

CS636: Concurrent Data Structures

Swarnendu Biswas

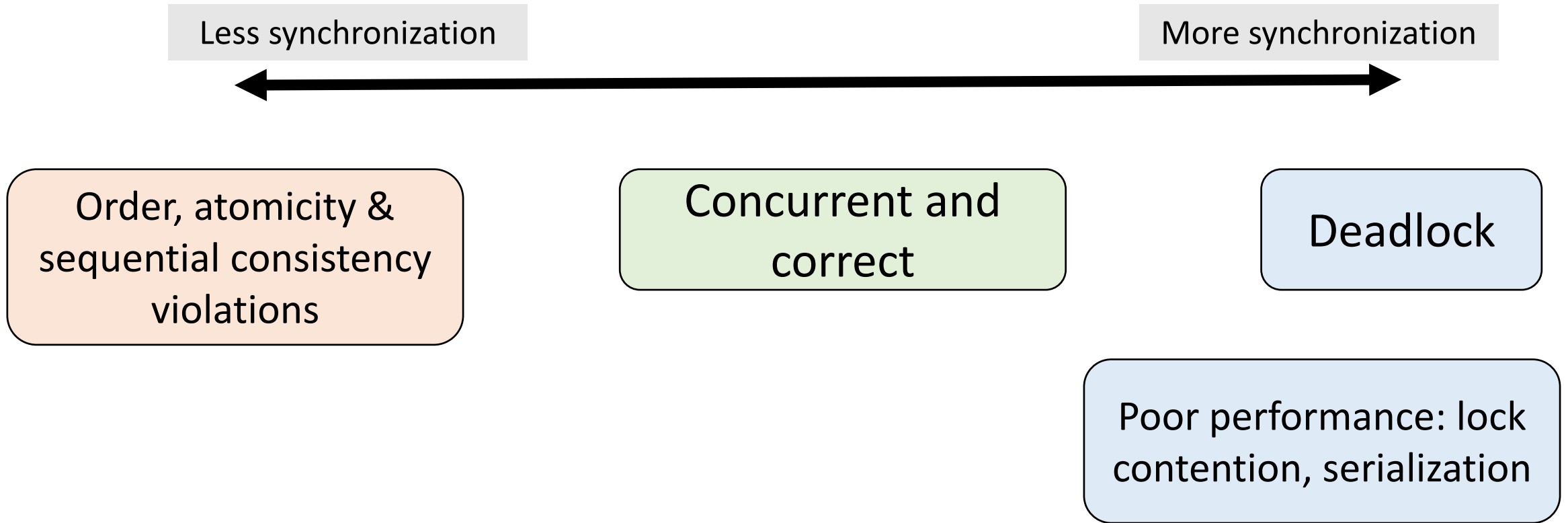
Semester 2018-2019-II
CSE, IIT Kanpur

Need for Concurrent Data Structures

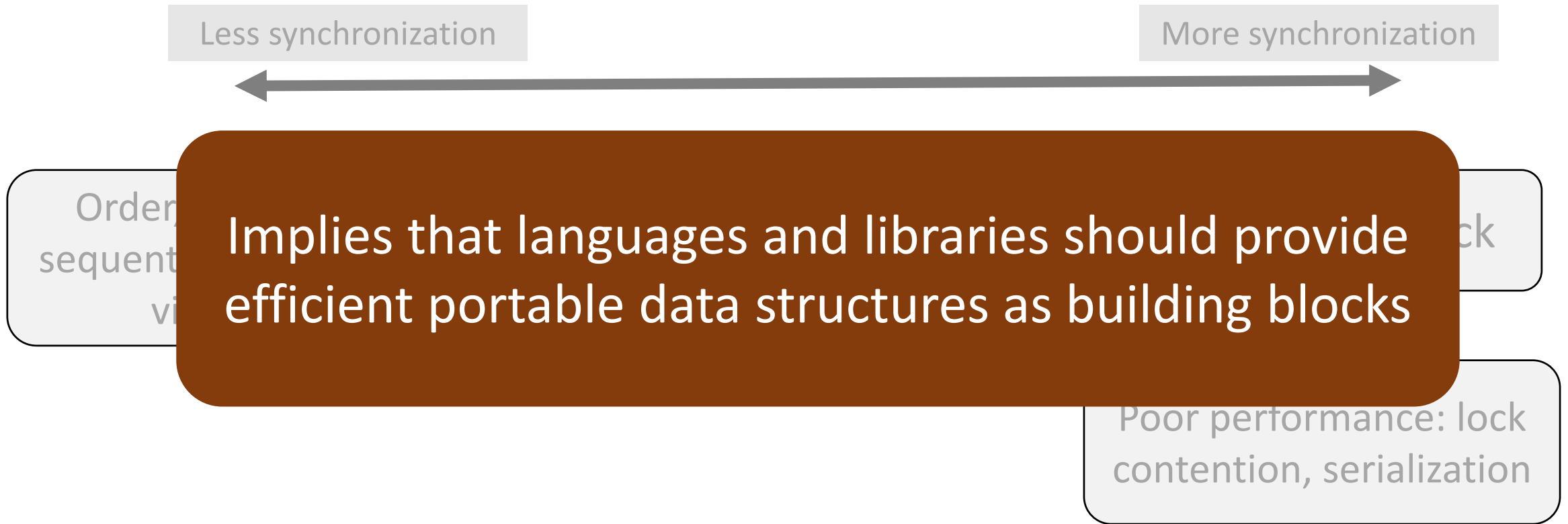
Multithreaded/concurrent programming is now mainstream

Using more hardware resources may not always translate to speedup

Challenges with Concurrent Programming



Need for Concurrent Data Structures



Designing a Concurrent Set Data Structure

Designing A Set Data Structure

```
public interface Set<T> {  
    boolean add(T x);  
    boolean remove(T x);  
    boolean contains(T x);  
}
```

add(x)

- adds x to the set and returns true if and only if x was not already present

remove(x)

- removes x from the set and returns true if and only if x was present

contains(x)

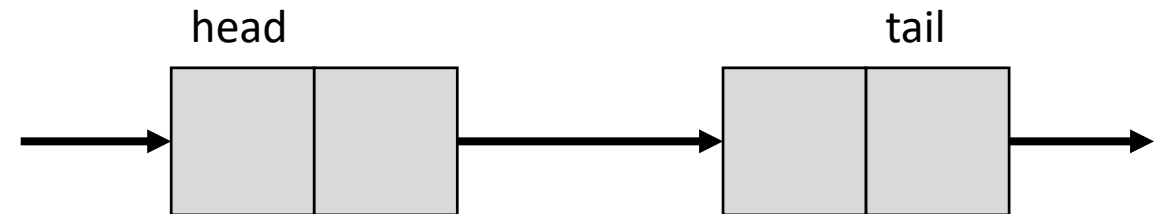
- returns true if and only if x is present in the set

Designing A Set Data Structure using Linked Lists

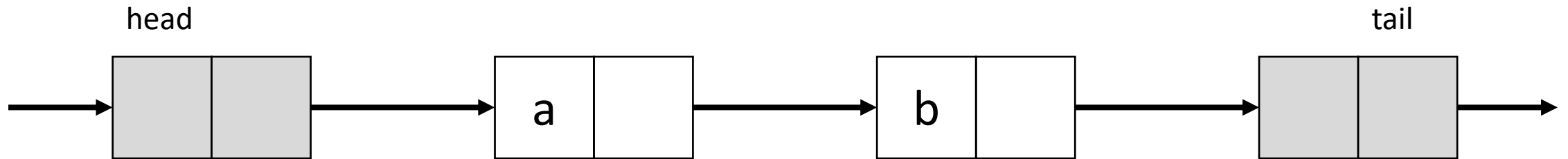
```
class Node {  
    T data;  
    int key;  
    Node next;  
}
```

- key field is the data's hash code, to help with efficient search.

- Two sentinel nodes
 - head and tail



A Set Instance



Invariants

- No duplicates
- Nodes are sorted based on the key value
- `tail` is reachable from `head`

A Thread Unsafe Set Data Structure

```
public class UnsafeList<T> {  
    private Node head;  
  
    public UnsafeList() {  
        head = new Node(Integer.MIN_VALUE);  
        head.next = new Node(Integer.MAX_VALUE);  
    }  
}
```

A Thread Unsafe Set Data Structure: add()

```
public boolean add(T x) {  
    Node pred, curr;  
    int key = x.hashCode();  
    pred = head;  
    curr = pred.next;  
    while (curr.key < key) {  
        pred = curr;  
        curr = curr.next;  
    }  
}
```

```
    if (key == curr.key) {  
        return false;  
    } else {  
        Node node = new Node(x);  
        node.next = curr;  
        prev.next = node;  
        return true;  
    }  
}
```

A Thread Unsafe Set Data Structure: remove()

```
public boolean remove(T x) {  
    Node pred, curr;  
    int key = x.hashCode();  
    pred = head;  
    curr = pred.next;  
    while (curr.key < key) {  
        pred = curr;  
        curr = curr.next;  
    }
```

```
        if (key == curr.key) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }
```

A Thread Unsafe Set Data Structure: contains()

