

# CS698L: Concurrent Data Structures

Swarnendu Biswas

Semester 2019-2020-I  
CSE, IIT Kanpur

---

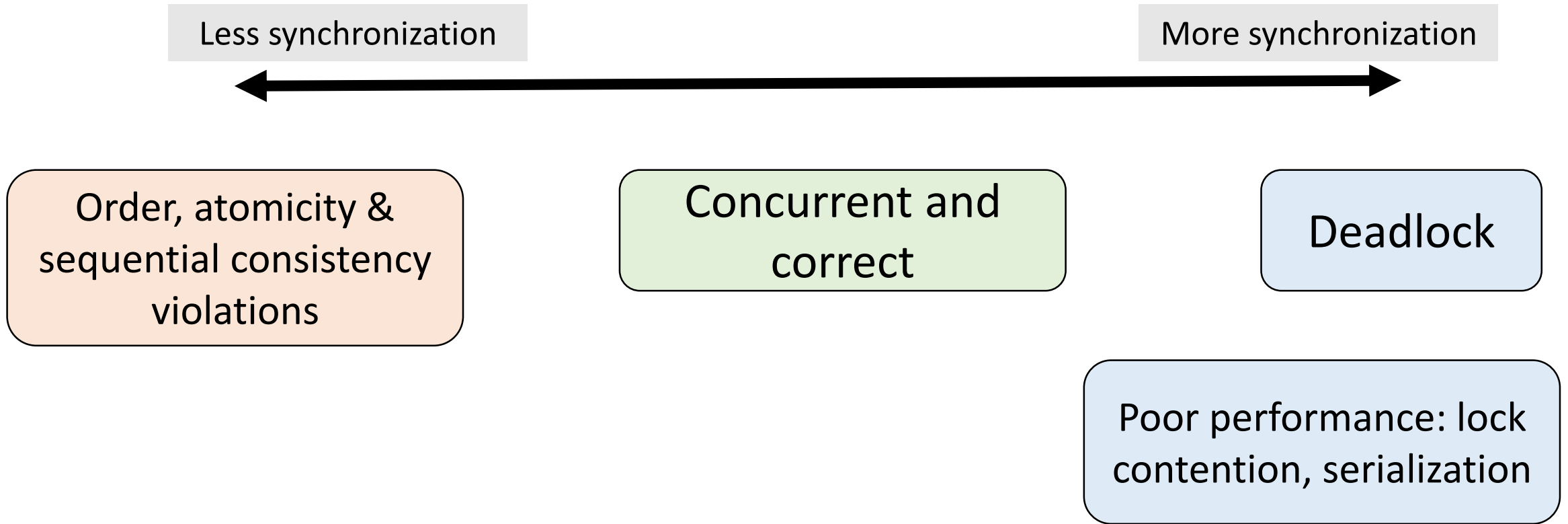
Content influenced by many excellent references, see References slide for acknowledgements.

# 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

Less synchronization

More synchronization

Implies that languages and libraries should provide efficient portable data structures as building blocks

Order  
sequent  
vi

ck

Poor performance: lock  
contention, serialization

# 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)**

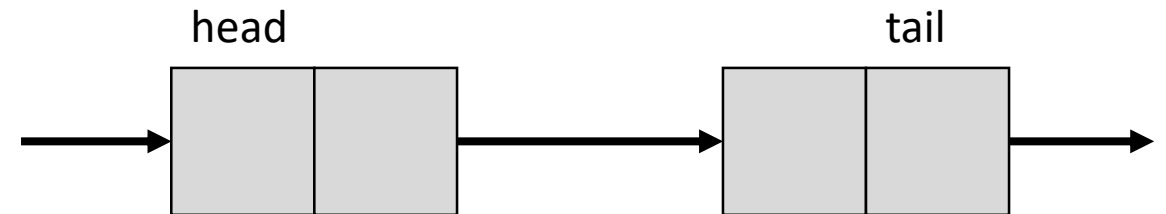
- returns true if and only if x is present in the set

# Designing A Set Data Structure using Linked Lists

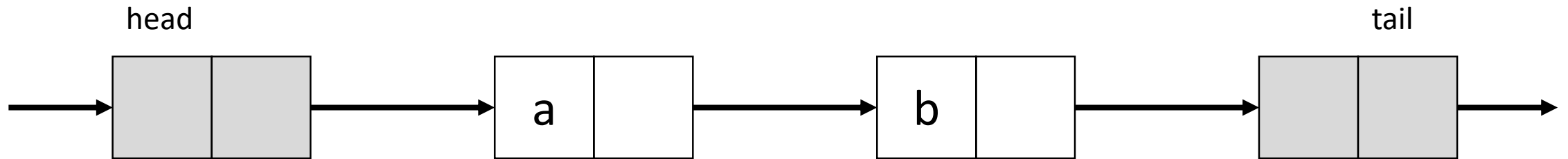
```
class Node {  
    T data;  
    int key;  
    Node next;  
}
```

- key field is the data's hash code, to help with efficient search
- Assume that all hash codes are unique

- Two immutable 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()

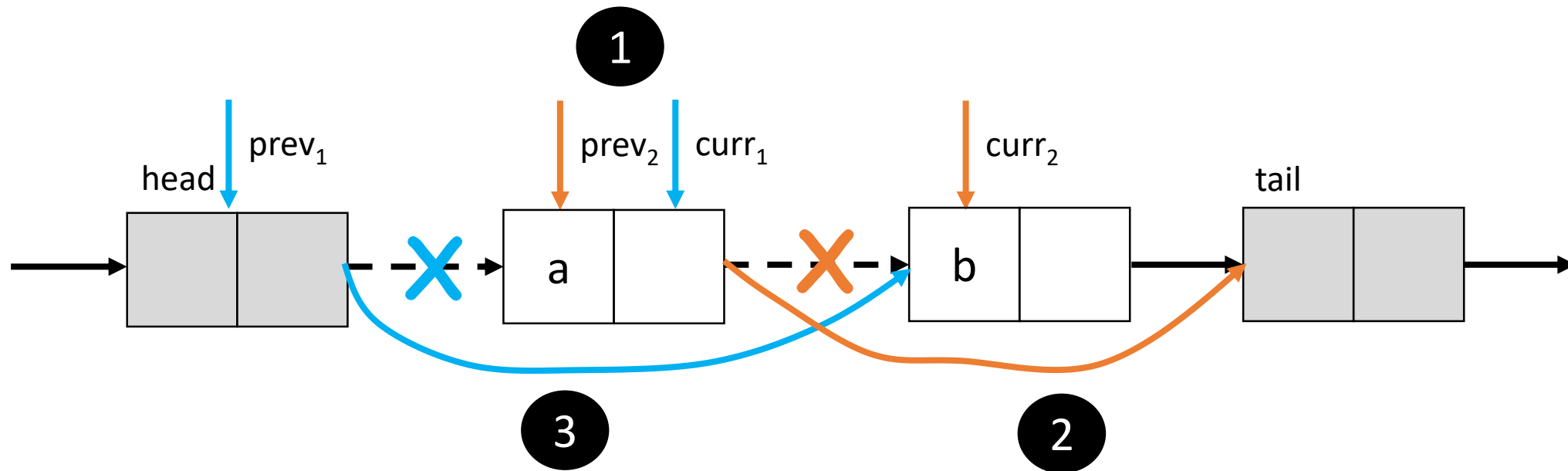
```
public boolean contains(T x) {
    Node pred, curr;
    int key = x.hashCode();
    pred = head;
    curr = pred.next;
    while (curr.key < key) {
        pred = curr;
        curr = curr.next;
    }
    if (key == curr.key) {
        return true;
    } else {
        return false;
    }
}
```

