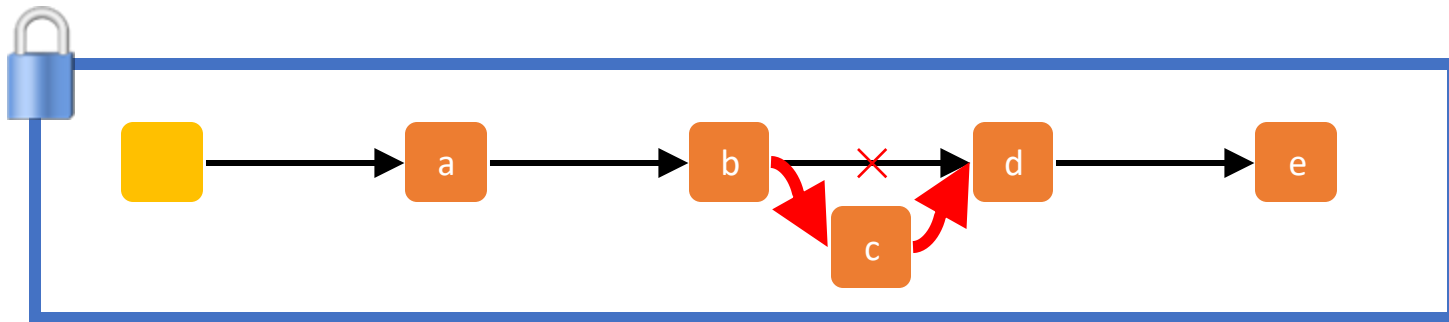
Coarse Grained Locking

```
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

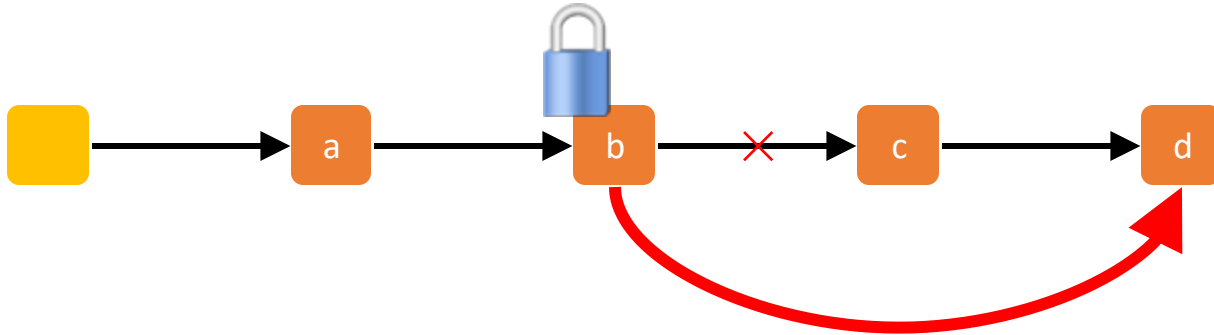
Let's try this

`remove(c)`



Let's try this

`remove(c)`

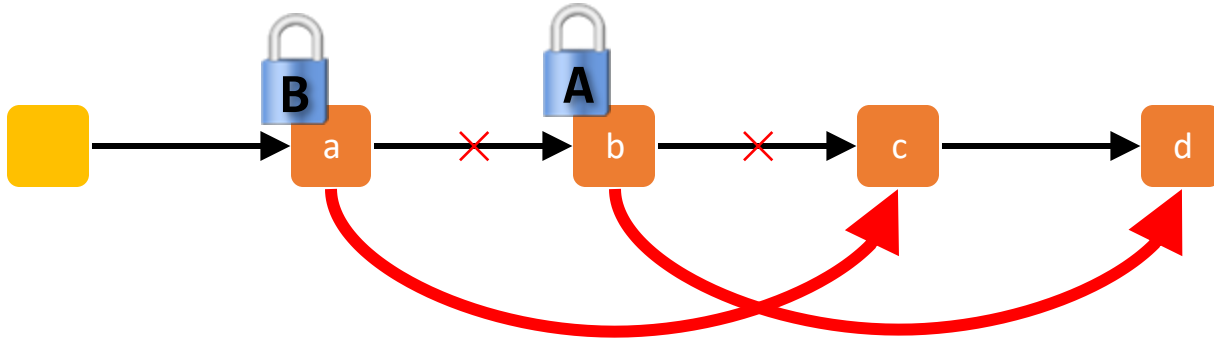


Locking the predecessor is ok?

Let's try this

A: `remove(c)`

B: `remove(b)`

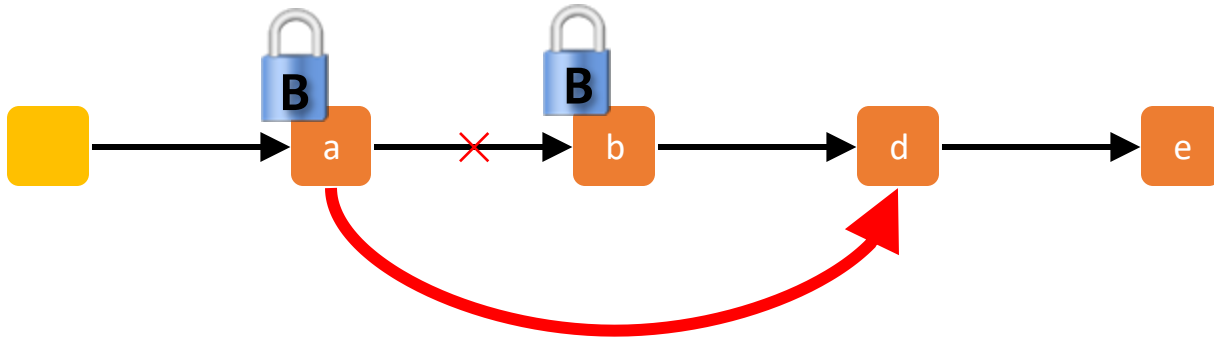


c not deleted!

What's the problem?

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

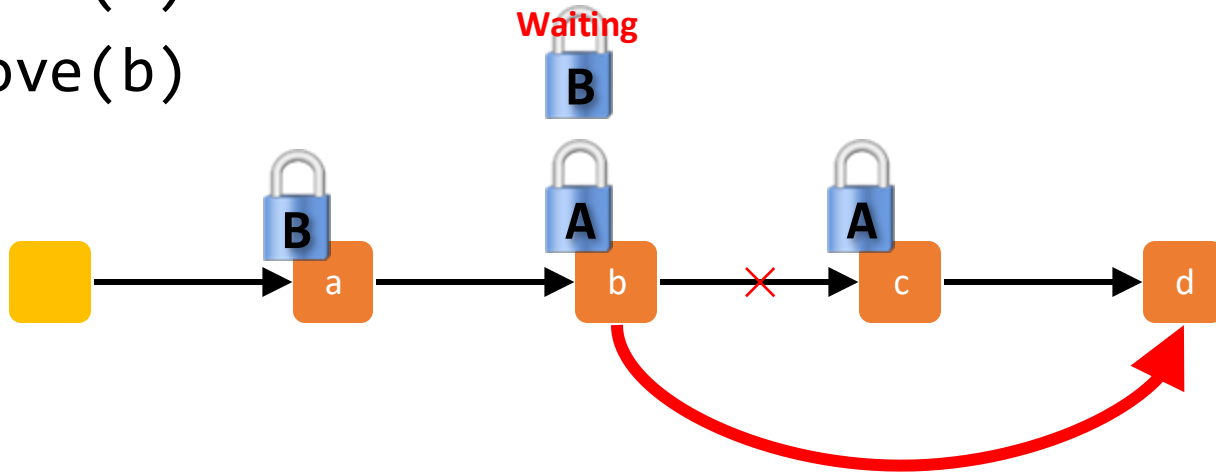
find a and b
`a.next=b.next`



Let's try this

A: remove(c)

B: remove(b)



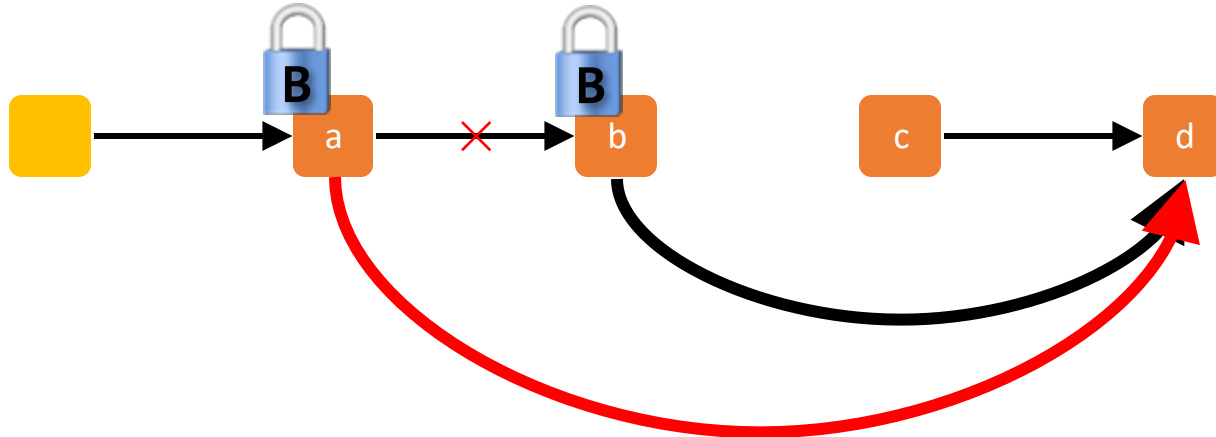
