

# *Concurrent Data Structures*



*Roger Wattenhofer*

# Overview

- Concurrent Linked List
  - Fine-grained synchronization
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- Hashing
  - Fine-grained locking
  - Recursive split ordering

# Handling Multiple Threads

- Adding threads should not **lower** the throughput
  - Contention effects can mostly be fixed by queue locks
- Adding threads should **increase** throughput
  - Not possible if the code is inherently sequential
  - Surprising things are parallelizable!
- How can we guarantee **consistency** if there are many threads?

# Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks
  - Mostly easy to reason about
  - This is the standard Java model (**synchronized** blocks and methods)
- Problem: Sequential bottleneck
  - Threads “stand in line”
  - Adding more threads does not improve throughput
  - We even struggle to keep it from getting worse...
- So why do we even use a multiprocessor?
  - Well, some applications are inherently parallel...
  - We focus on exploiting non-trivial parallelism

# Exploiting Parallelism

- We will now talk about four “patterns”
  - Bag of tricks ...
  - Methods that work more than once ...
- The goal of these patterns are
  - Allow concurrent access
  - If there are more threads, the throughput increases!

# Pattern #1: Fine-Grained Synchronization

- Instead of using a single lock split the concurrent object into **independently-synchronized** components
- Methods conflict when they access
  - The same component
  - At the same time

## Pattern #2: Optimistic Synchronization

- Assume that nobody else wants to access your part of the concurrent object
- Search for the specific part that you want to lock without locking any other part on the way
- If you find it, try to lock it and perform your operations
  - If you don't get the lock, start over!
- Advantage
  - Usually cheaper than always assuming that there may be a conflict due to a concurrent access

## Pattern #3: Lazy Synchronization

- Postpone hard work!
- Removing components is tricky
  - Either remove the object physically
  - Or logically: Only mark component to be deleted



# Pattern #4: Lock-Free Synchronization

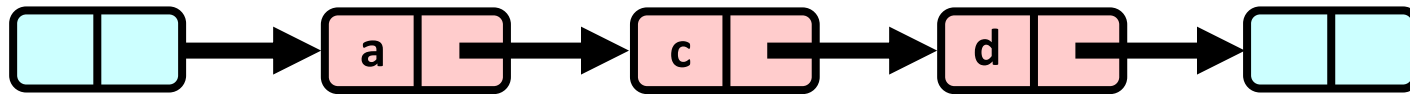
- Don't use locks at all!
  - Use `compareAndSet()` & other RMW operations!
- Advantages
  - No scheduler assumptions/support
- Disadvantages
  - Complex
  - Sometimes high overhead

# Illustration of Patterns

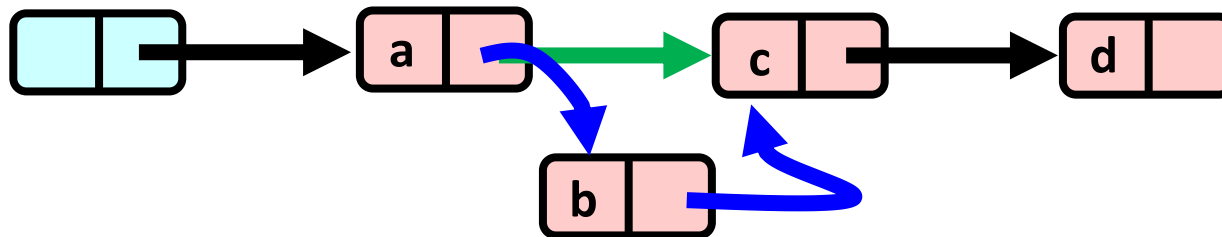
- In the following, we will illustrate these patterns using a **list-based set**
  - Common application
  - Building block for other apps
- A set is a collection of items
  - No duplicates
- The operations that we want to allow on the set are
  - **add(x)** puts **x** into the set
  - **remove(x)** takes **x** out of the set
  - **contains(x)** tests if **x** is in the set

# The List-Based Set

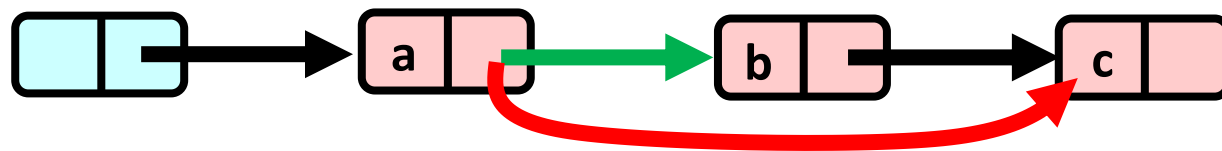
- We assume that there are sentinel nodes at the beginning (head) and end (tail) of the linked list