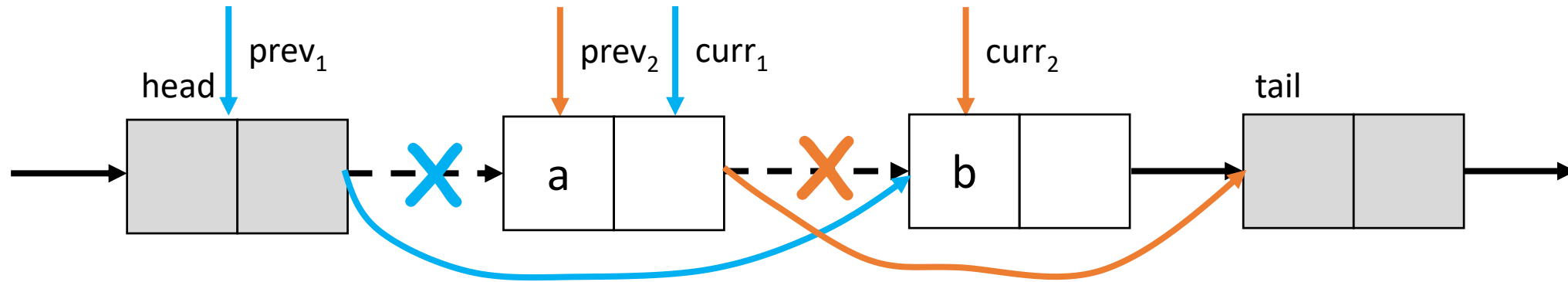
```
public boolean contains(T x) {  
    Node pred, curr;  
    int key = x.hashCode();  
    pred = head;  
    curr = pred.next;  
    while (curr.key < key) {  
        pred = curr;  
        curr = curr.next;  
    }  
    if (key == curr.key) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

A Thread Unsafe Set Data Structure: remove()

```
public boolean remove(T x) {  
    Node pred, curr;  
    int key = x.hashCode();  
    pred = null;  
    curr = head;  
    while (curr != null) {  
        if (key == curr.key) {  
            pred.next = curr.next;  
            return true;  
        }  
        pred = curr;  
        curr = curr.next;  
    }  
}
```

Can you give an example to show remove() is not Thread Safe?

Unsafe Set: Incorrect remove()



- Thread 1 is executing remove(a)
- Thread 2 is executing remove(b)

A Concurrent Set Data Structure

```
public class CoarseList<T> {  
    private Node head;  
    private Lock lock = new ReentrantLock();  
  
    public CoarseList() {  
        head = new Node(Integer.MIN_VALUE);  
        head.next = new Node(Integer.MAX_VALUE);  
    }  
}
```

A Concurrent Set Data Structure: add()

```
public boolean add(T x) {  
    Node pred, curr;  
    int key = x.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
    }
```

```
        if (key == curr.key) {  
            return false;  
        } else {  
            Node node = new Node(x);  
            node.next = curr;  
            prev.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```


A Concurrent Set Data Structure: remove()

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

Performance Metrics of Concurrent Data Structures

- Speedup measures how effectively is an application utilizing resources
 - Linear speedup is desirable
 - Data structures whose speedup grows with resources is desirable
- Amdahl's law says we need to reduce amount of serialized code
- Lock contention
 - Lock implementations with single memory location can introduce additional coherence traffic and memory traffic due to unsuccessful acquires
- Blocking or nonblocking

Challenges in Designing Concurrent Data Structures

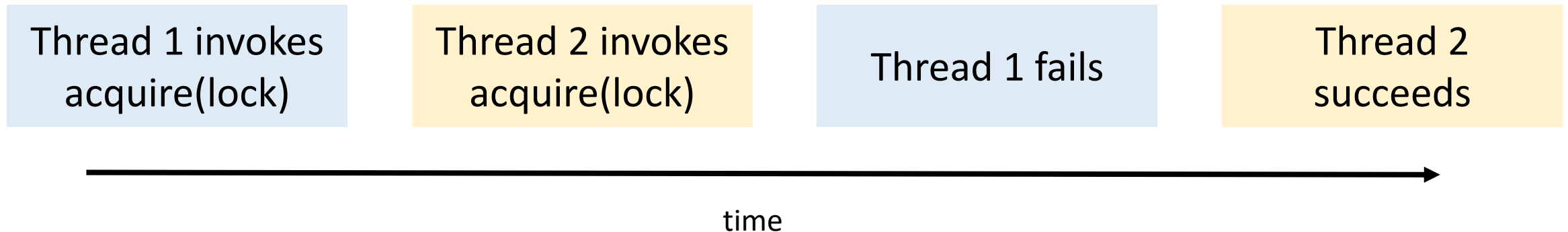
- Multiple threads can access a shared object
 - E.g., a node in our Set data structure
- Situation:
 - Thread 1 is checking for contains(a)
 - Thread 2 is executing remove(a)
- How do you reason about the outcome?

Reasoning about Correctness

- Identify invariants and make sure they always hold
 - An item is in the set if and only if it is reachable from head
- Safety property is linearizability
- Liveness property are starvation and deadlock-freedom

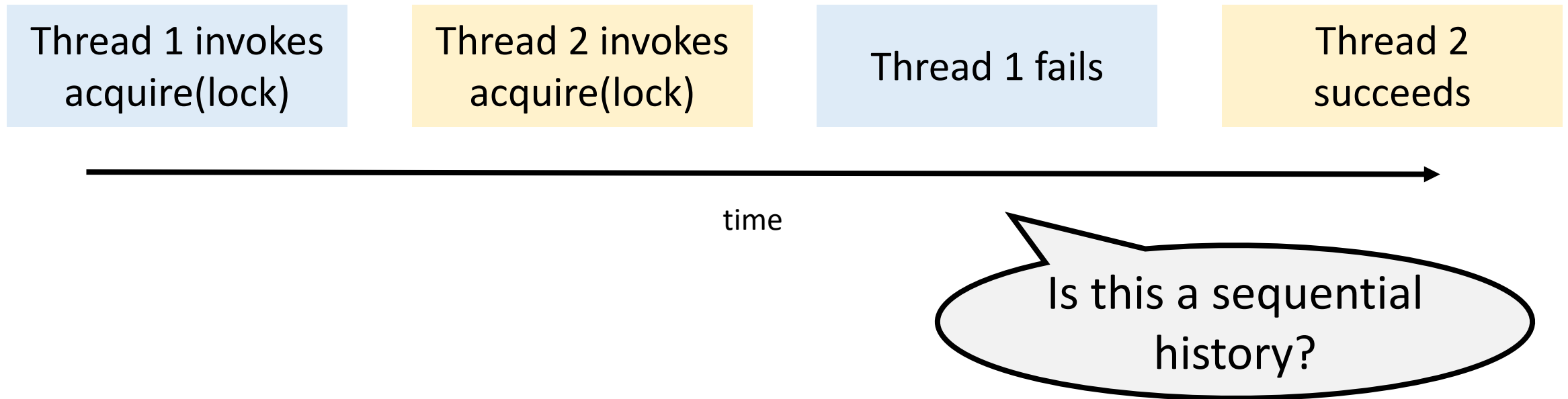
Understanding Linearizability

- Say you perform some operations on an object
 - Each operation requires an invocation on that object, followed by a response
- A history is a sequence of invocations and responses on an object made by concurrent threads