When removing, lock the **successor** defensively.

Problem solved: c not deleted!

Let's try this

A: `remove(c)`

B: `remove(b)`

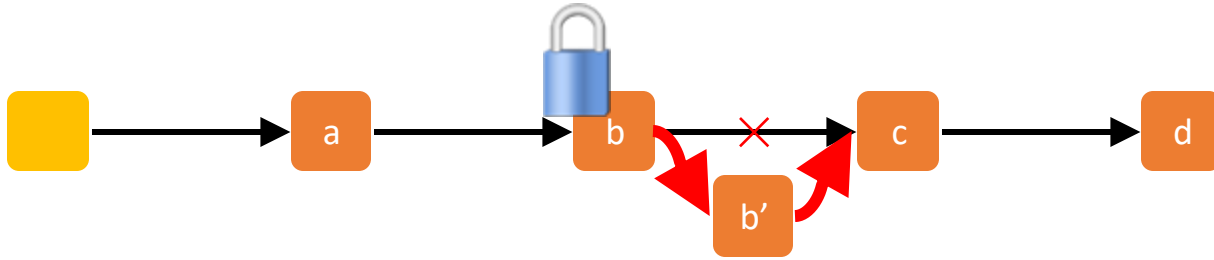


When removing, lock the **successor** defensively.

Problem solved: c not deleted!

What about add?

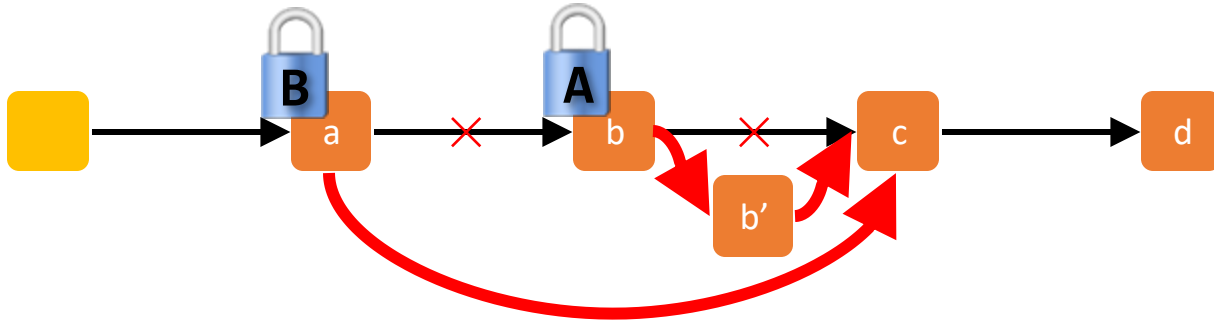
`add(b')`



What about add?

A: `add(b')`

B: `remove(b)`

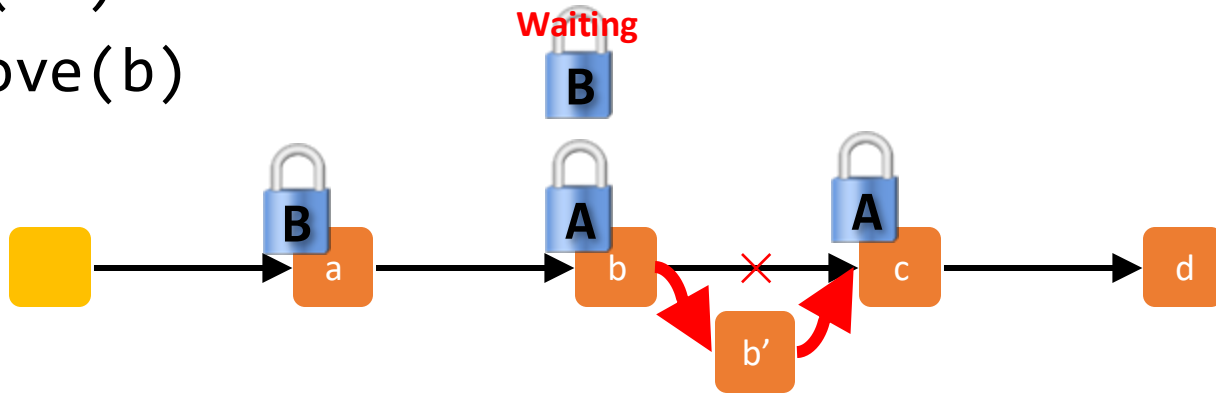


`b'` not added!

What about add?

A: `add(b')`

B: `remove(b)`



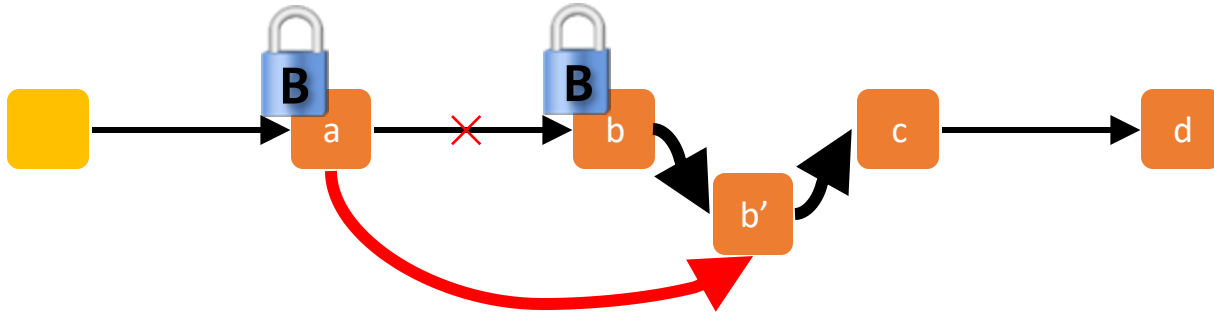
Also when adding lock the **successor** defensively.

Problem solved: `b'` not added!

What about add?

A: `add(b')`