# 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

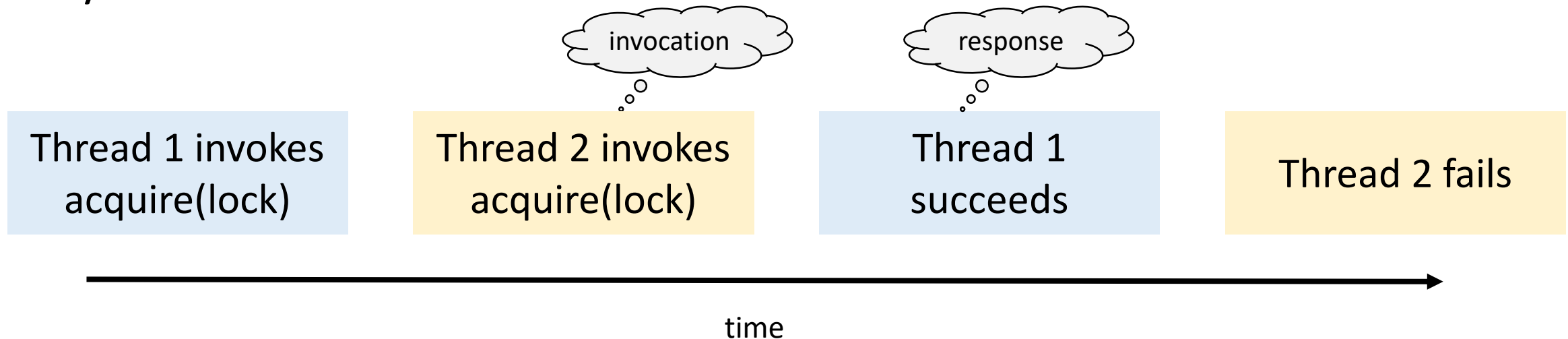
- Multiple threads can access a shared object
  - E.g., a node in our Set data structure
- Situation:
  - Thread 1 is checking for `contains(a)`
  - Thread 2 is executing `remove(a)`
- How do you reason about the outcome?

# 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

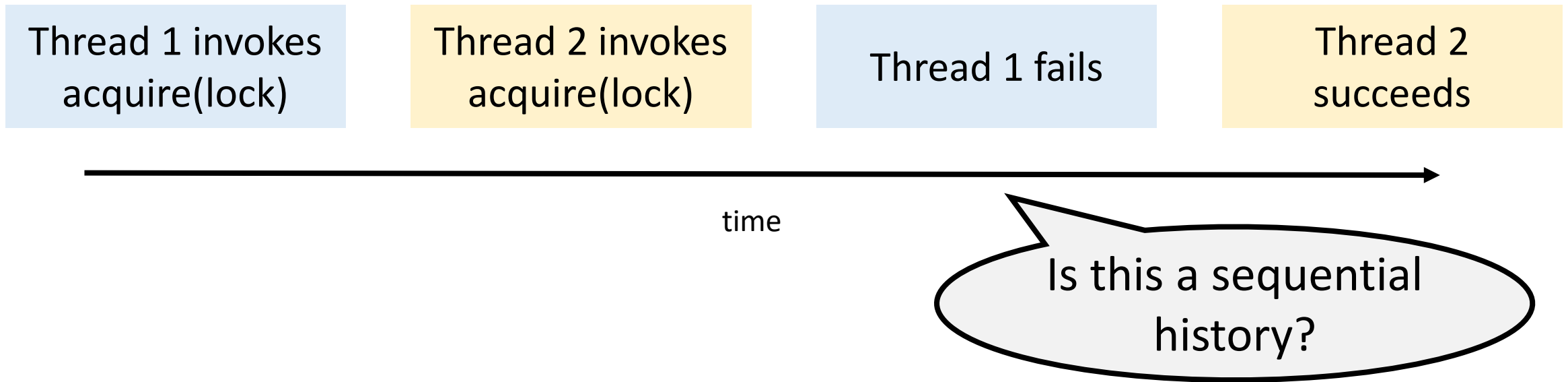
# Understanding Linearizability

- Say you perform some operations on an object (for e.g., a method call)
  - 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



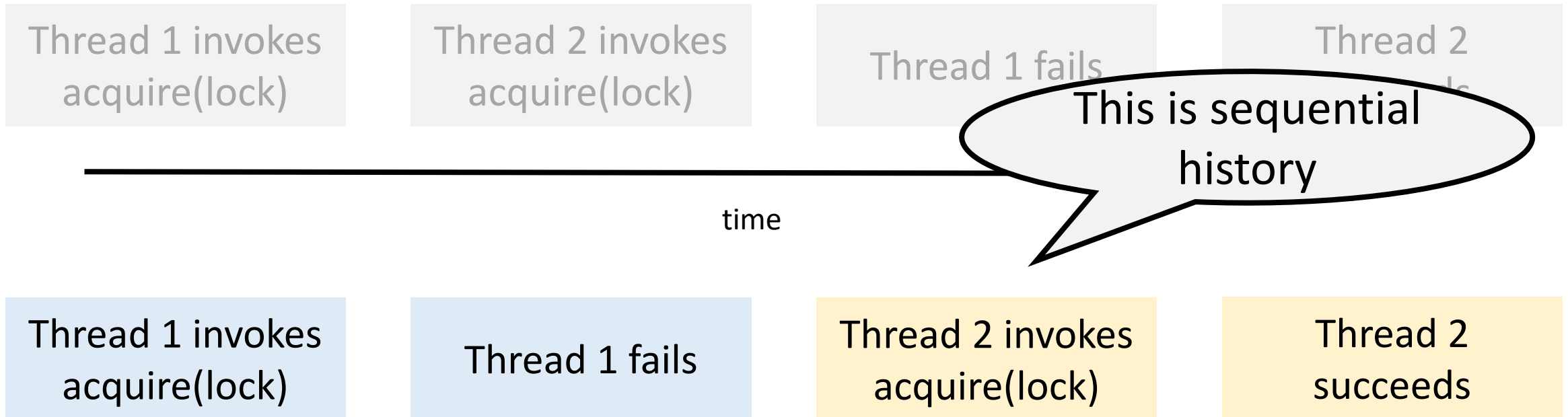
# Understanding Linearizability

- **Sequential history** is where all invocations and responses are instantaneous
  - Starts with an invocation, last invocation may not have a response
  - Method calls do not overlap