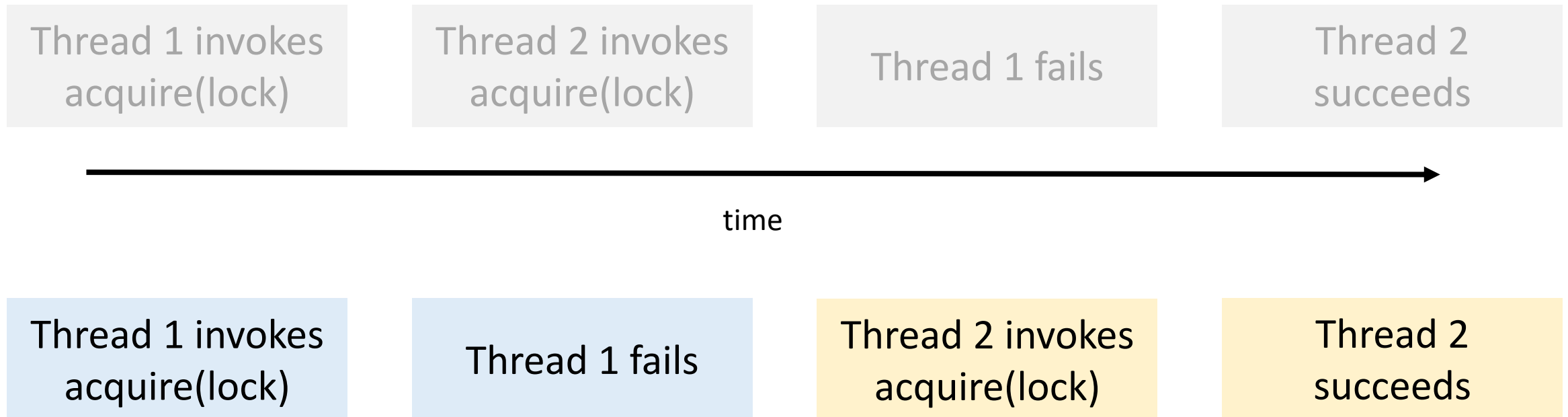
Understanding Linearizability

- Sequential history is where all invocations and responses are instantaneous



Understanding Linearizability

- Sequential history is where all invocations and responses are instantaneous



Linearizability

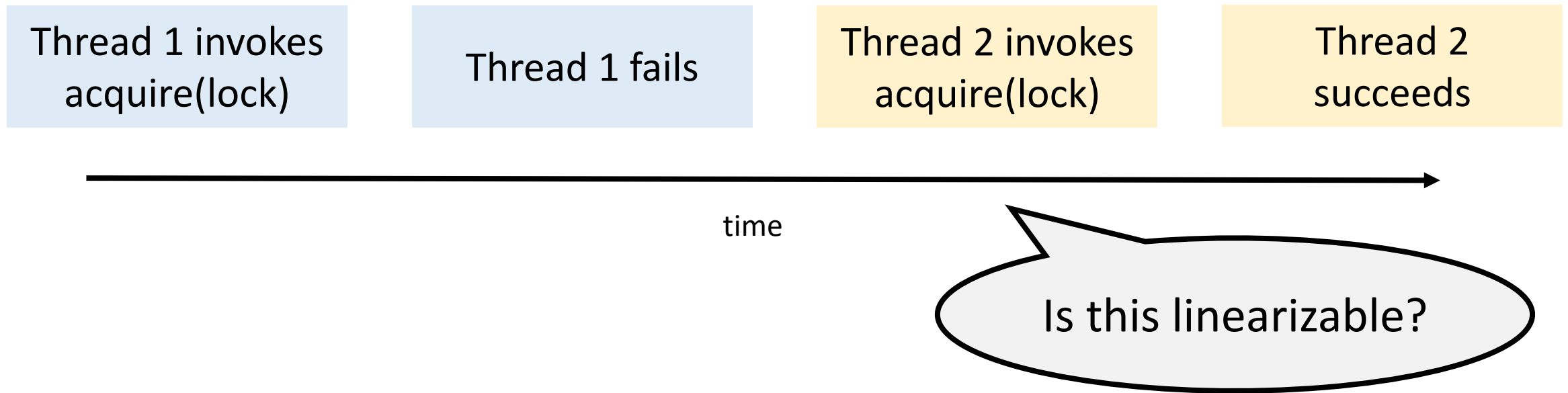
- A history (set of operations) σ is **linearizable** if
 - For every completed operation in σ , the operation returns the same result in the execution as it would return if every operation in σ would have been completed one after the other
 - If an operation $op1$ completes before operation $op2$, then $op1$ precedes $op2$ in σ .

Linearizability

- A history (set of operations) σ is **linearizable** if
 - For every completed operation in σ , the operation returns the same result in the execution as it would return if every operation in σ would have been completed one after the other
 - If an operation $op1$ completes before operation $op2$, then $op1$ precedes $op2$ in σ .
- Simpler words
 - Invocations and response can be reordered to form a sequential history
 - Sequential history is correct according to the semantics of the object
 - If a response preceded an invocation in the original history, it must still precede it in the sequential reordering

Understanding Linearizability

- Sequential history



Understanding Linearizability

- Sequential history



- Successful linearization

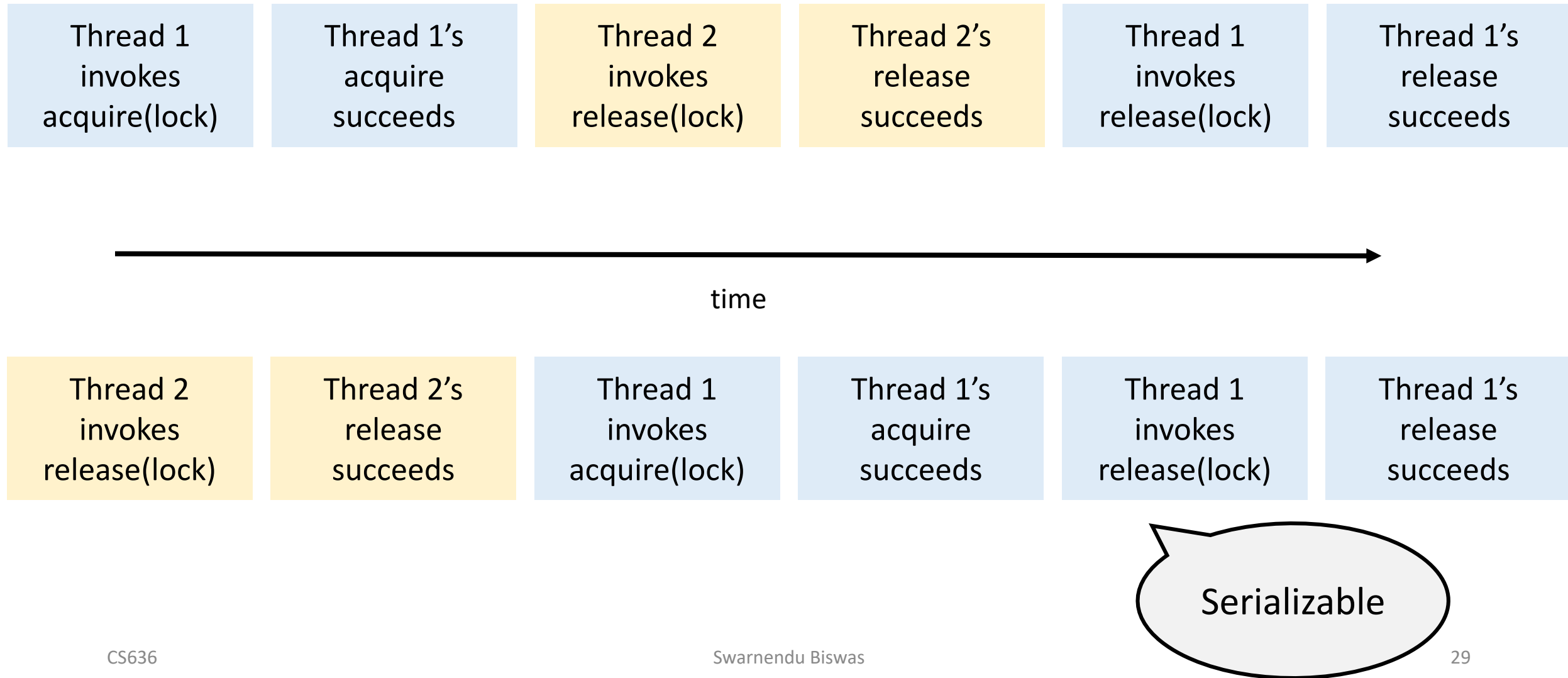


Linearization Point

- Linearization point is between the function invocation and response
- A single atomic step where the method call “takes effect”

What are the linearization points for `add()`, `remove()` and `contains()` for the coarsely synchronized Set?

Linearizability vs Serializability



Linearizability vs Serializability

Linearizability

- Property about operations on individual objects
 - Local property
- Requires real-time ordering

Serializability

- Property about transactions or group of operations on one or more objects
 - Global property
- Requires output is equivalent to some serial ordering

Linearizability vs Serializability

Linearizability

- Property about operations on individual objects
 - Local property
- Requires real-time ordering

Serializability

- Property about transactions or group of operations on one or more objects
 - Global property
- Requires output is equivalent to some serial ordering

“Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object” – Herlihy and Wing

Types of Synchronization

Coarse-grained synchronization

Fine-grained synchronization

Optimistic synchronization

Lazy synchronization

Nonblocking synchronization

Fine-Grained Synchronization

- Add a lock object to each list node

```
class Node {  
    T data;  
    int key;  
    Node next;  
    Lock lock;  
}
```

What are a few possible ideas to implement add() and remove()?

Is one lock per node enough?

Thread 1

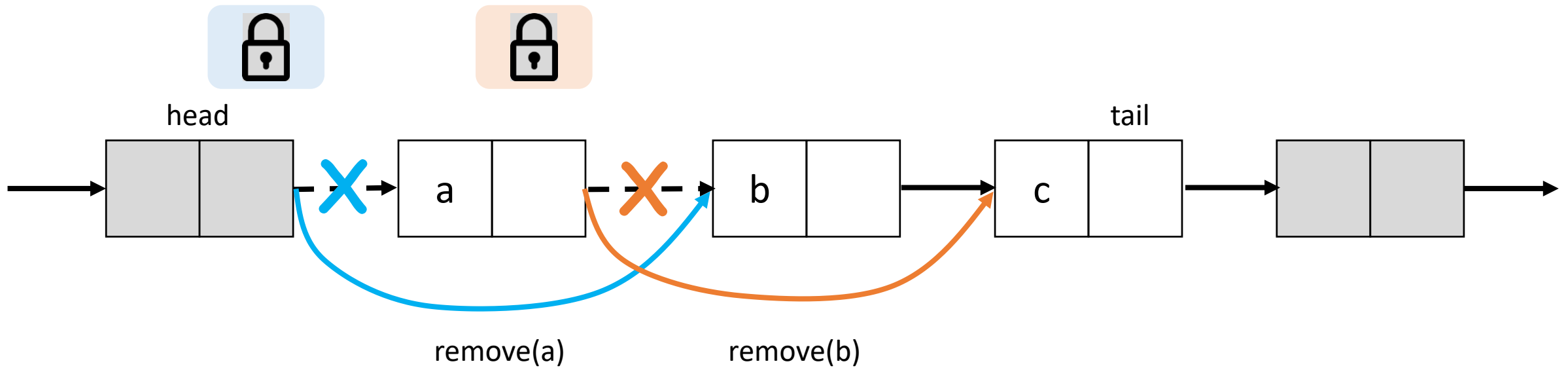
```
node0.mtx_lock.lock();  
node1 = node0.next;  
node0.mtx_lock.unlock();
```

Thread 2

```
// Remove node1 from list
```

```
node1.mtx_lock.lock();
```

Is one lock per node enough?