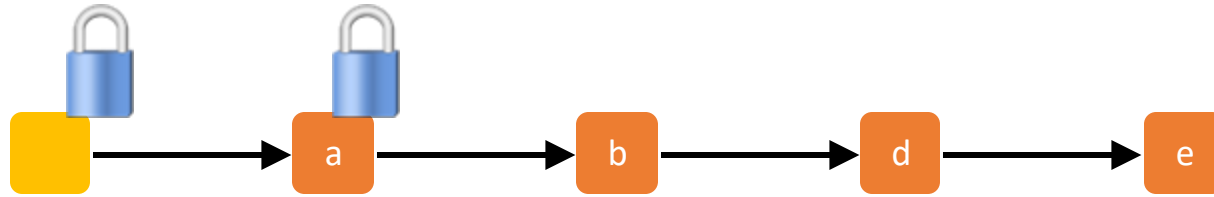
B: `remove(b)`



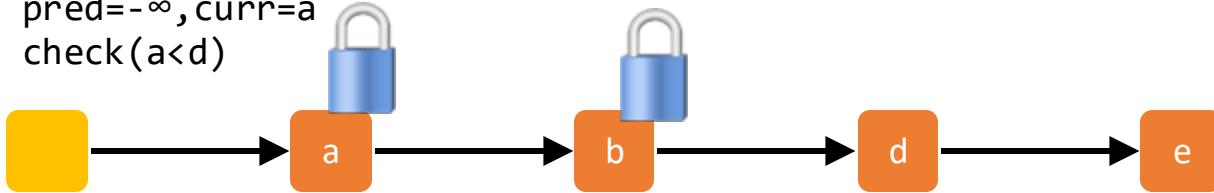
Also when adding lock the **successor** defensively.

Problem solved: b' not added!

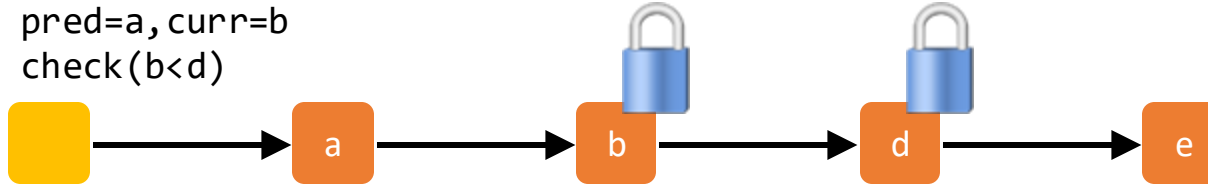
Hand-over-hand locking (remove d)



pred = $-\infty$, curr = a
check(a < d)

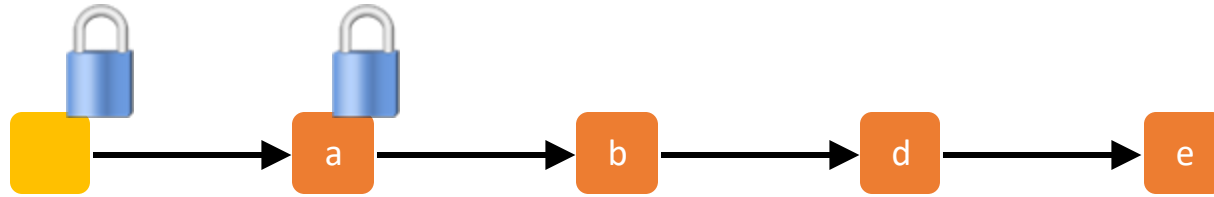


pred = a, curr = b
check(b < d)

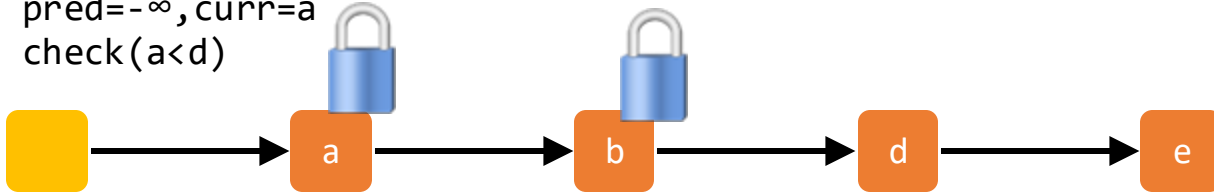


pred = b, curr = d
check(d < d)
if (d == d)
 remove(d)

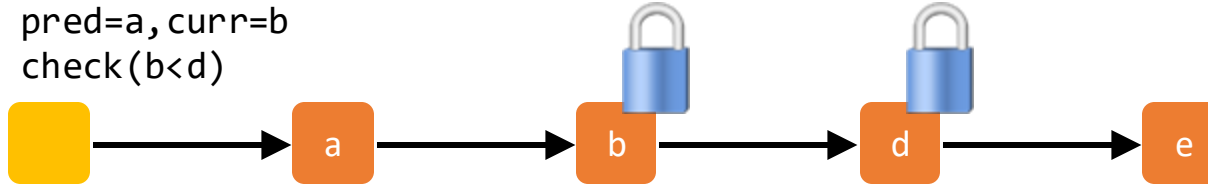
Hand-over-hand locking (remove d)



pred = $-\infty$, 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** (they cannot overtake another operation).