- Add node b:

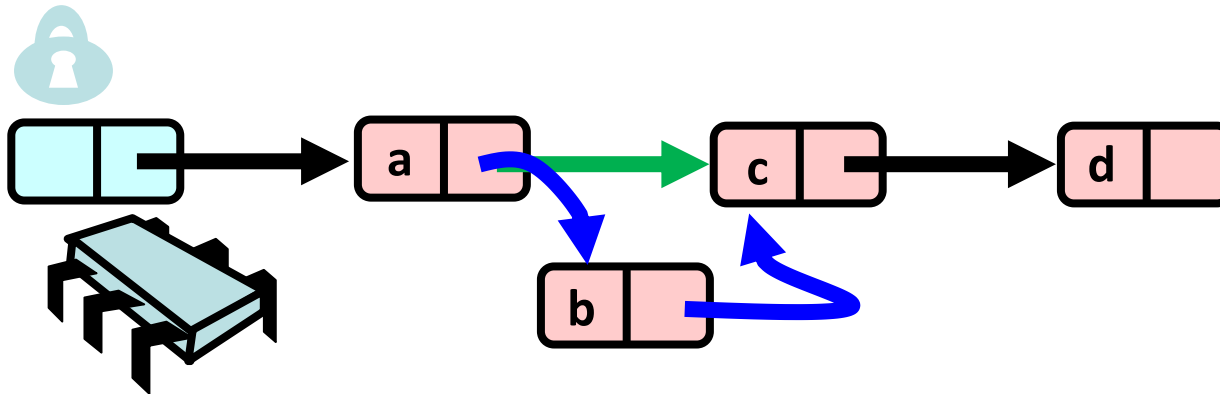


- Remove node b:



# Coarse-Grained Locking

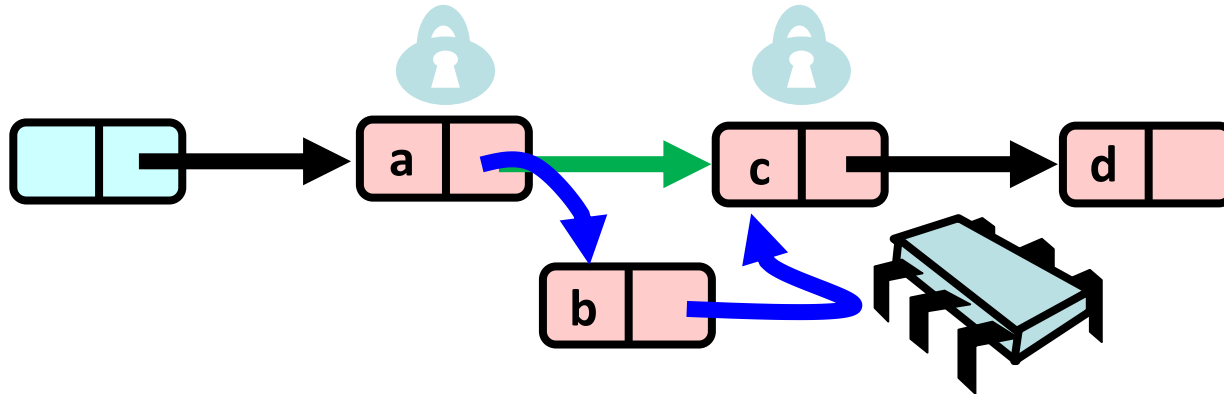
- A simple solution is to lock the entire list for each operation
  - E.g., by locking the head



- Simple and clearly correct!
- Works poorly with contention...

# Fine-Grained Locking

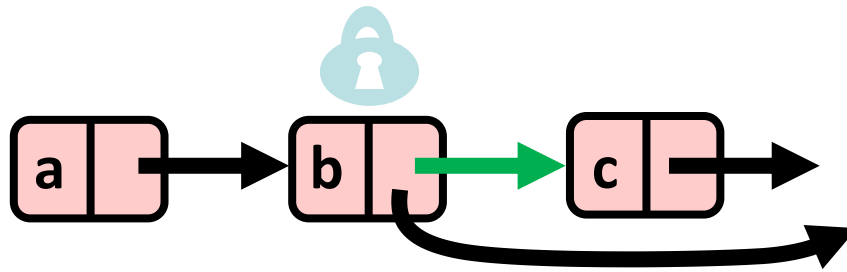
- Split object (list) into pieces (nodes)
  - Each piece (each node in the list) has its own lock
  - Methods that work on disjoint pieces need not exclude each other