# Understanding Linearizability

- Sequential history is where all invocations and responses are instantaneous



# Linearizability

- A history (set of operations)  $\sigma$  is **linearizable** if
  - For every completed operation in  $\sigma$ , the operation returns the same result in the execution as it would return if every operation in  $\sigma$  would have been completed one after the other
  - If an operation  $op_1$  completes before operation  $op_2$  in sequential history, then  $op_1$  precedes  $op_2$  in  $\sigma$

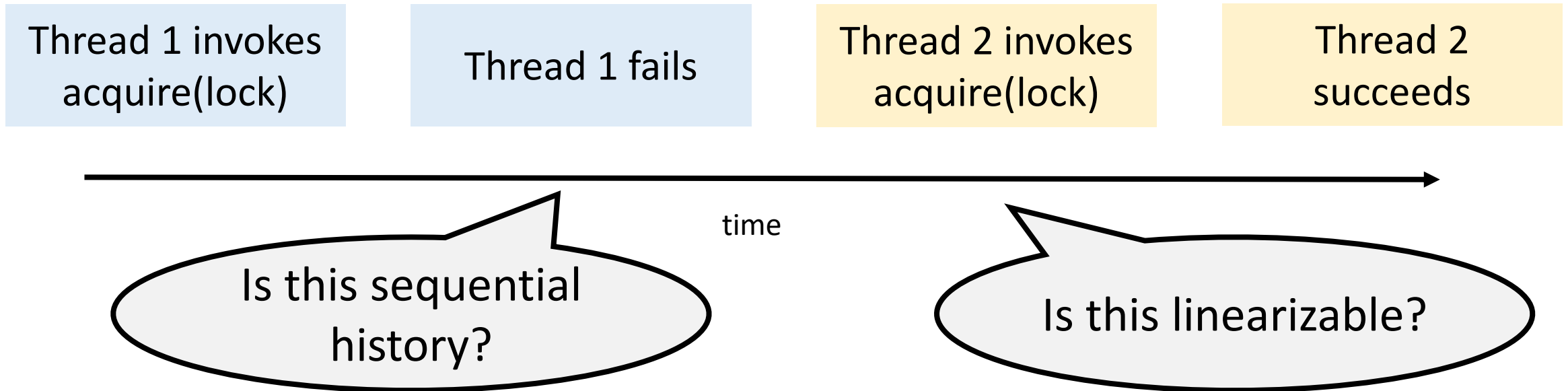


# Linearizability

- A history (set of operations)  $\sigma$  is **linearizable** if
  - For every completed operation in  $\sigma$ , the operation returns the same result in the execution as it would return if every operation in  $\sigma$  would have been completed one after the other
  - If an operation  $op_1$  completes before operation  $op_2$ , then  $op_1$  precedes  $op_2$  in  $\sigma$ .
- **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

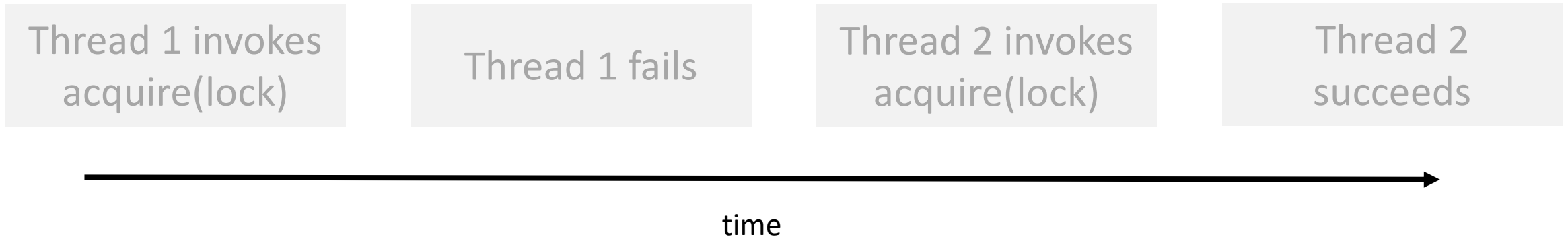
# Understanding Linearizability

- 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?

# Sequential Consistency vs Linearizability

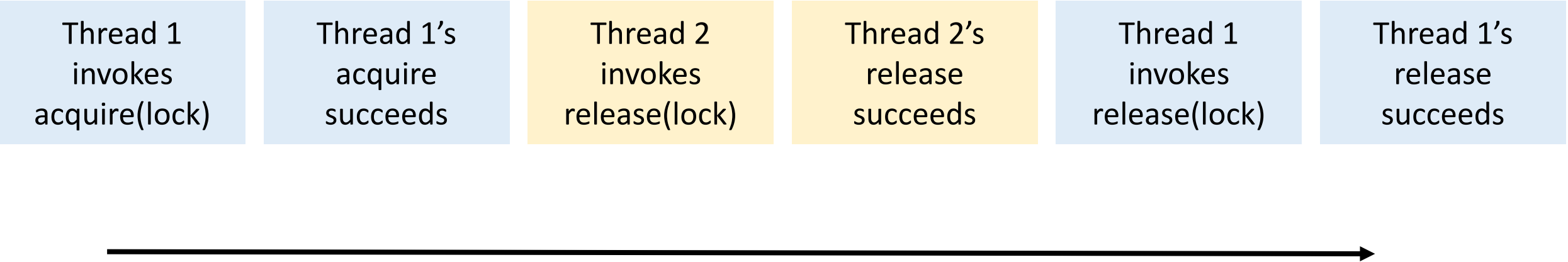
## Sequential Consistency

- Method calls appear to happen instantaneously in some sequential order
- A sequentially consistent history is not necessarily linearizable