- 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

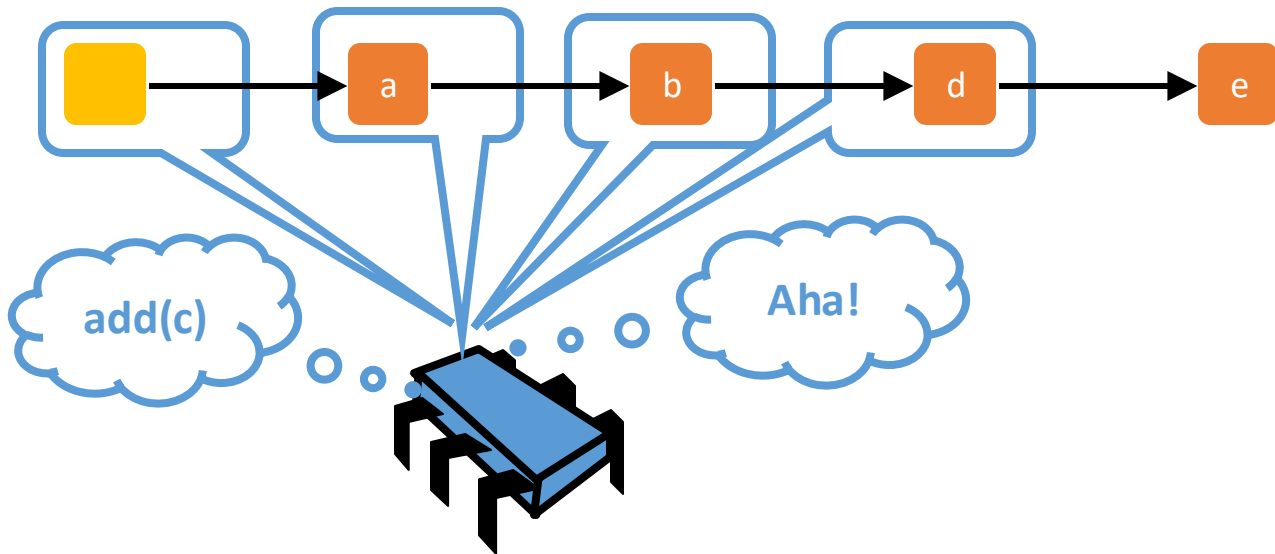
Idea

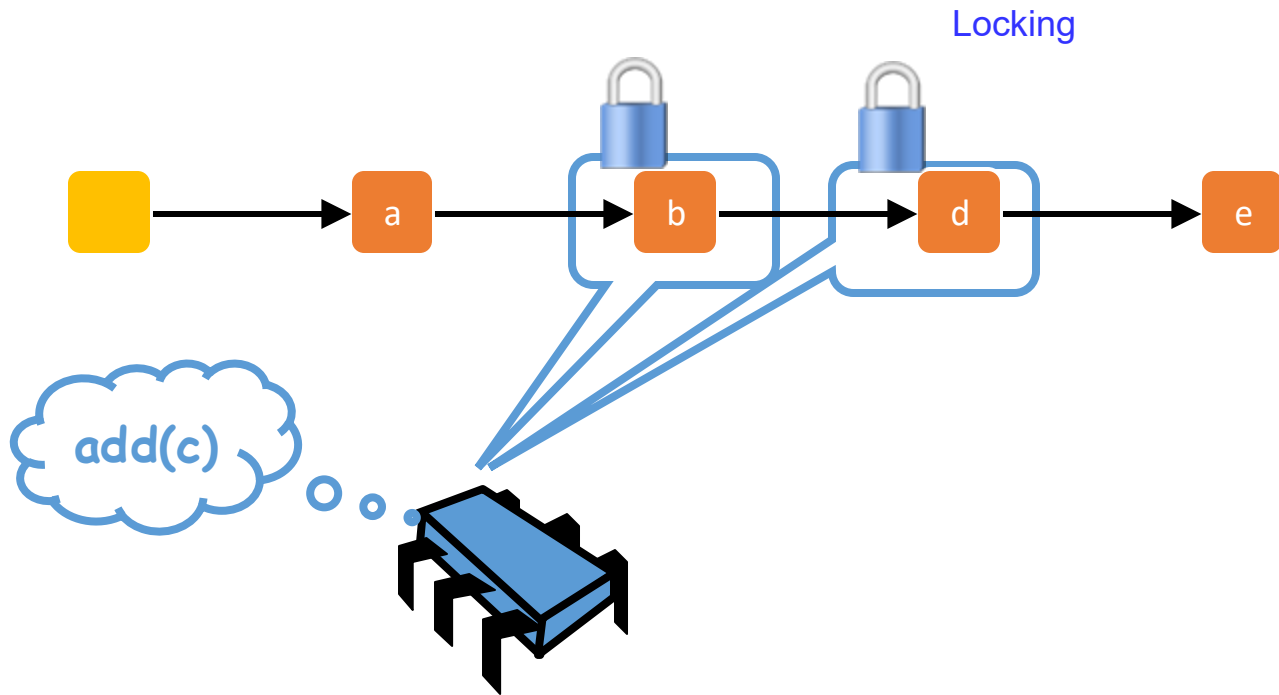
Algorithm:

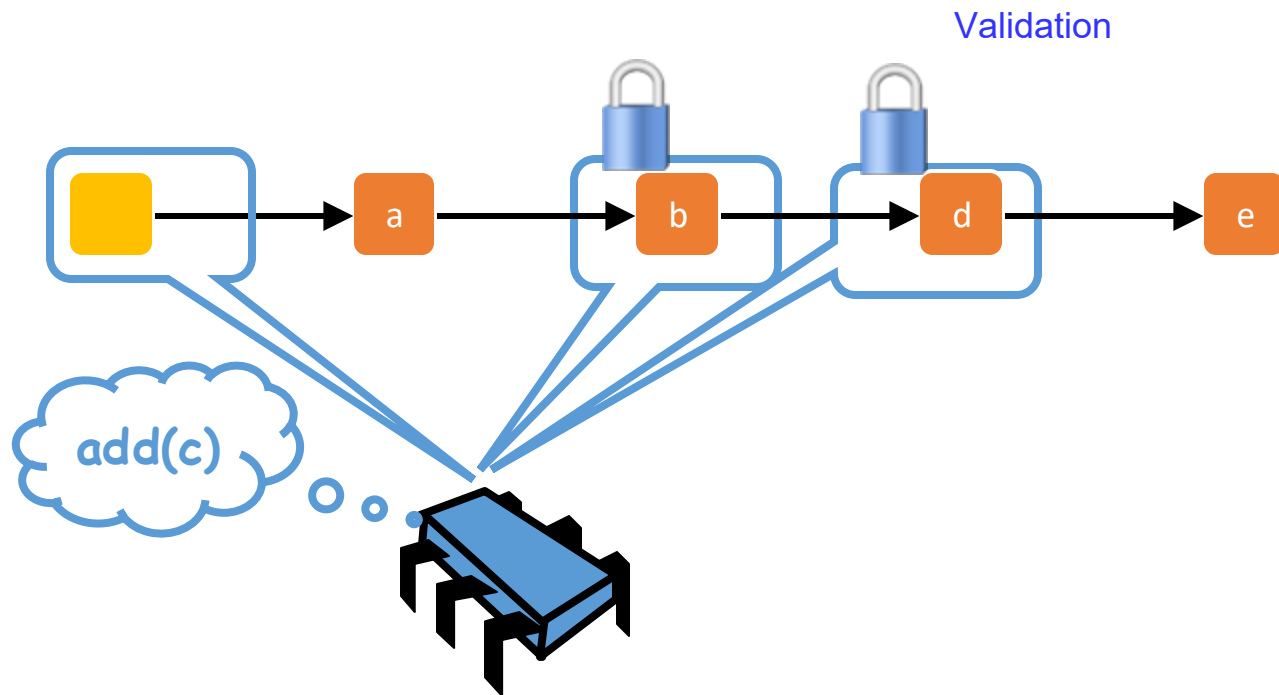
- find nodes without locking,
- then lock the two nodes and
- check that everything is ok (**validation**)
 - if so perform the operation (add, remove or contains) and return true
 - if not return false
- finally release the two locks

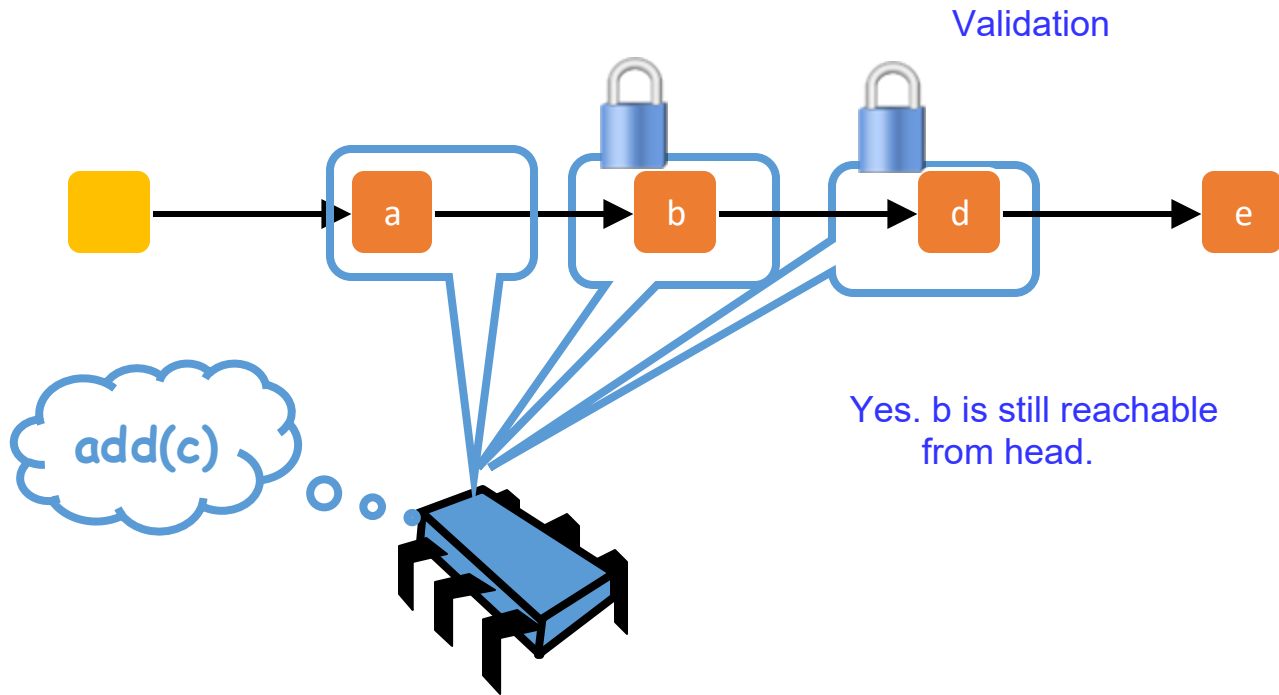
e.g. `add(c)`

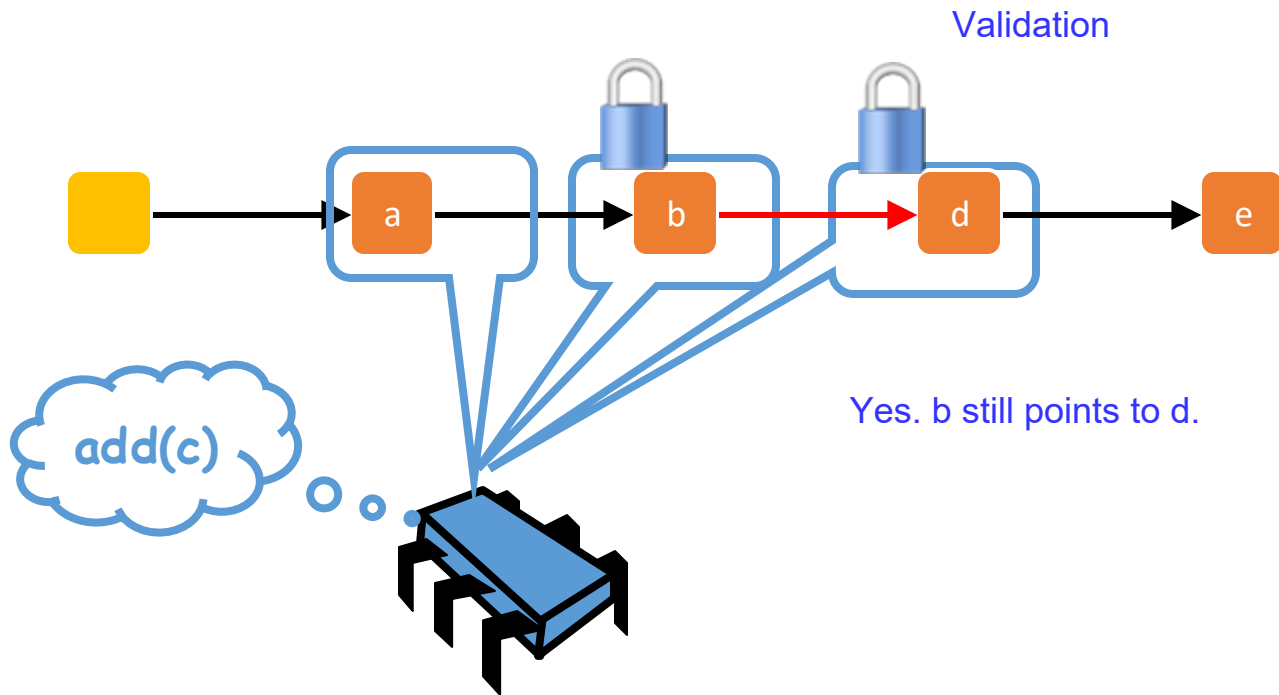
Finding without locking

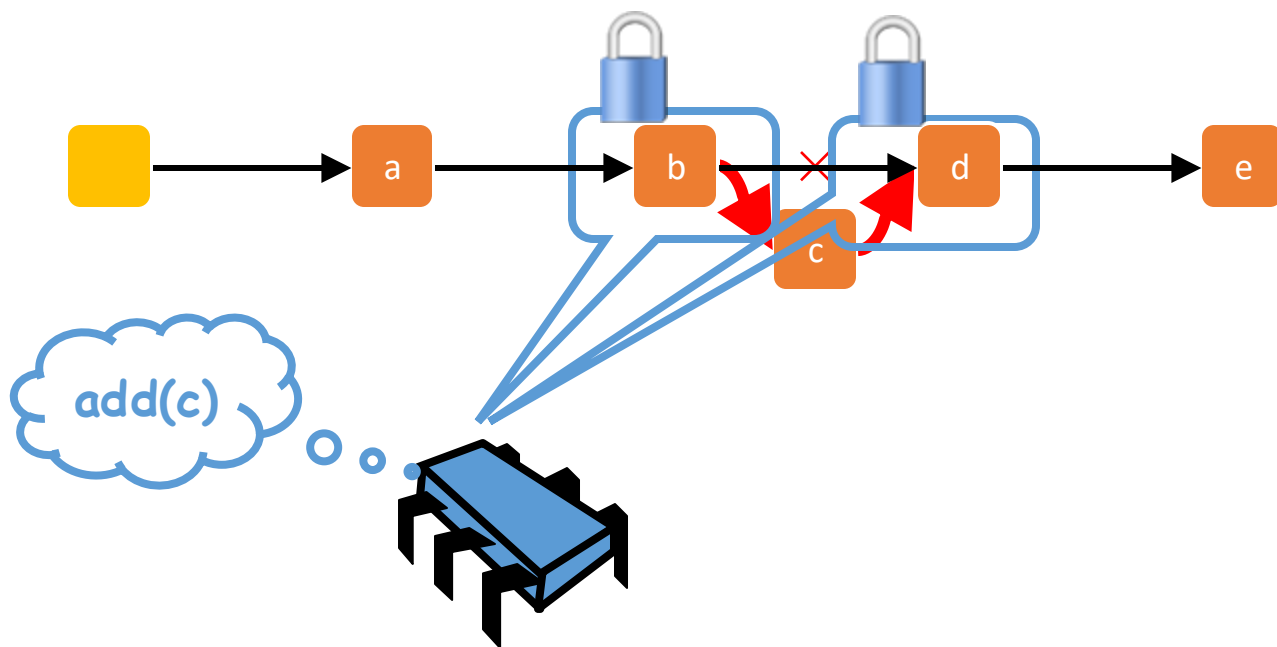


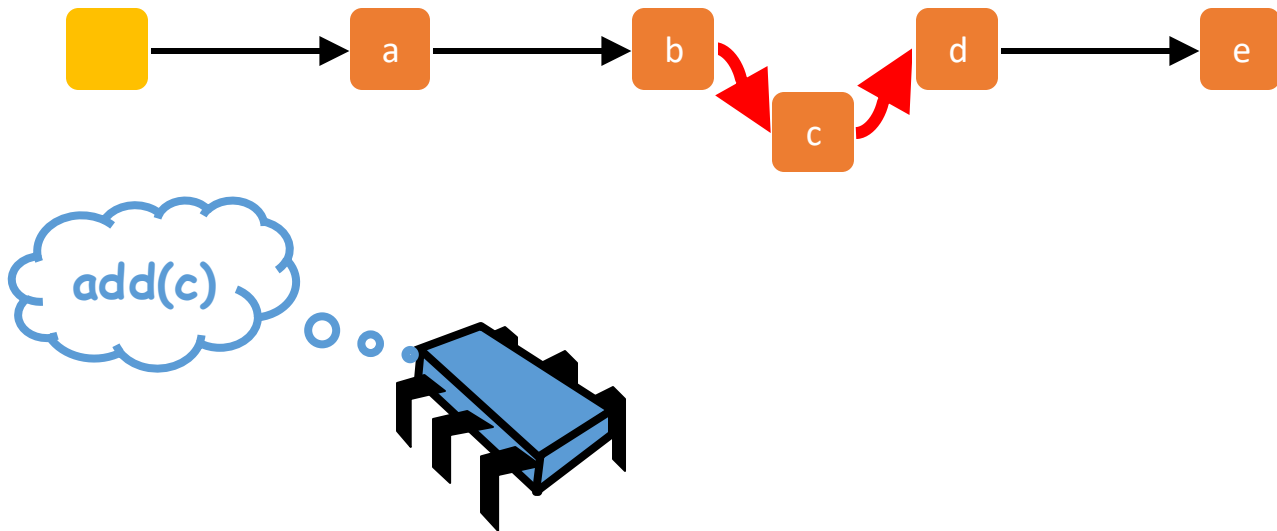