- Thread 1 is executing remove(a)
- Thread 2 is executing remove(b)

Fine-Grained Synchronization: add()

```
public boolean add(T x) {  
    int key = x.hashCode();  
    head.lock();  
    Node pred = head;  
    try {  
        Node curr = pred.next;  
        curr.lock();  
        try {  
            while (curr.key < key) {  
                pred.unlock();  
                pred = curr;  
                curr = curr.next;  
                curr.lock();  
            }  
            if (key == curr.key) {  
                return false;  
            } else {  
                Node node = new Node(x);  
                node.next = curr;  
                pred.next = node;  
                return true;  
            }  
        } finally {  
            curr.unlock();  
        }  
    } finally {  
        pred.unlock();  
    }  
}
```

Fine-Grained Synchronization: remove()

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

Need to avoid Deadlocks

- Deadlocks are always a problem with fine-grained locking
- For the Set data structure, each thread must acquire locks in some pre-determined order

Fine-Grained Set Design

Are there other problems with our fine-grained Set design?

Optimistic Synchronization

Optimistic strategy

- Access data without acquiring a lock
- Lock only when required
- **Validate** that the condition before locking is still valid
- If valid, then continue with access/mutation
- If invalid, start over

Optimistic strategy works well if conflicts are rare

Optimistic Synchronization: add()

```
public boolean add(T x) {
    int key = x.hashCode();
    while (true) {
        Node pred = 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) {
                    return false;
                } else {
                    Node node = new Node(x);
                    node.next = curr; prev.next = node;
                    return true;
                }
            }
        } finally {
            curr.unlock(); pred.unlock();
        }
    }
}
```

How could you validate?

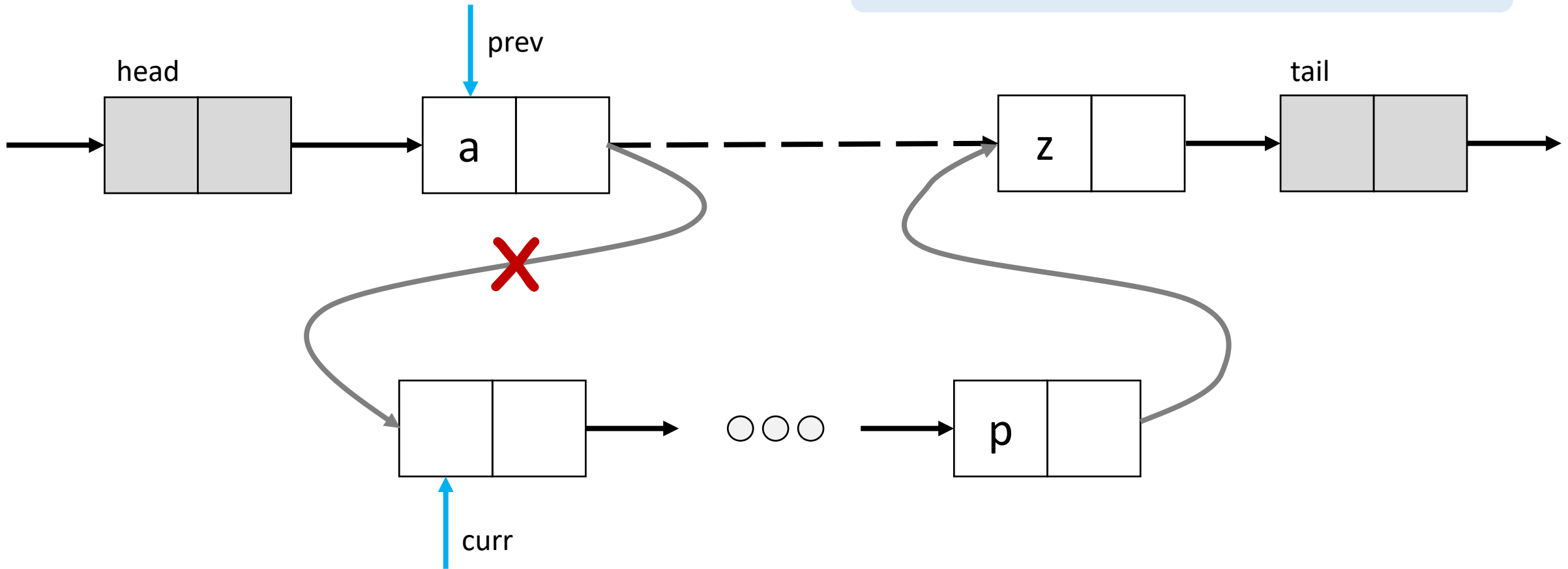
- Double check that the optimistic result is still valid
- Check that prev is reachable from head and `prev.next == curr`

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

Is validation necessary?

Is validation necessary?

- Thread 1 is executing remove(p)



Optimistic Synchronization: remove()

```
public boolean remove(T x) {  
    int key = x.hashCode();  
    while (true) {  
        Node pred = 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 {  
            curr.unlock(); pred.unlock();  
        }  
    }  
}
```

Optimistic Synchronization: contains()

```
public boolean contains(T x) {  
    int key = x.hashCode();  
    while (true) {  
        Node pred = head;  
        Node curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        pred.lock(); curr.lock();
```

```
        try {  
            if (validate(pred, curr)) {  
                return curr.key == key;  
            }  
        } finally {  
            curr.unlock(); pred.unlock();  
        }  
    }  
}
```

Optimistic Synchronization Design

Are there problems with our optimistic synchronization-based Set design?

Lazy Synchronization

Delay mutation
operations for
a later time

- Add a mark/flag on each node to indicate say deletion
- **Invariant:** every unmarked node is reachable from head

Behavior

- contains(): needs only one wait-free traversal
- add(): traverses the list, locks the predecessor, and inserts the node
- remove(): mark the target node logically removing it, then redirect the predecessor's next link physically removing it

Lazy Synchronization: add()

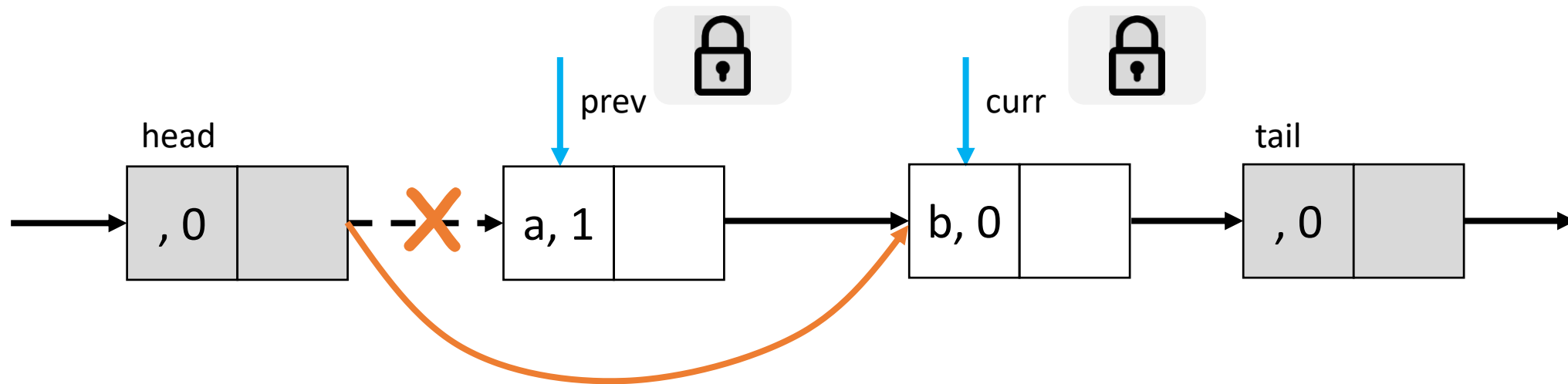
```
public boolean add(T x) {
    int key = x.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred, curr)) {
                    if (curr.key == key) {
                        return false;
                    } else {
                        Node node = new Node(x);
                        node.next = curr;
                        prev.next = node;
                        return true;
                    }
                }
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}
```

How could you validate?