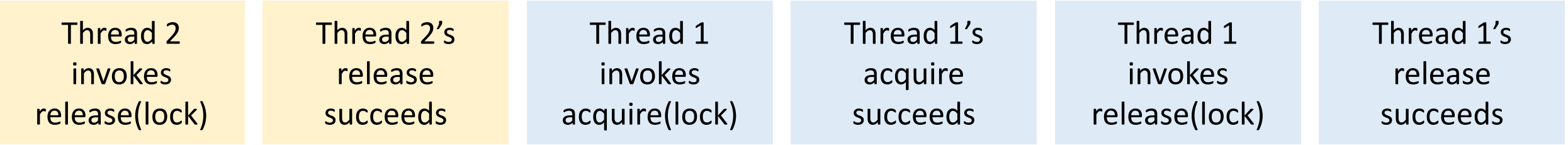
## Linearizability

- Method calls appear to happen instantaneously at some point between its invocation and response
- Every linearizable history is sequentially consistent

# Linearizability vs Serializability



Not linearizable



Serializable

# 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



# Ideas in Implementing a Concurrent Data Structure

## Coarse-grained synchronization

- Easy to get right, low concurrency, not scalable

## Fine-grained synchronization

- Difficult to get right, more concurrent and scalable

???

# Ideas in Implementing a Concurrent Data Structure

## Coarse-grained synchronization

- Easy to get right, low concurrency, not scalable

## Fine-grained synchronization

- Difficult to get right, more concurrent and scalable

## Optimistic synchronization

- Avoid synchronization to search, good for low contention cases

## Lazy synchronization

- Defer expensive data structure manipulation operations

## Nonblocking synchronization

# 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 possible ideas to implement `add()` and `remove()`?

# Is one lock per node enough?

## Thread 1

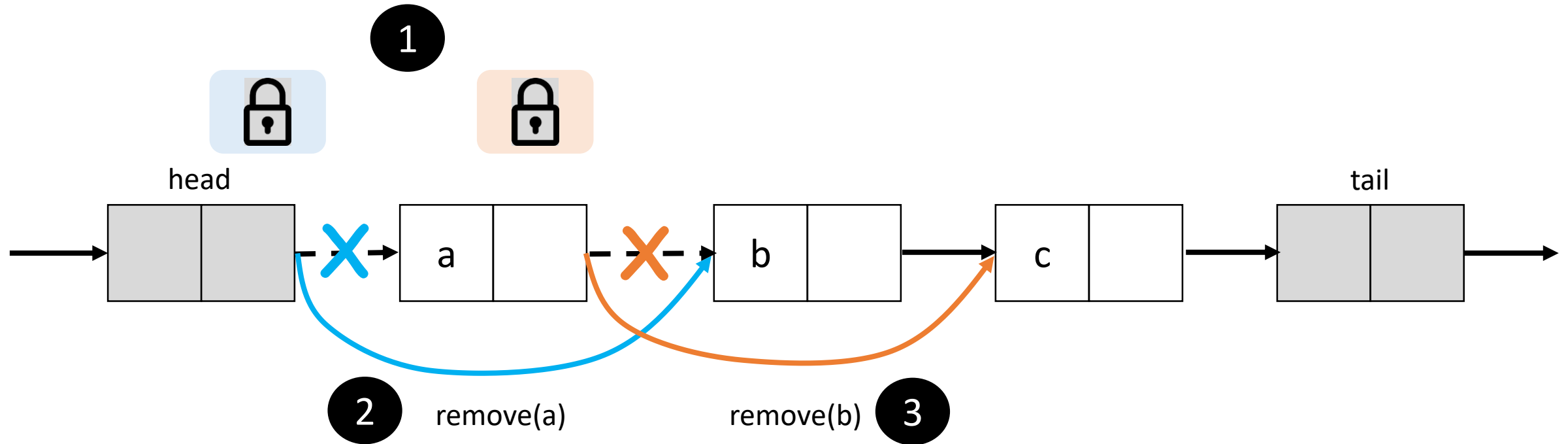
```
node0.mtx_lock.lock();  
node1 = node0.next;  
node0.mtx_lock.unlock();
```

## 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;
                prev.next = node;
                return true;
            }
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

# 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;
                prev.next = node;
            } finally {
                pred.unlock();
            }
        }
    }
}
```

Where is the linearization point?



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

Where is the linearization point?

- x is absent, predecessor node is locked
- x is present, next higher node is locked

# 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();
    }
}
```

# 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 {
            pred.unlock();
        }
    }
}
```

What is the linearization point?

# Fine-Grained Synchronization: remove()

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

Where is the linearization point?

- x is present, predecessor node is locked
- x is absent, next higher node is locked

# Need to avoid Deadlocks

- Deadlocks are always a problem with lock-based programming
- 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?

# Fine-Grained Set Design

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

- Potentially long sequence of lock acquire and release operations
- Prohibits concurrent accesses to disjoint parts of the data structure

# 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?

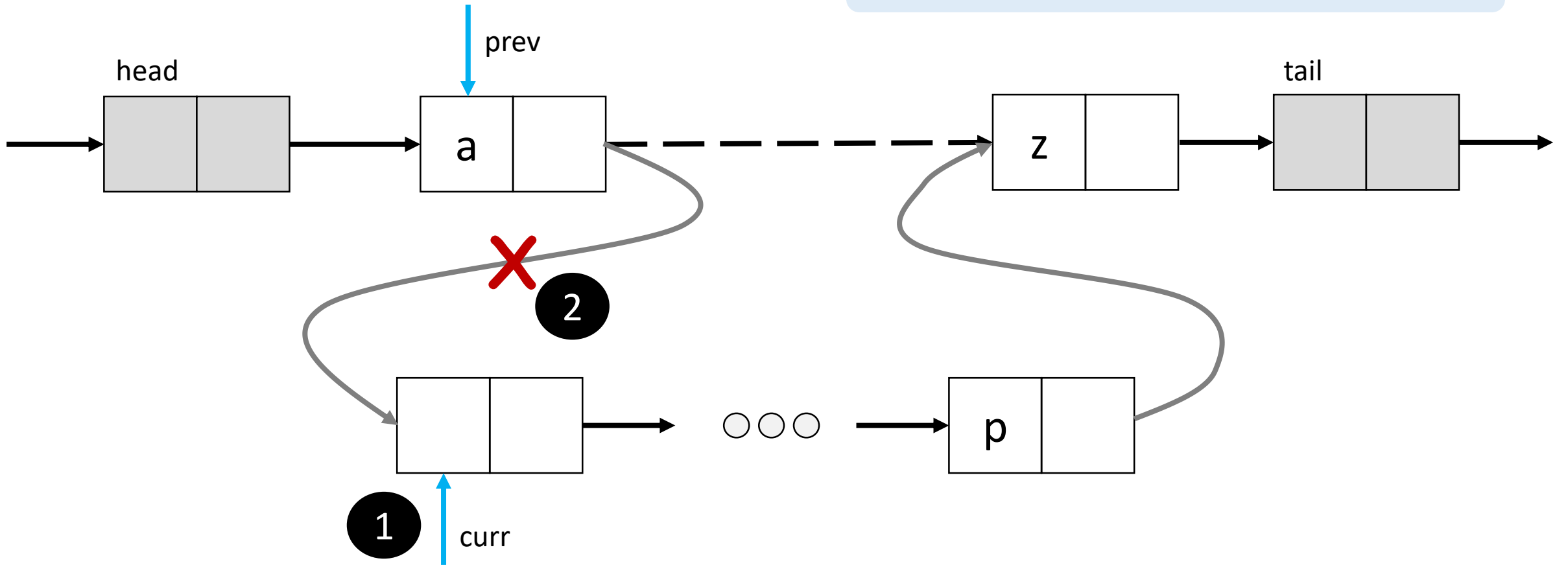
- Double check that the optimistic result is still valid
- Check that prev is reachable from head and  
`prev.next == curr`