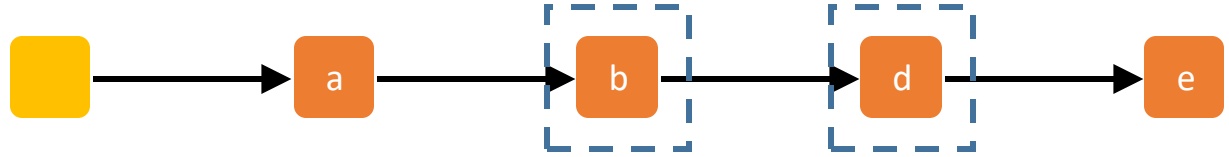




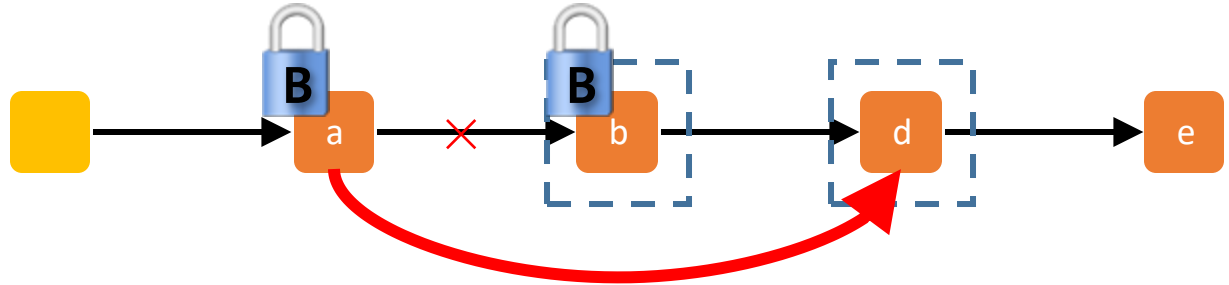
Validation: what can go wrong?

A: add(c)

A: find insertion point



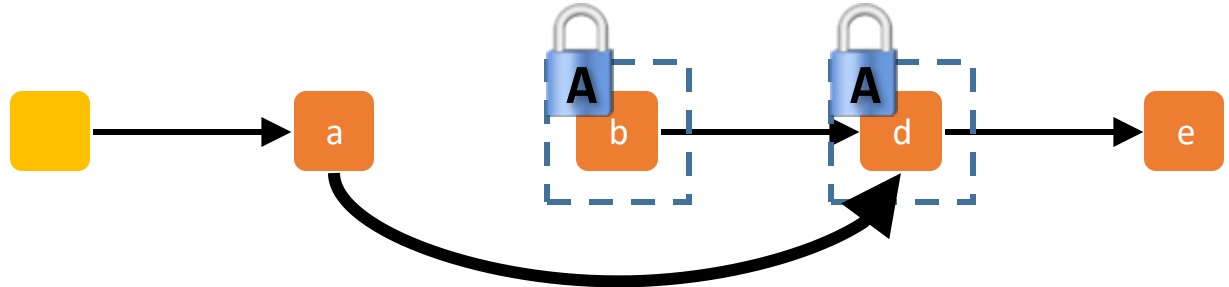
B: remove(b)



A: lock

A: validate: rescan

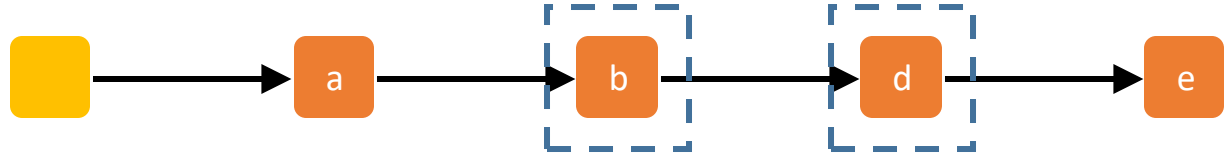
A: b not reachable
→ return false



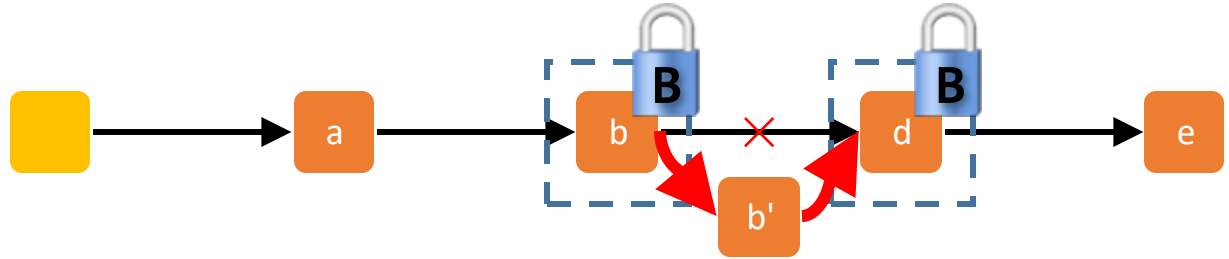
Validation: what can go wrong?

A: add(c)

A: find insertion point



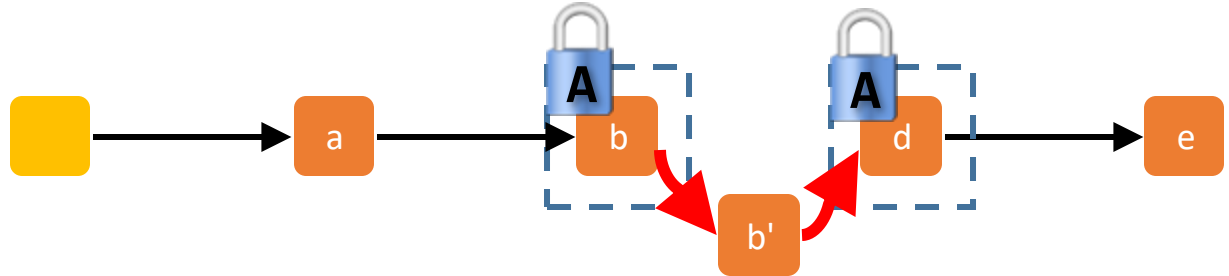
B: add(b')



A: lock

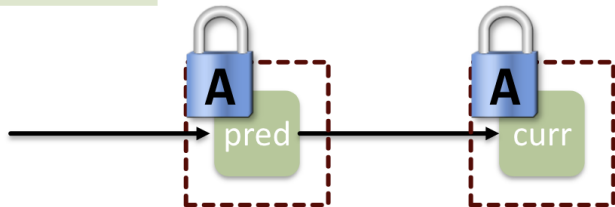
A: validate: rescan

A: d != succ(b)
→ return false



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;  
}
```