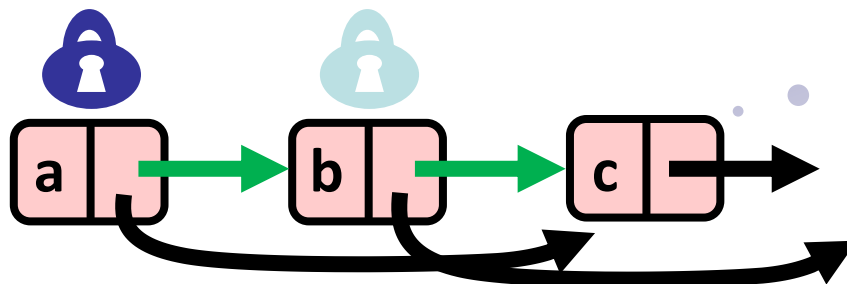
- Hand-over-hand locking: Use two locks when traversing the list
  - Why two locks?

# Problem with One Lock

- Assume that we want to delete node c
- We lock node b and set its next pointer to the node after c



- Another thread may concurrently delete node b by setting the next pointer from node a to node c

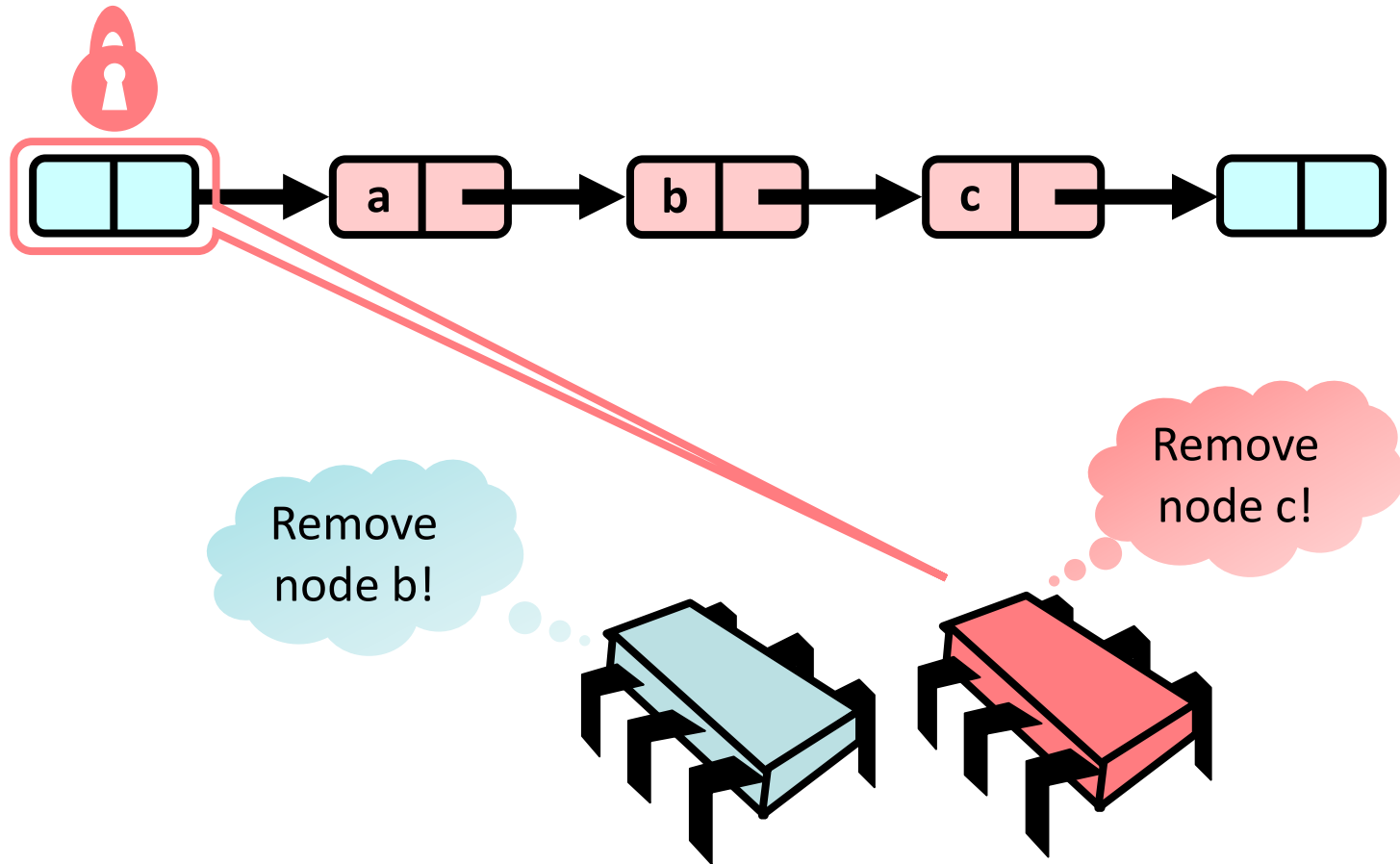


# Insight

- If a node is locked, no one can delete the node's *successor*
- If a thread locks
  - the node to be deleted
  - and also its predecessor
- then it works!
- That's why we (have to) use two locks!