- Check that both prev and curr are unmarked and prev.next == curr

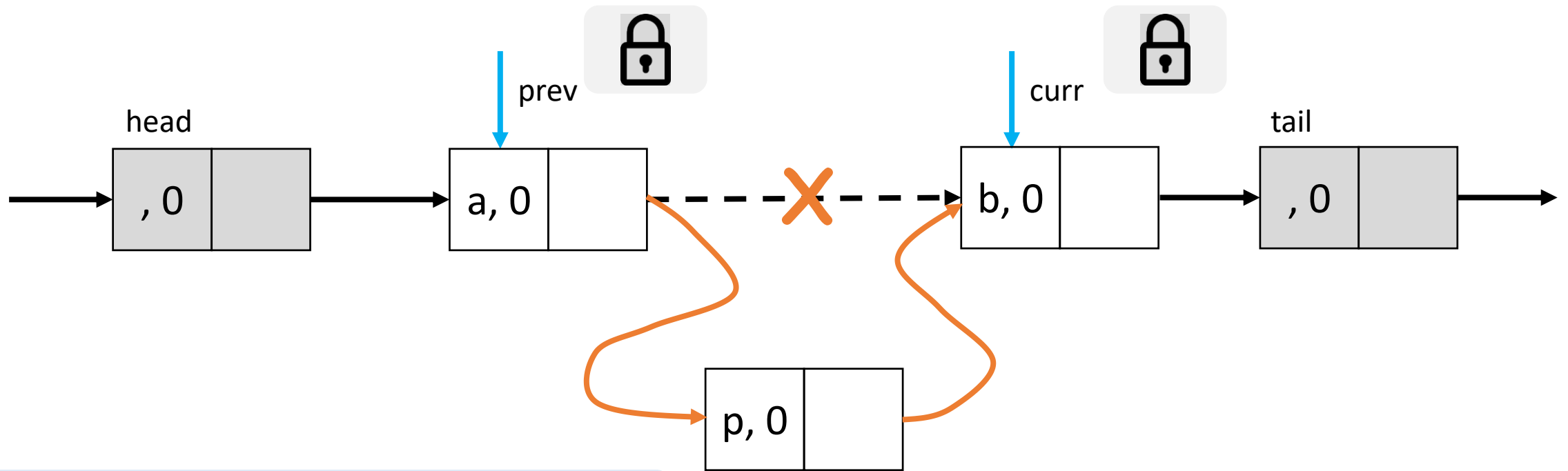
```
boolean validate(Node prev, Node curr) {  
    return !prev.marked && !curr.marked &&  
    prev.next == curr;  
}
```

Is validation really necessary?



- Thread 1 is executing `remove(b)`
- Thread 2 is executing `remove(a)`

Is validation really necessary?



- Thread 1 is executing `remove(b)`
- Thread 2 is executing `add(p)`

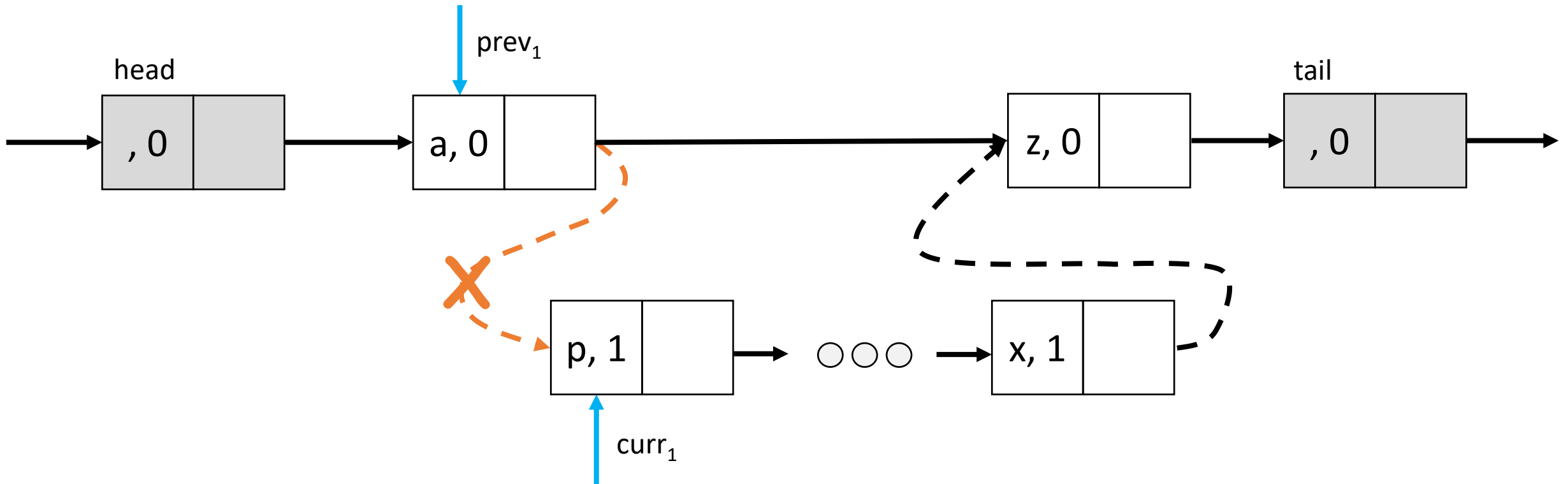
Lazy Synchronization: remove()

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

Lazy Synchronization: contains()

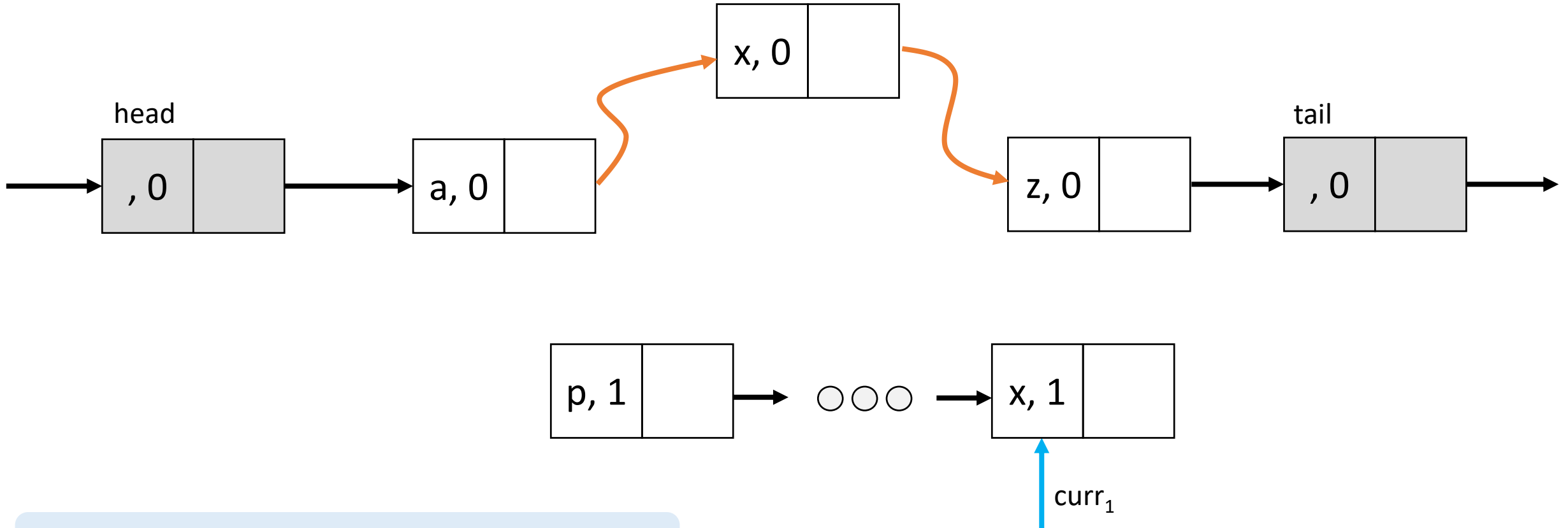
```
public boolean contains(T x) {  
    int key = x.hashCode();  
    Node curr = head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Detecting Conflicting Accesses: Example 1



- Thread 1 is executing `contains(x)`
- Thread 2 executes `remove(p..x)`

Detecting Conflicting Accesses: Example 2

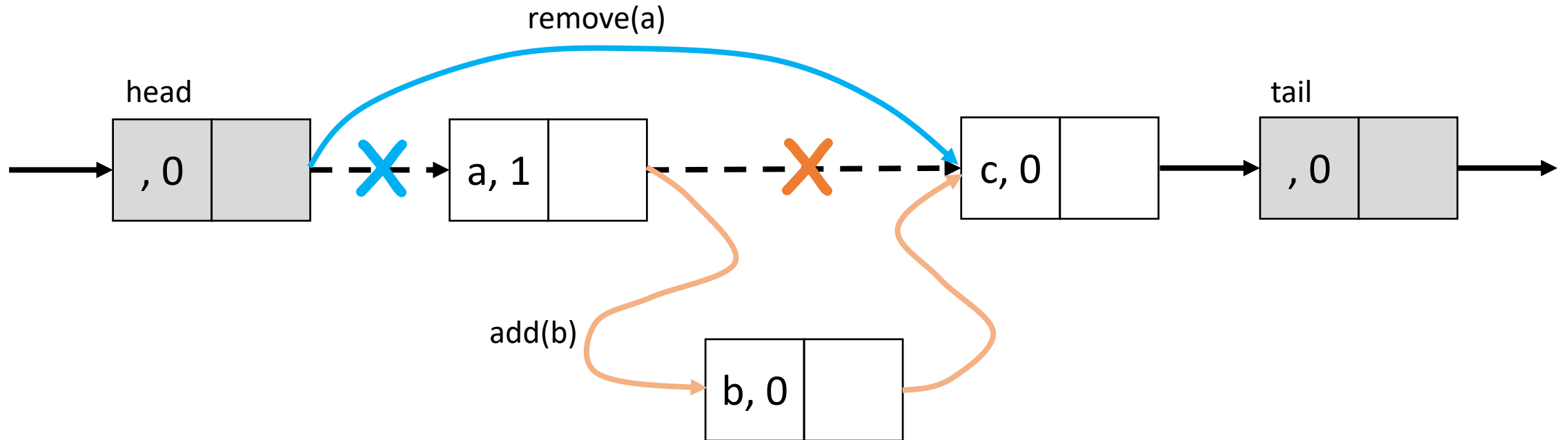


- Thread 1 is executing contains(x)
- Thread 2 is executing remove(p..x)

Nonblocking Synchronization

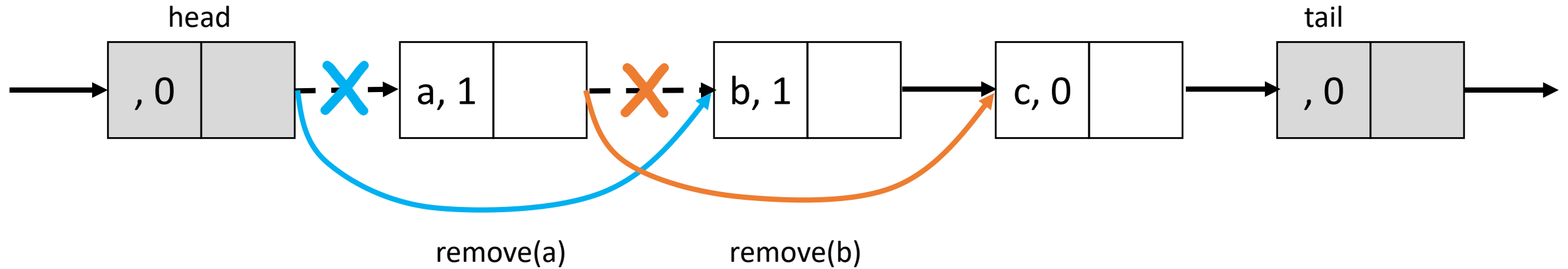
- Why do we need nonblocking designs?
- Eliminate locks altogether
- **Idea:** Use RMW instructions like CAS to update next field

Nonblocking Synchronization with CAS



- Thread 1 is executing `remove(a)`
- Thread 2 is executing `add(b)`

Nonblocking Synchronization with CAS

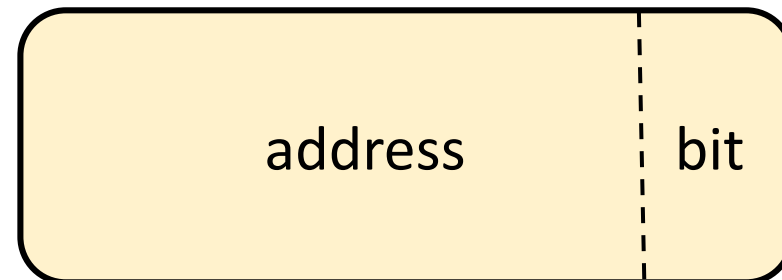


- Thread 1 is executing `remove(a)`
- Thread 2 is executing `remove(b)`

Possible Workaround

- Cannot allow updates to a node once it has been logically or physically removed from the list
- Treat the next and marked fields as atomic

In Java, we have `AtomicMarkableReference<T>` from the `java.util.concurrent.atomic` package

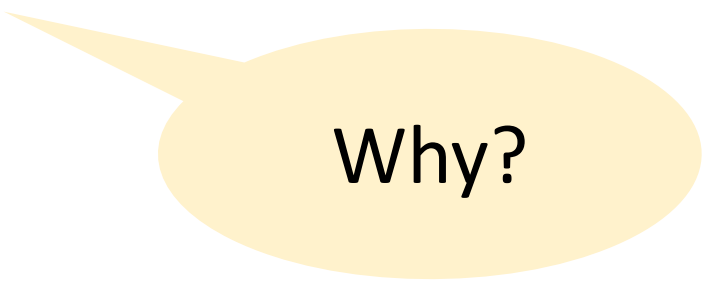


AtomicMarkableReference<T>

- `public boolean compareAndSet(T expectedReference,
T newReference,
boolean expectedMark,
boolean newMark);`
- `public boolean attemptMark(T expectedReference,
boolean newMark);`
- `public T get(boolean[] marked);`

Designing the Nonblocking Set

- The next field is of type `AtomicMarkableReference<Node>`
- A thread logically removes a node by setting the mark bit in the next field
- As threads traverse the list, they clean up the list by physically removing marked nodes
- Threads performing `add()` and `remove()` do not traverse marked nodes, they **remove them before** continuing



Why?

Helper Code

- Helper method `public Window find(Node head, int key)`
 - Traverses the list seeking to set `pred` to the node with the largest key less than `key`, and `curr` to the node with the least key greater than or equal to `key`