Correctness (remove c)

If

- nodes b and c both locked
- node b still accessible
- node c still successor to b

then

- neither will be deleted
- ok to delete and return true

If

- nodes b and d both locked
- node b still accessible
- node d still successor to b

then

- neither will be deleted
- no thread can add between b and d
- ok to 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



Why didn't we need to validate with hand-over-hand locking?

Lazy Synchronisation

Lazy List

Like optimistic list but

- scan only once
- contains() never locks

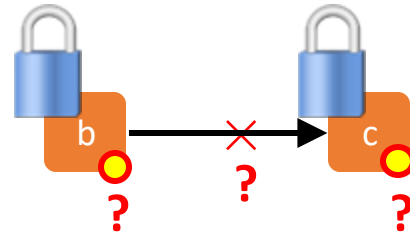
How?

- 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)

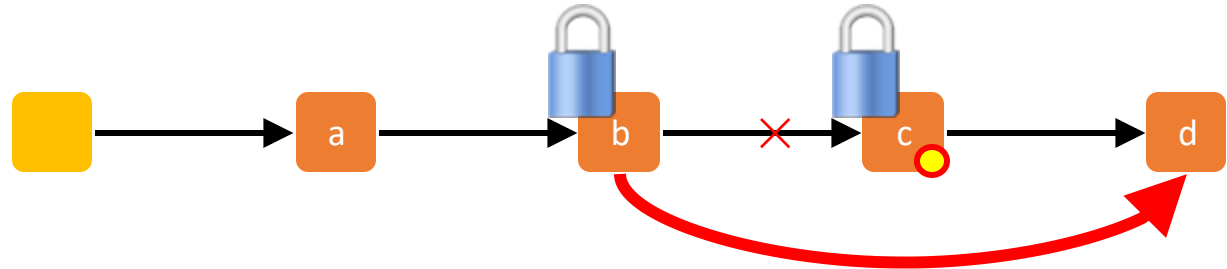
Validate (new validation)

Logical delete: mark current node as removed

Physical delete: redirect predecessor's next



e.g. remove(c)



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)

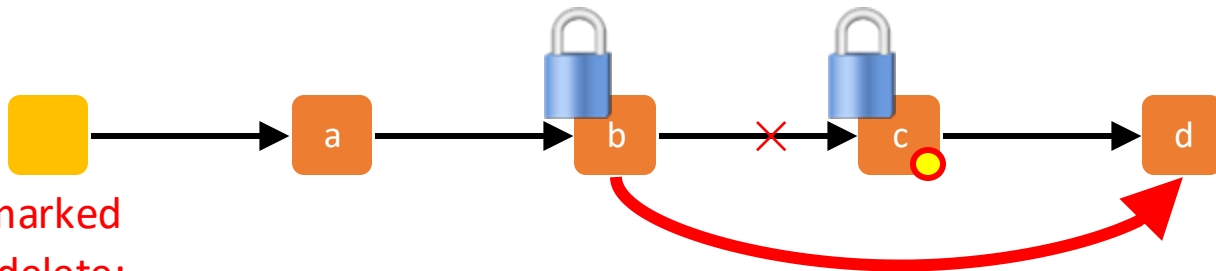
lock

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

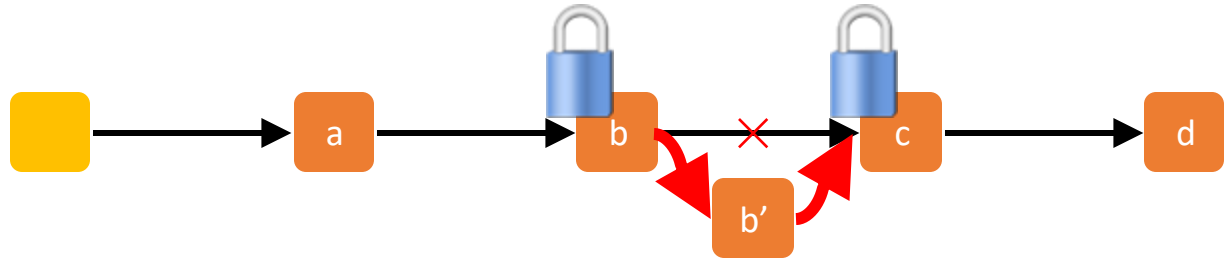
Find nodes to where to add (as before)

Lock predecessor and current (as before)

Validate (new validation)

Physical add: change predecessor's next

e.g. `add(b')`

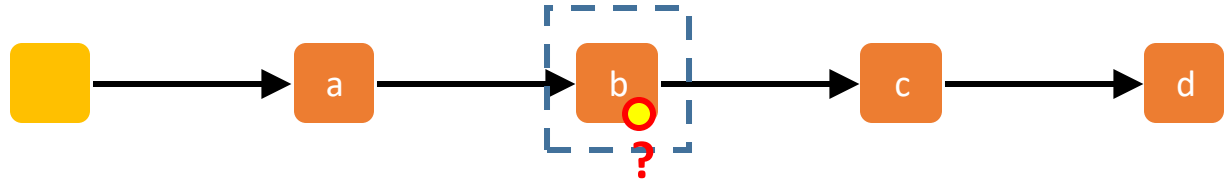


Lazy List: Contains

Find nodes to return without locking

Return true if node is not marked

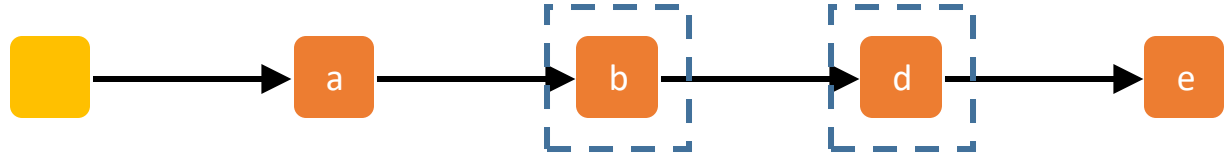
e.g. `contains(b)`



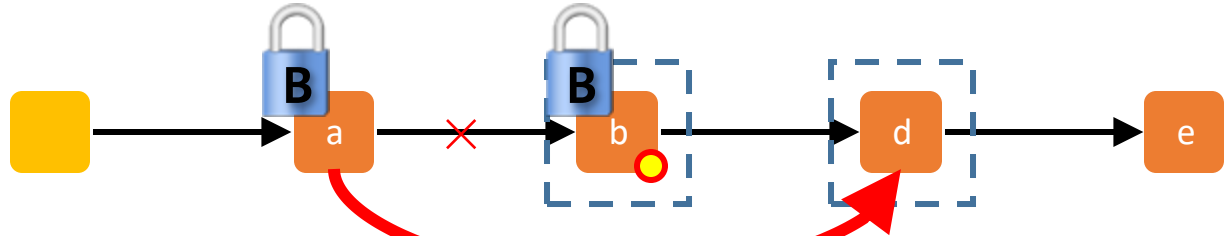
New Validation: What can go wrong?

A: add(c)

A: find insertion point



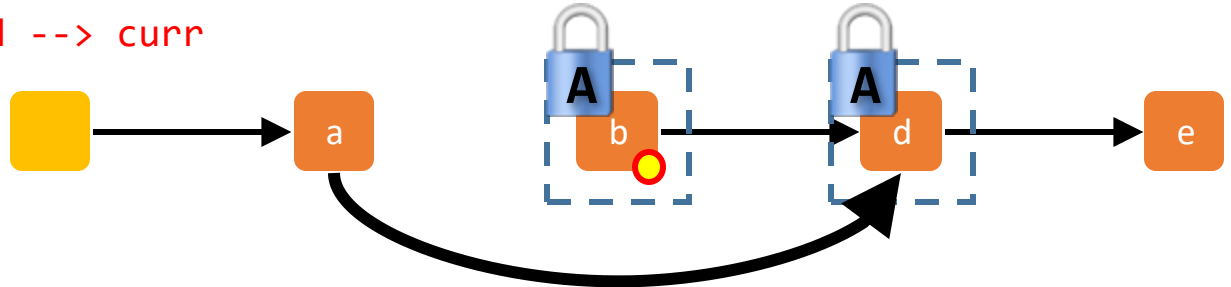
B: remove(b)