```
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?

# Optimistic Synchronization Design

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

- Validation can be costly (for e.g., need to traverse the list)
- Need lock operations for `contains()`
  - Bad design in general



# Lazy Synchronization

Delay  
mutation  
operations for  
a later time

- Add a mark/flag on each node to indicate logical 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?

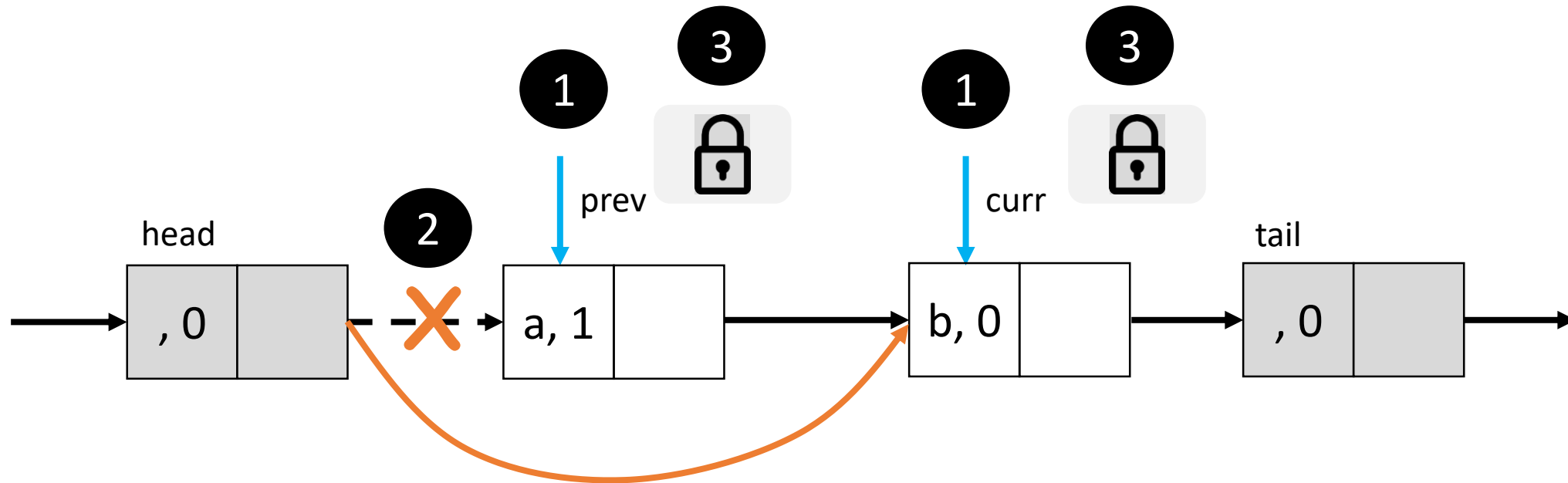
- Check that both prev and curr are unmarked and  
`prev.next == curr`