```
class Window {  
    public Node pred, curr;  
    Window(Node myPred, Node myCurr) {  
        pred = myPred; curr = myCurr;  
    }  
}
```

Helper Code

```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                snip = pred.next.compareAndSet(curr, succ, false,
false);
                if (!snip) continue retry;
                curr = succ;
                succ = curr.next.get(marked);
            }
        }
    }
```

```
        if (curr.key >= key)
            return new Window(pred,
curr);
        pred = curr;
        curr = succ;
    }
}
```


Nonblocking Synchronization: add()

```
public boolean add(T x) {  
    int key = x.hashCode();  
    while (true) {  
        Window w = find(head, key);  
        Node pred = w.pred, curr = w.curr;  
        if (curr.key == key) return false;  
        else {  
            Node node = new Node(x);  
            node.next = new AtomicMarkableReference(curr, false);  
            if (pred.next.compareAndSet(curr, node, false, false))  
                return true;  
        }  
    }  
}
```

Nonblocking Synchronization: remove()

```
public boolean remove(T x) {  
    int key = x.hashCode();  
    boolean snip;  
    while (true) {  
        Window w = find(head, key);  
        Node pred = w.pred, curr = w.curr;  
        if (curr.key != key) return false;  
        else {  
            Node succ = curr.next.getReference();  
            snip = curr.next.attemptMark(succ, true);  
            if (!snip) continue;  
            pred.next.compareAndSet(curr, succ, false, false);  
            return true;  
        }  
    }  
}
```

Nonblocking Synchronization: contains()

```
public boolean contains(T x) {  
    boolean[] marked = new boolean[];  
    int key = x.hashCode();  
    Node curr = head;  
    while (curr.key < key) {  
        curr = curr.next;  
        Node succ = curr.next.get(marked);  
    }  
    return curr.key == key && !marked[0];  
}
```

Pool Data Structure

Pools

Allows duplicates

May not support membership test (i.e., no contains() method)

Example: stack, queue, bounded/unbounded buffers

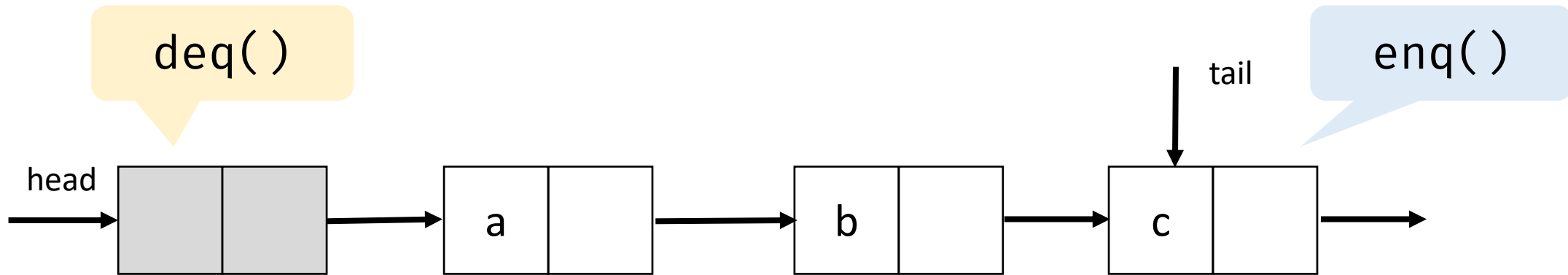
Data Structure Variants

- Bounded vs Unbounded
 - Different requirements and implementation challenges

```
public interface Pool<T> {  
    void put(T item);  
    T get();  
}
```

- Different method call invocation semantics
 - Blocking vs nonblocking
 - Synchronous vs asynchronous
 - Total vs partial

Bounded Partial Queue



Enqueue and dequeue operations are at the two ends
– allows for concurrent modifications

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

- Lock for mutual exclusion of enqueues and dequeues?

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use:

- ReentrantLock

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use:

- ReentrantLock
- Condition

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues
- Condition variable to indicate queue is empty
- Condition variable to indicate queue is full

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use:

- ReentrantLock
- Condition
- AtomicInteger

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues
- Condition variable to indicate queue is empty
- Condition variable to indicate queue is full
- An atomic variable to track the current size

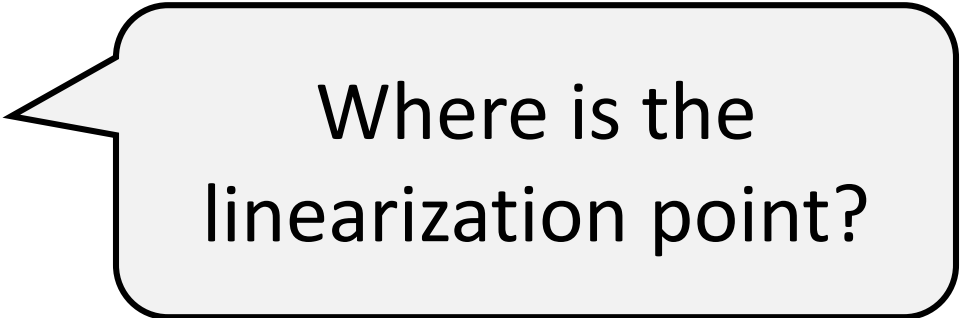
Bounded Partial Queue: enq()