A: lock

A: validate: marks + pred --> curr

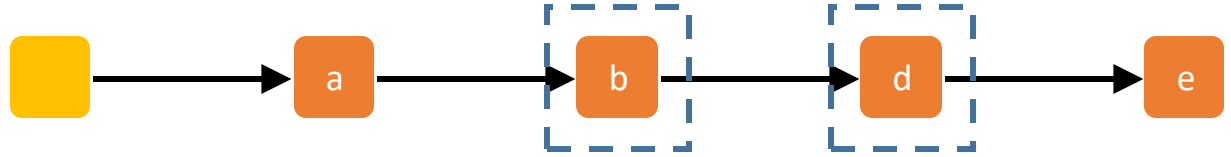
A: b marked
→return false



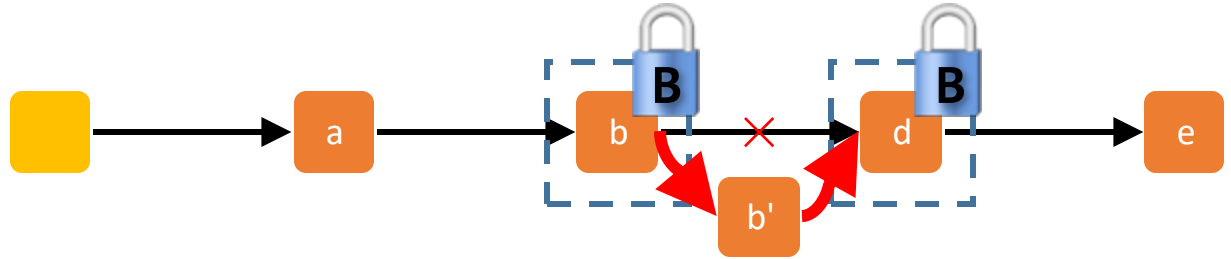
New Validation: What can go wrong?

A: add(c)

A: find insertion point



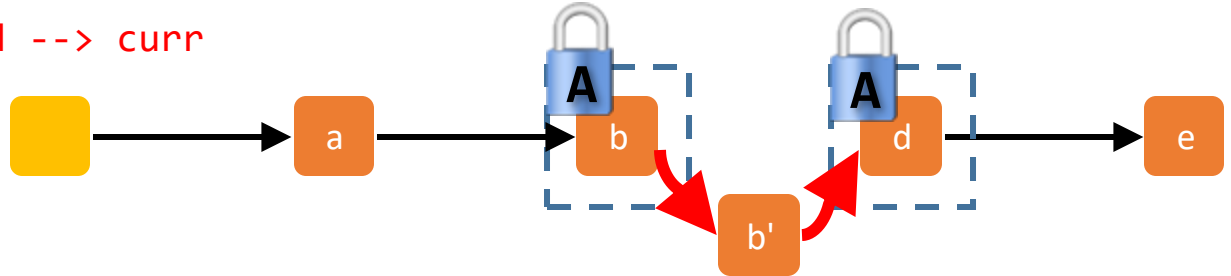
B: add(b')



A: lock

A: validate: marks + pred --> curr

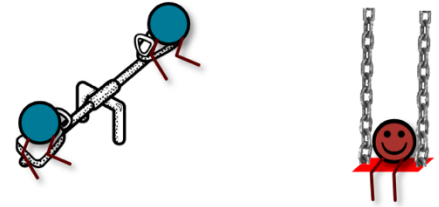
A: pred -x-> curr
→return false



Locks performance

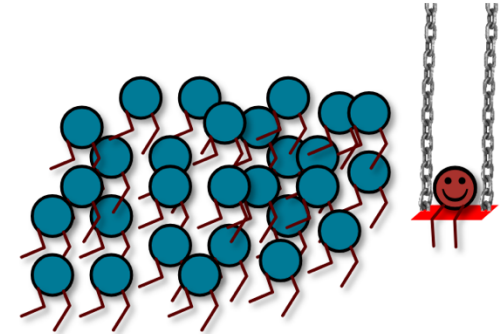
■ Uncontended case

- 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

Lock-free Datastructures

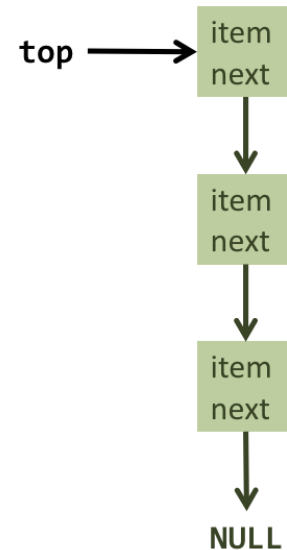
Stack Node

```
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;  
    }  
}
```



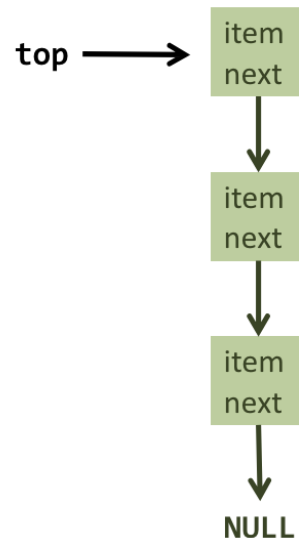
Blocking Stack

```
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;  
    }  
}
```



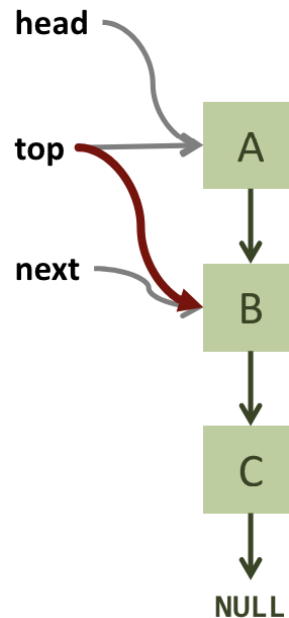
Non-blocking Stack

```
public class ConcurrentStack {  
    AtomicReference<Node> top = new AtomicReference<Node>();  
  
    public void push(Long item) { ... }  
    public Long pop() { ... }  
}
```



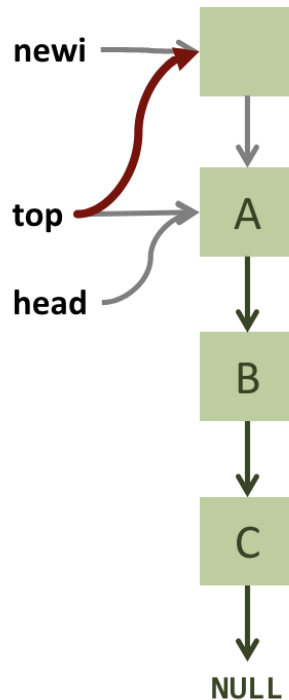
Pop

```
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;  
}
```