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

# Lazy Synchronization: remove()

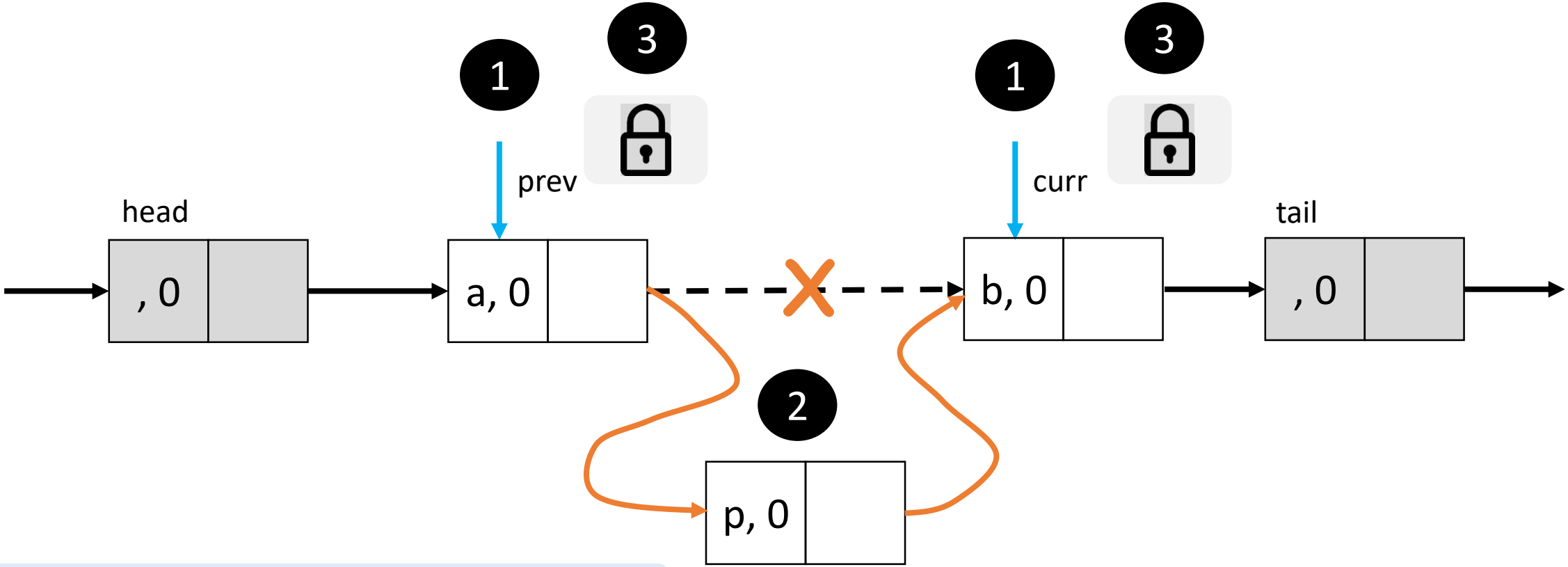
```
public boolean remove(T x) {
    int key = x.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {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

# Does this validation scheme work?



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

# Does this validation scheme work?



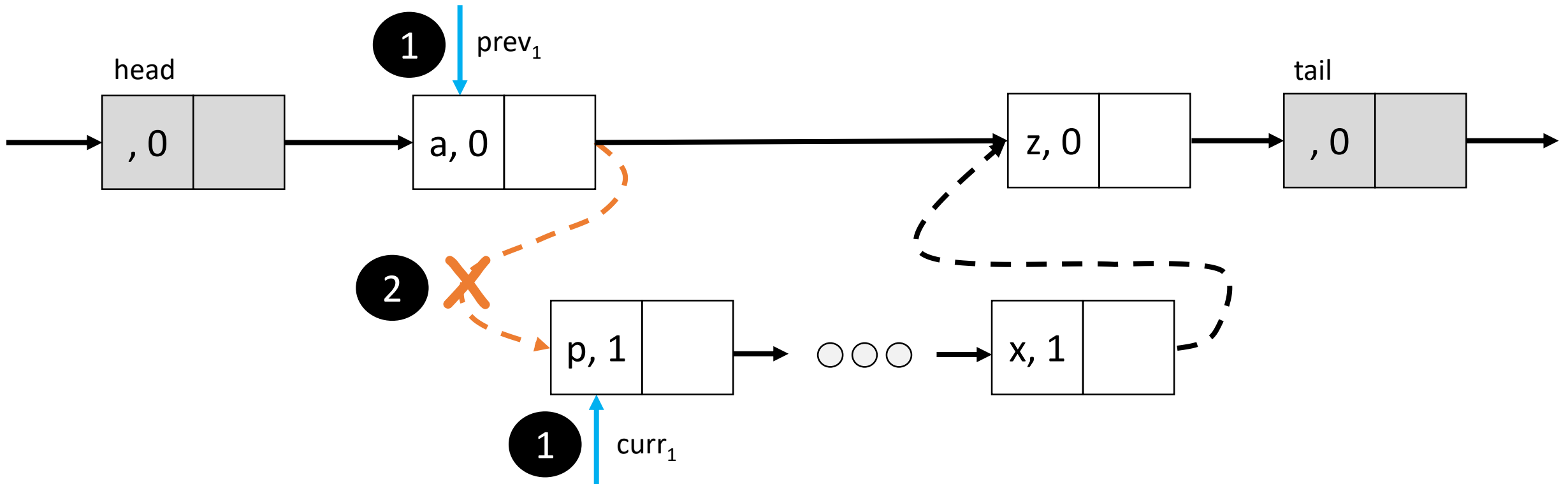
- Thread 1 is executing remove(b)

- Thread 2 is executing add(p)

# 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

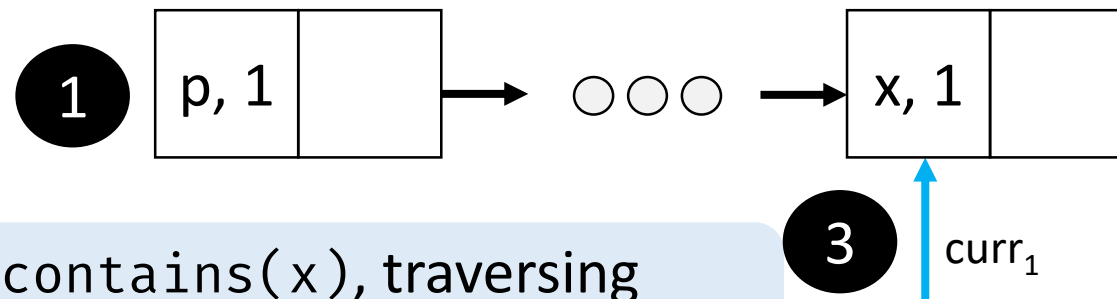
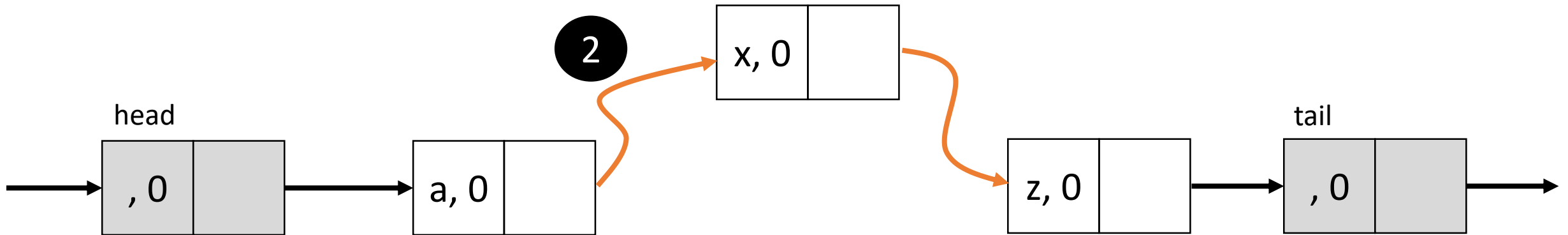


- Thread 1 is executing `contains(x)`

- Thread 2 executes `remove(p..x)`



# Detecting Conflicting Accesses: Example 2

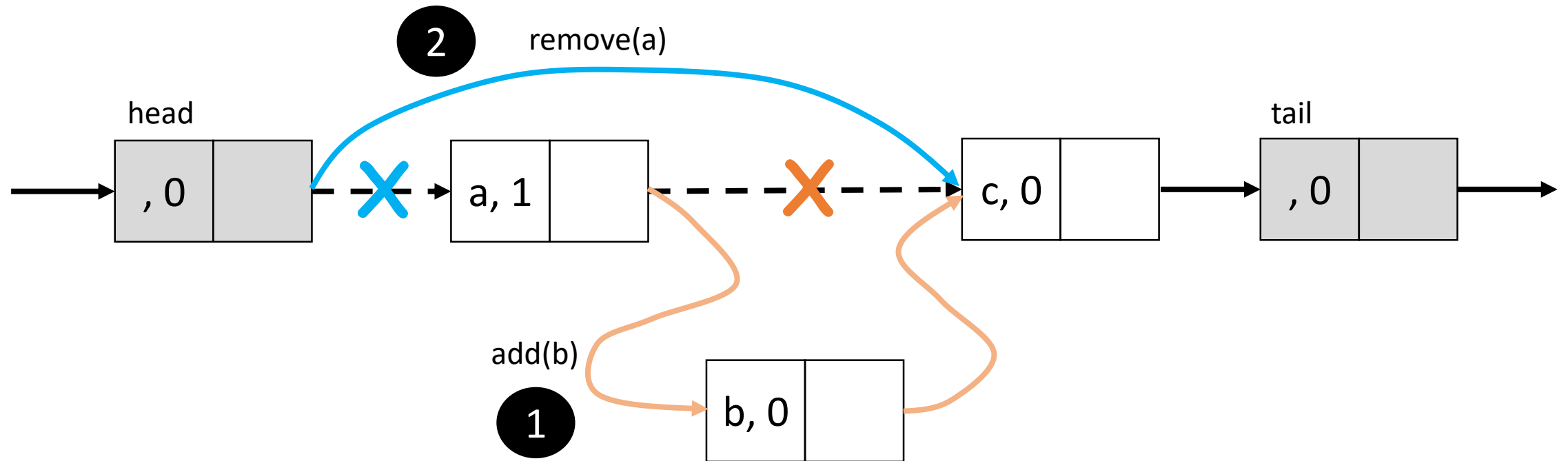


- Thread 1 is executing `contains(x)`, traversing along the marked portion of the list (`p...x`)
- Thread 2 is executing `add(x)`

# Nonblocking Synchronization

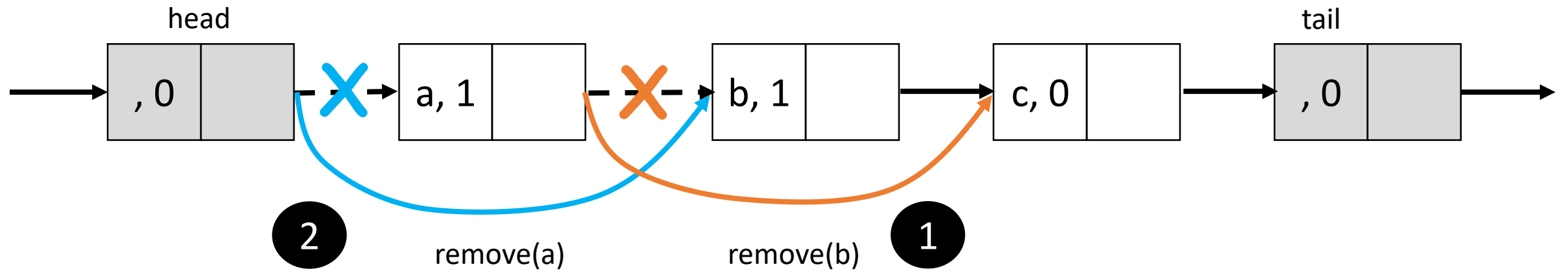
- Why do we need nonblocking designs?
- Eliminate locks altogether
- **Idea:** Use RMW instructions like CAS to update next field

# Nonblocking Synchronization with CAS



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

# Nonblocking Synchronization with CAS

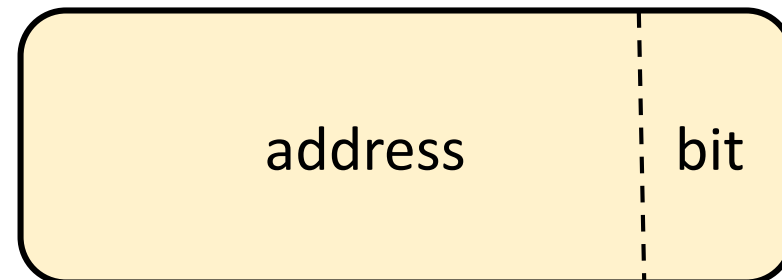


- Thread 1 is executing `remove(a)`
- Thread 2 is executing `remove(b)`

# 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

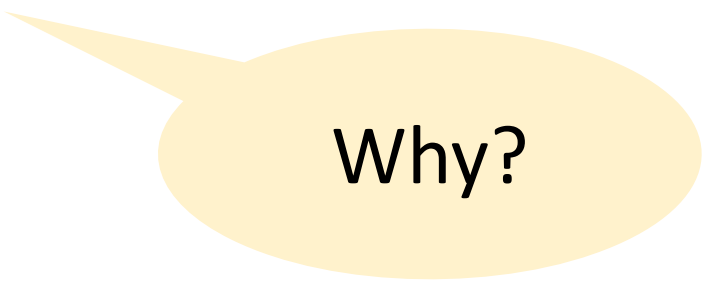


# AtomicMarkableReference<T>

- `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;
        }
    }
}
```