```
public void enq(T x) {  
    boolean wakeDeq = false;  
    enqLock.lock();  
    try {  
        while (size.get() == MAX_CAP)  
            notFull.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;  
        if (size.getAndIncrement() == 0)  
            wakeDeq = true;  
    } finally {  
        enqLock.unlock();  
    }  
  
    if (wakeDeq) {  
        deqLock.lock();  
        try {  
            notEmpty.signalAll();  
        } finally {  
            deqLock.unlock();  
        }  
    } // end if (wakeDeq)  
} // end enq()
```

Bounded Partial Queue: enq()

```
public void enq(T x) {  
    boolean wakeDeq = false;  
    enqLock.lock();  
    try {  
        while (size.get() == MAX_CAP)  
            notFull.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;  
        if (size.getAndIncrement() == 0)  
            wakeDeq = true;  
    } finally {  
        enqLock.unlock();  
    }  
}
```

```
    if (wakeDeq) {  
        deqLock.lock();  
        try {  
            notEmpty.signalAll();  
        } finally {  
            deqLock.unlock();  
        }  
    } // end if (wakeDeq)  
} // end enq()
```



Where is the
linearization point?

Bounded Partial Queue: enq()

```
public void enq(T x) {
    boolean wakeDeq = false;
    enqLock.lock();
    try {
        while (size.get() == MAX_CAP)
            notFull.await();
        Node e = new Node(x);
        tail.next = e;
        tail = e;
        if (size.getAndIncrement() == 0)
            wakeDeq = true;
    } finally {
        enqLock.unlock();
    }
}
```

```
if (wakeDeq) {
    deqLock.lock();
    try {
        notEmpty.signalAll();
    } finally {
        deqLock.unlock();
    }
} // end if (wakeDeq)
} // end enq()
```

What if the queue was unbounded and the methods are total?

Bounded Partial Queue: deq()

```
public void deq() {
    boolean wakeEnq = false;
    T result;
    deqLock.lock();
    try {
        while (size.get() == 0)
            notEmpty.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndDecrement() == MAX_CAP)
            wakeEnq = true;
    } finally {
        deqLock.unlock();
    }

    if (wakeEnq) {
        enqLock.lock();
        try {
            notFull.signalAll();
        } finally {
            enqLock.unlock();
        }
    }
}
```

Evaluating the Bounded Partial Queue

- Need to ensure correct interleaving of concurrent calls to `enq()` and `deq()`
 - Special cases: Queue has zero or one element
- Shared updates to the `size` variable could be a bottleneck
 - Can we do something about it?

Unbounded Total Queue

- `enq()` always enqueues an item
 - It may run in to OOM error which we will ignore
- `deq()` returns an error if the queue is empty

Unbounded Total Queue

```
public void enq(T x) {
    enqLock.lock();
    try {
        Node e = new Node(x);
        tail.next = e;
        tail = e;
    } finally {
        enqLock.unlock();
    }
}
```

```
public T deq() {
    T result;
    deqLock.lock();
    try {
        if (head.next == null)
            return null;
        result = head.next.value;
        head = head.next;
    } finally {
        deqLock.unlock();
    }
    return result;
}
```

A Natural Next Step!

- Unbounded lock-free queue

Possible Java classes we can use:

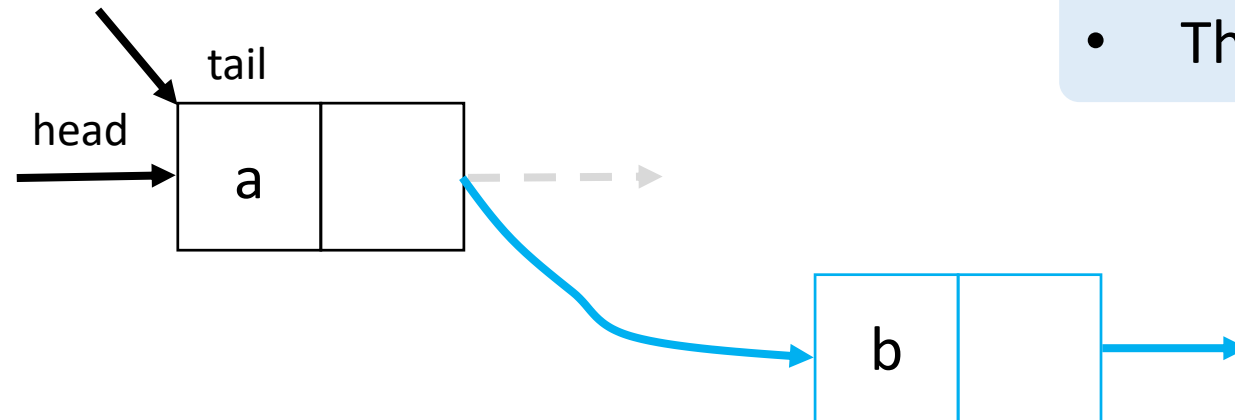
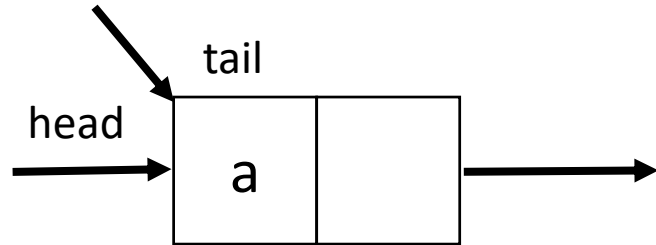
- `AtomicReference<T>`

Unbounded Lockfree Queue: enq()