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));  
}
```



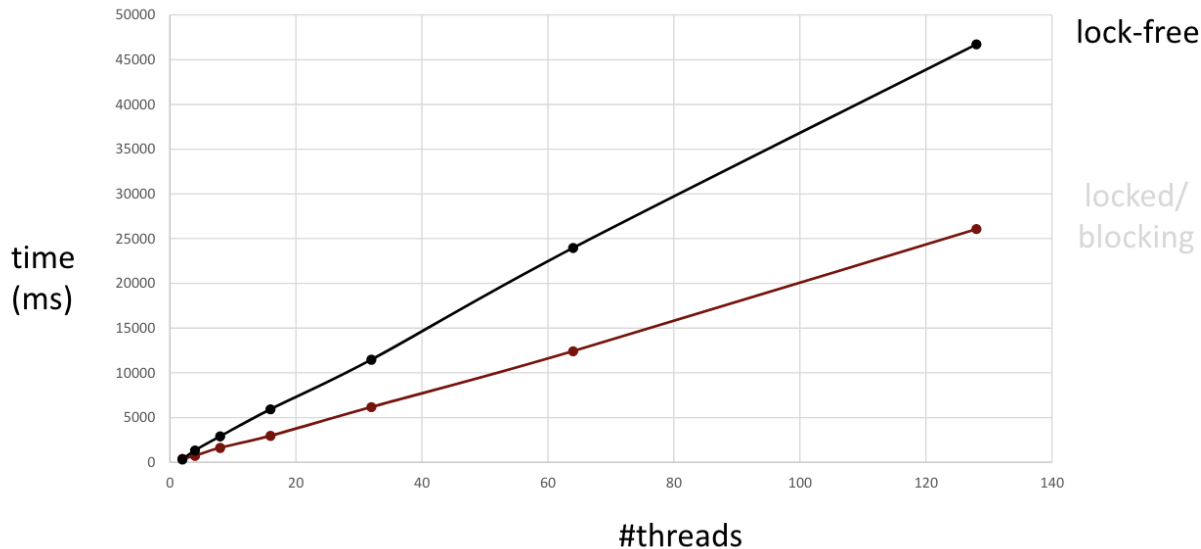
What's the benefit?

Lock-free programs are **deadlock-free** by design.

How about
performance?

n threads
100,000 push/pop operations
10 times

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

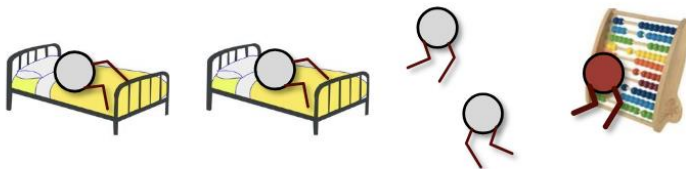


Performance

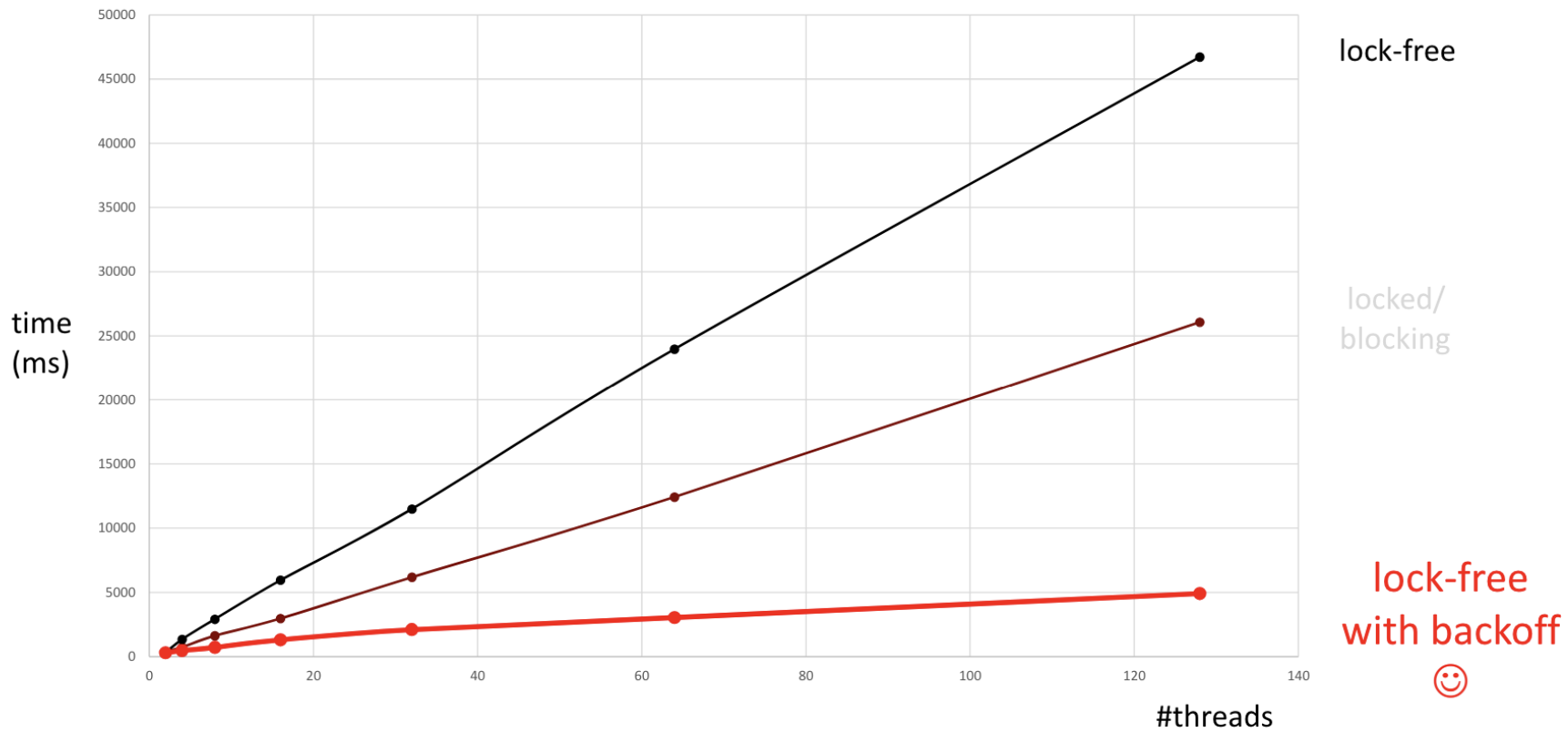
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

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

→ Backoff, again.

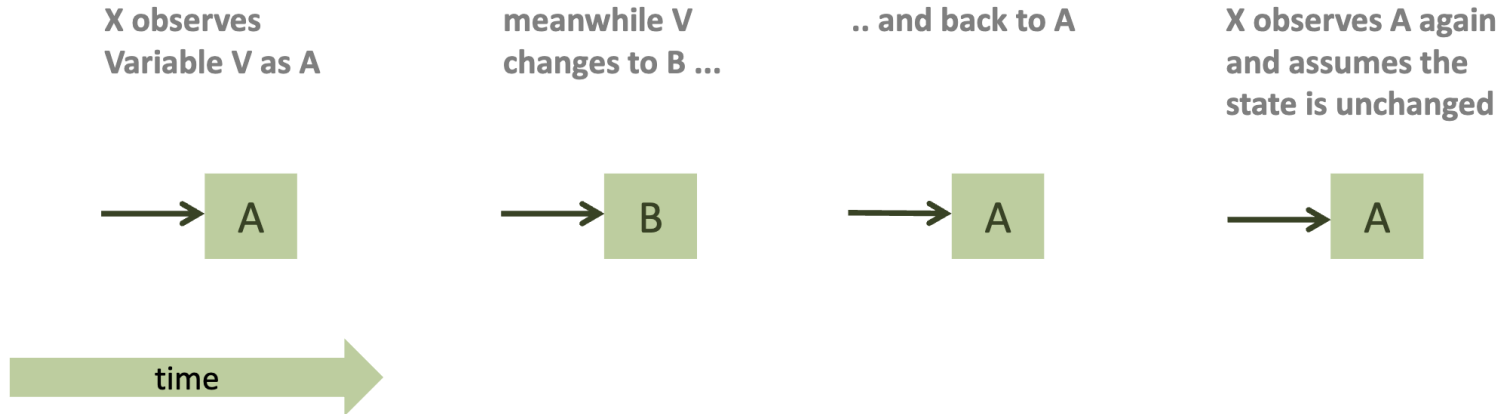


With backoff



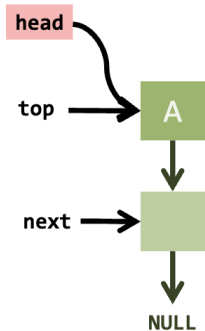
ABA Problem

"The ABA problem occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed."

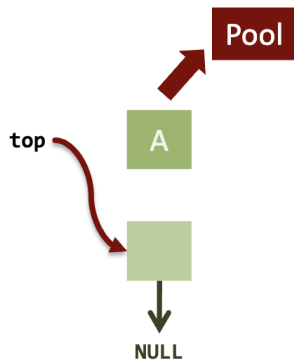


ABA Problem

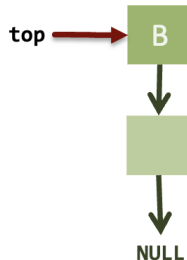
Thread X
in the middle
of pop: after read
but before CAS



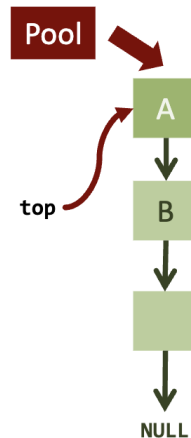
Thread Y
pops A



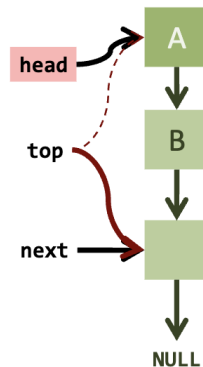
Thread Z
pushes B



Thread Z'
pushes A



Thread X
completes pop



time

```
public Long pop() {  
    Node head, next;  
    do {  
        head = top.get();  
        if (head == null) return null;  
        next = head.next;  
    } while (!top.compareAndSet(head, next));  
    Long item = head.item; pool.put(head); return item;  
}
```

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

Pointer Tagging

Use some bits of the address for an always incrementing counter.

ABA Problem less likely.

Still possible when the counter overflows.

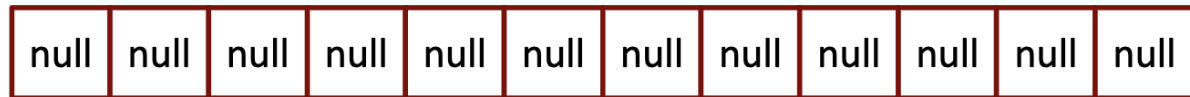
Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

- before X reads P, it marks it hazardous by entering it in one of the n (n= number threads) slots of an array associated with the data structure (e.g., the stack)
- When finished (after the CAS), process X removes P from the array
- Before a process Y tries to reuse P, it checks all entries of the hazard array

Hazard Pointers



0

nThreads-1

```
boolean isHazarduous(Node node) {  
    for (int i = 0; i < hazarduous.length(); ++i)  
        if (hazarduous.get(i) == node)  
            return true;  
    return false;  
}  
  
void setHazarduous(Node node) {  
    hazarduous.set(id, node); // id is current thread id  
}
```


Kahoot