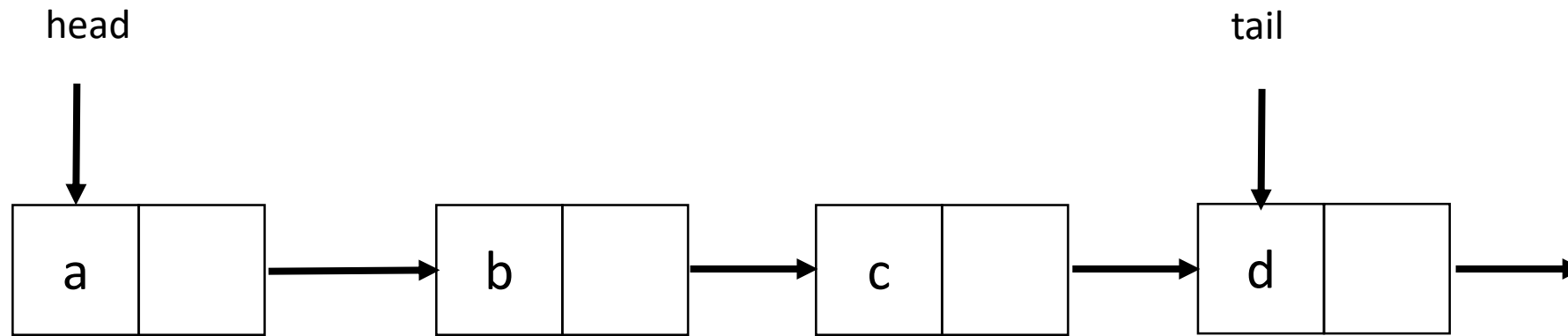
# Nonblocking Synchronization: remove()

```
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.compareAndSet(succ, succ, false, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}
```