```
public void enq(T x) {  
    Node node = new Node(x);  
    while (true) {  
        Node last = tail.get();  
        Node next = last.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next,  
node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                }  
            }  
        }  
        } else {  
            tail.compareAndSet(last, next);  
        }  
    } // end if (last == ...  
} // end while (true)  
} // end enq()
```

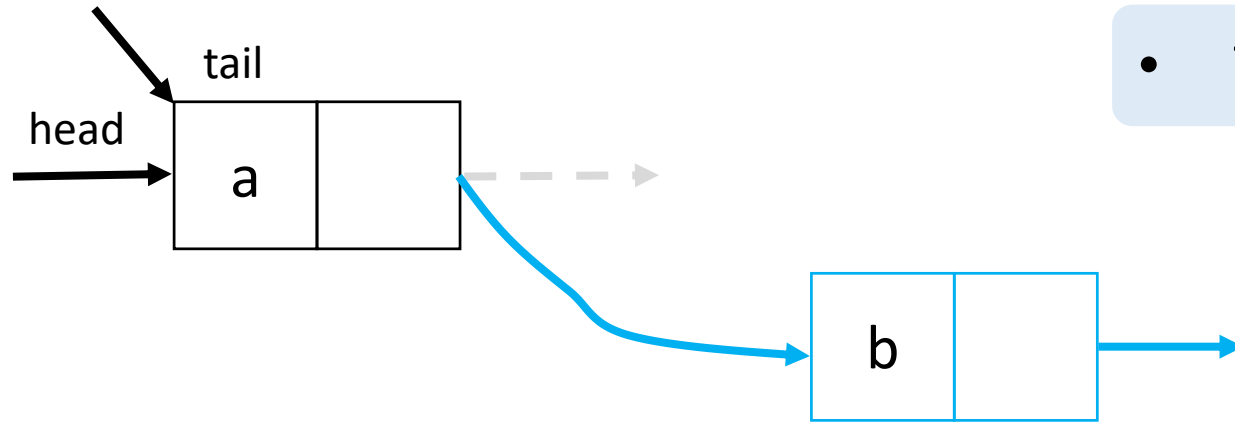
Where is the
linearization point?

Ensure that `tail` remains valid!

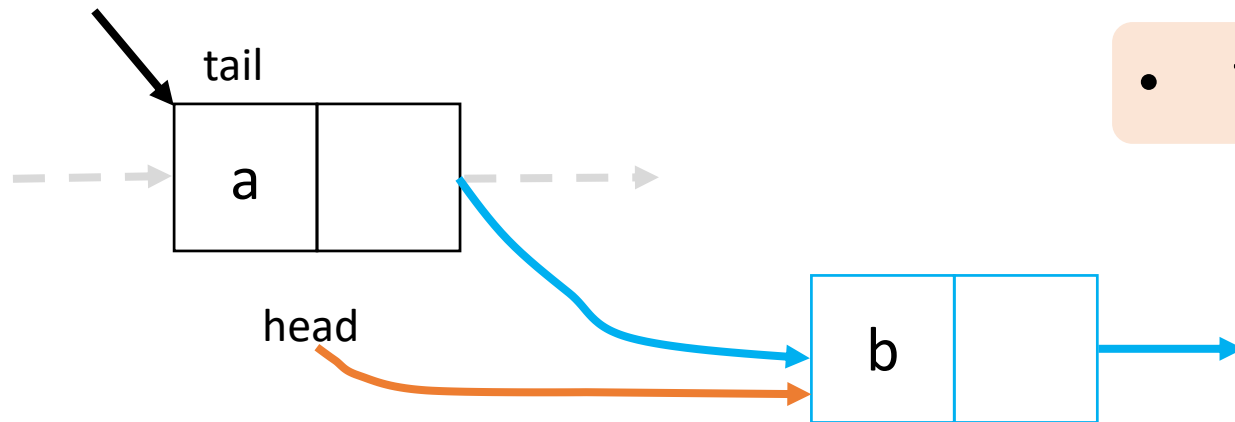


- Thread 1 is executing `enq(b)`

Ensure that `tail` remains valid!



- Thread 1 is executing `enq(b)`

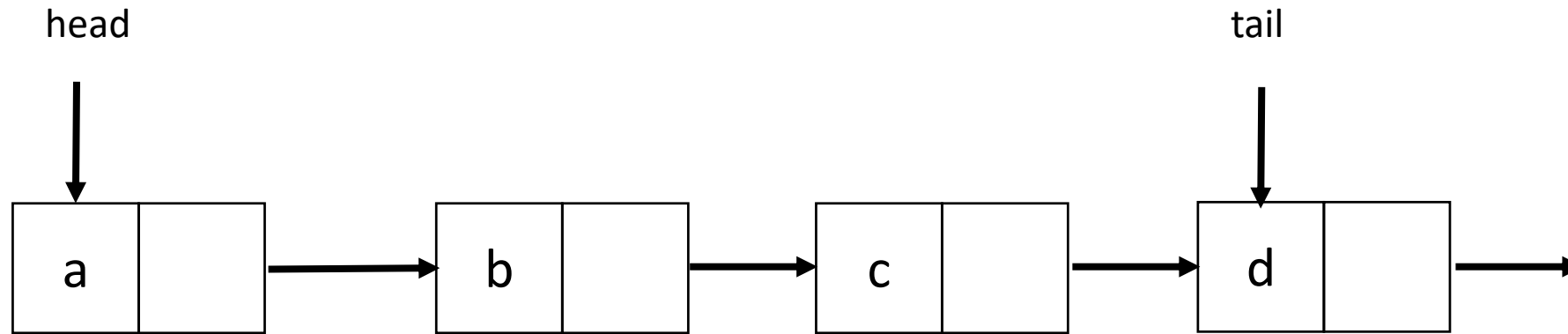


- Thread 2 is executing `deq(a)`

Unbounded Lockfree Queue: deq()

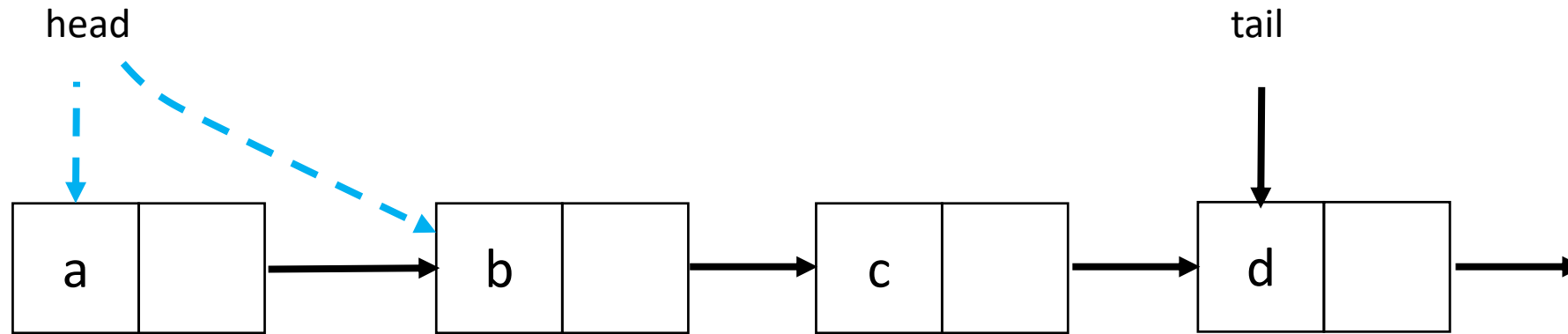
```
public void deq(void) {
    while (true) {
        Node first = head.get();
        Node last = tail.get();
        Node next = first.next.get();
        if (first == head.get()) {
            if (first == last) {
                if (next == null)
                    return null;
                tail.compareAndSet(last, next);
            } else {
                T val = next.value;
                if (head.compareAndSet(first, next))
                    return val;
            } // end if (first == head...)
        } // end while (true)
    } // end deq()
}
```

Lock-free Programming and ABA Problem



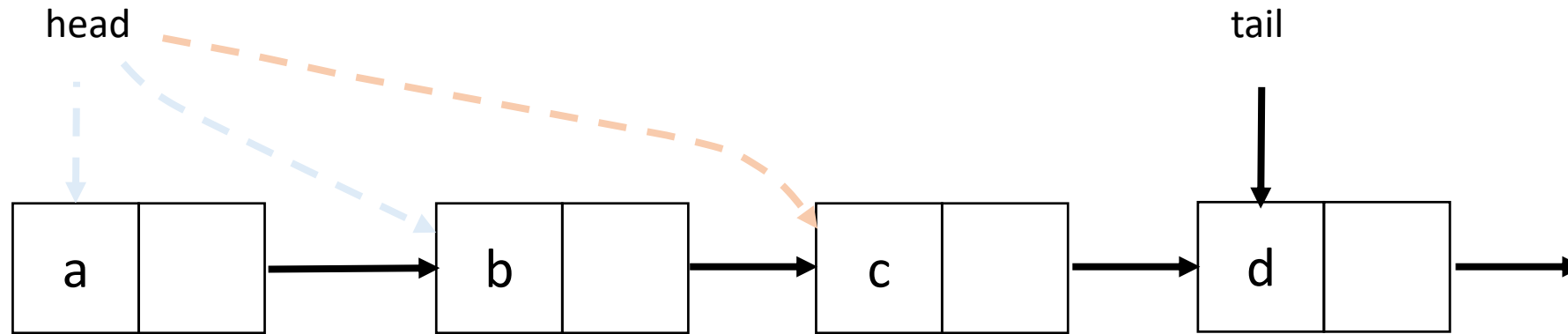
- Thread 1 will execute `deq(a)`

Lock-free Programming and ABA Problem



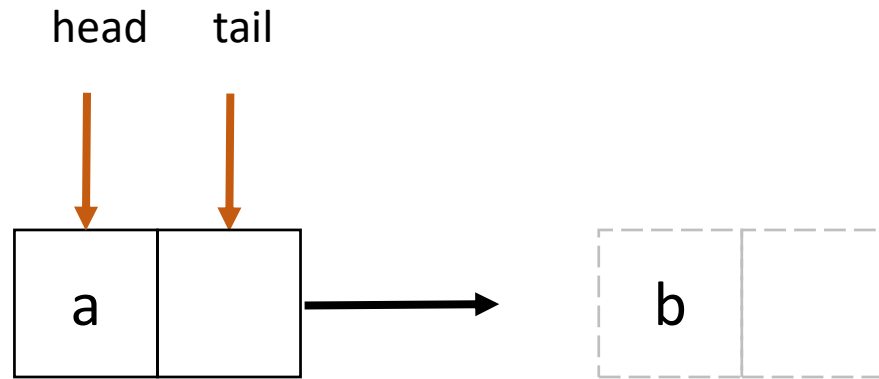
- Thread 1 is executing `deq(a)`, gets delayed

Lock-free Programming and ABA Problem



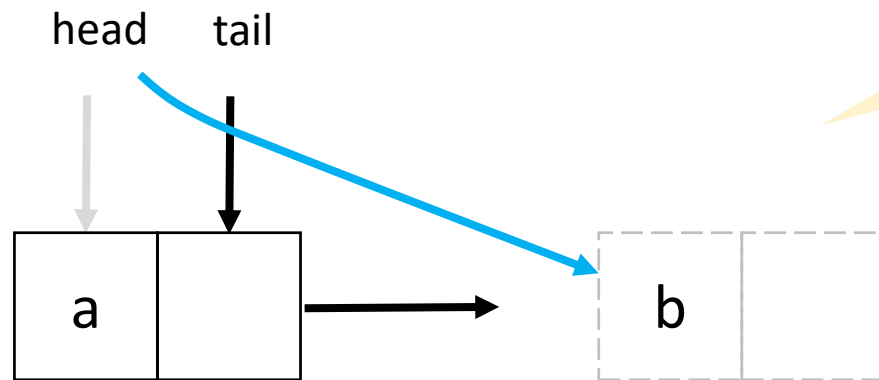
- Other threads execute `deq(a, b, c, d)`, then execute `enq(a)`

Lock-free Programming and ABA Problem



- Other threads execute `deq(a, b, c, d)`, then execute `enq(a)`

Lock-free Programming and ABA Problem



- Thread 1 is executes CAS for `deq(a)`, CAS succeeds

To Lock or Not to Lock!

Use a middle path more often than not

- Combine blocking and nonblocking schemes
- For e.g., lazily synchronized Set
 - `add()` and `remove()` were blocking
 - `contains()` was nonblocking

Please spend several hours reasoning about the correctness of your concurrent data structures, if you are writing one!

References

- M. Herlihy and N. Shavit – The Art of Multiprocessor Programming.
- M. Moir and N. Shavit – Concurrent Data Structures.
- Stephen Tu – Techniques for Implementing Concurrent Data Structures on Modern Multicore Machines.