# Nonblocking Synchronization: contains()

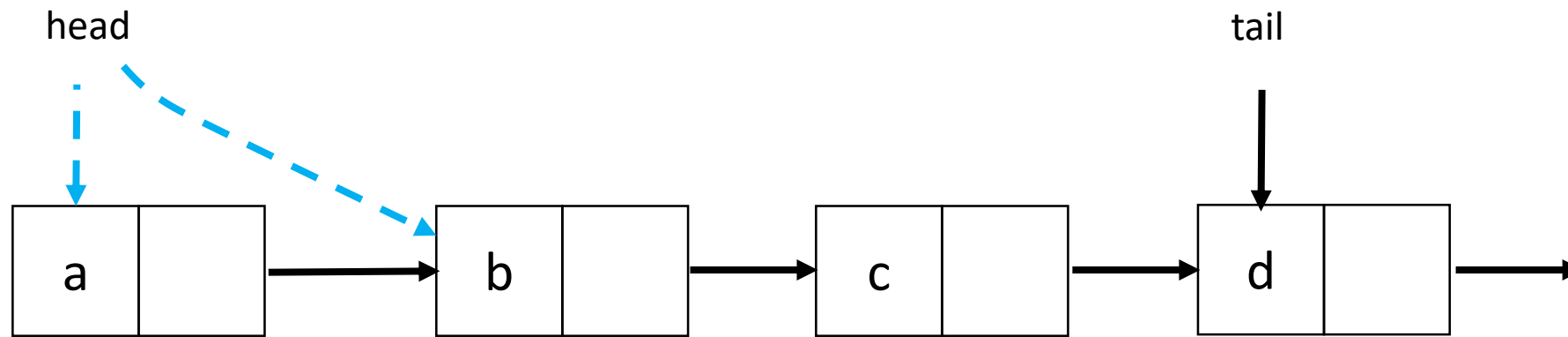
```
public boolean contains(T x) {
    boolean[] marked = new boolean[];
    int key = x.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
        Node succ = curr.next.get(marked);
    }
    return curr.key == key && !marked[0];
}
```

# Lock-free Programming and ABA Problem



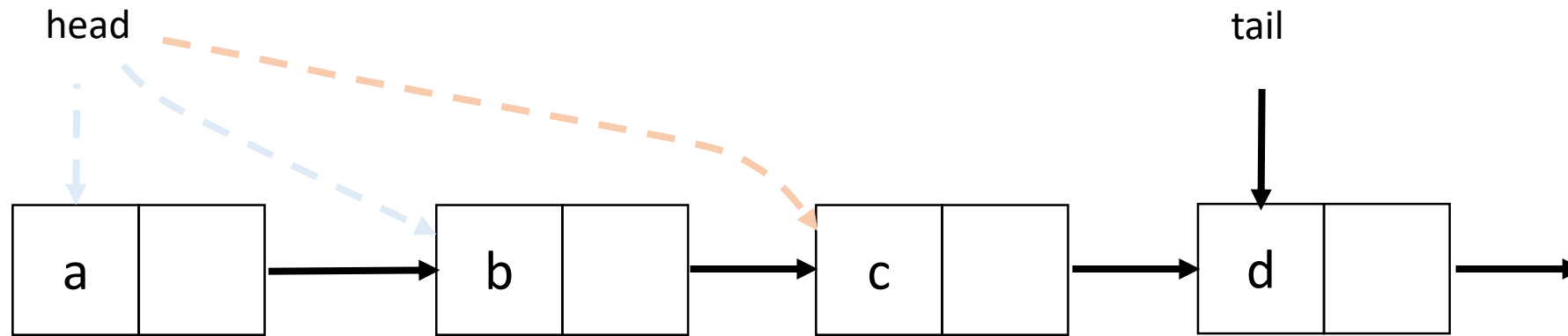
- Thread 1 will execute `deq(a)`

# Lock-free Programming and ABA Problem



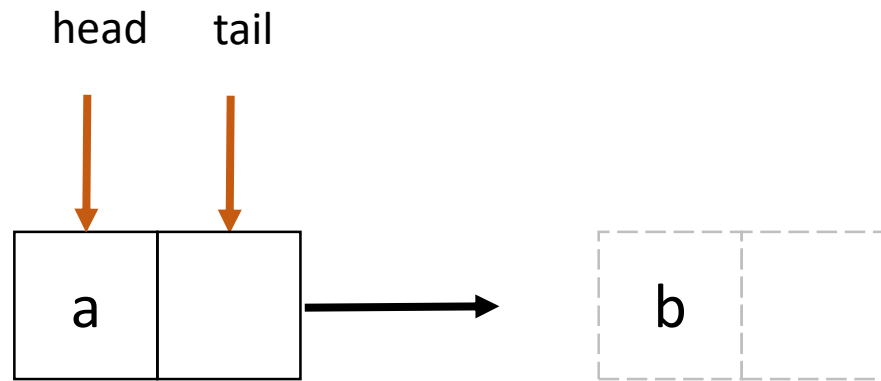
- Thread 1 is executing `deq(a)`, gets delayed

# Lock-free Programming and ABA Problem



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

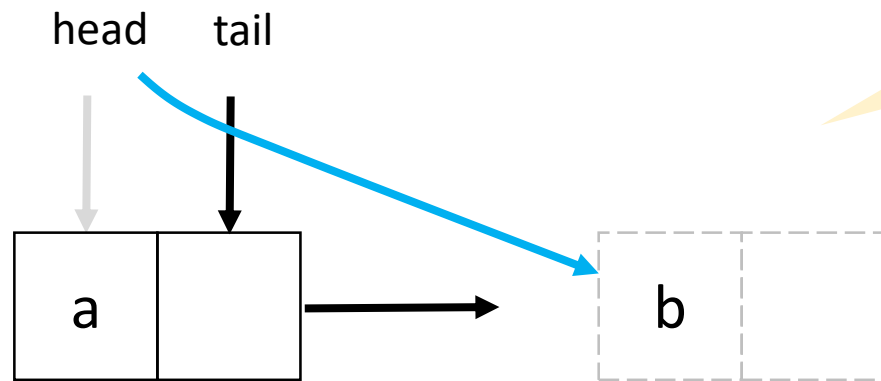
# Lock-free Programming and ABA Problem



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



# Lock-free Programming and ABA Problem



```
head.compareAndSet(first, next)
```

- 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.