# Hand-Over-Hand Locking: Removing Nodes

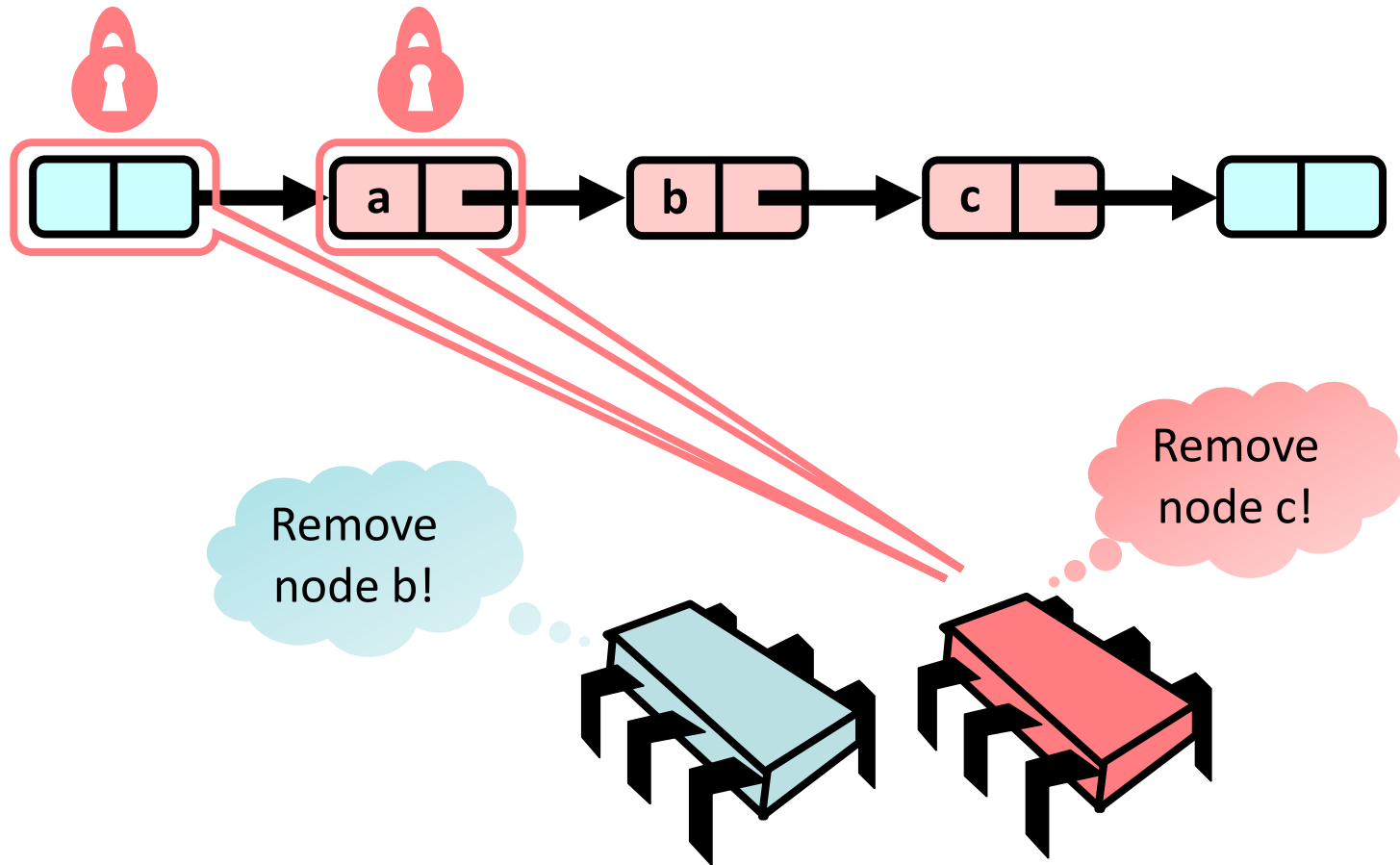
- Assume that two threads want to remove the nodes b and c
- One thread acquires the lock to the sentinel, the other has to wait





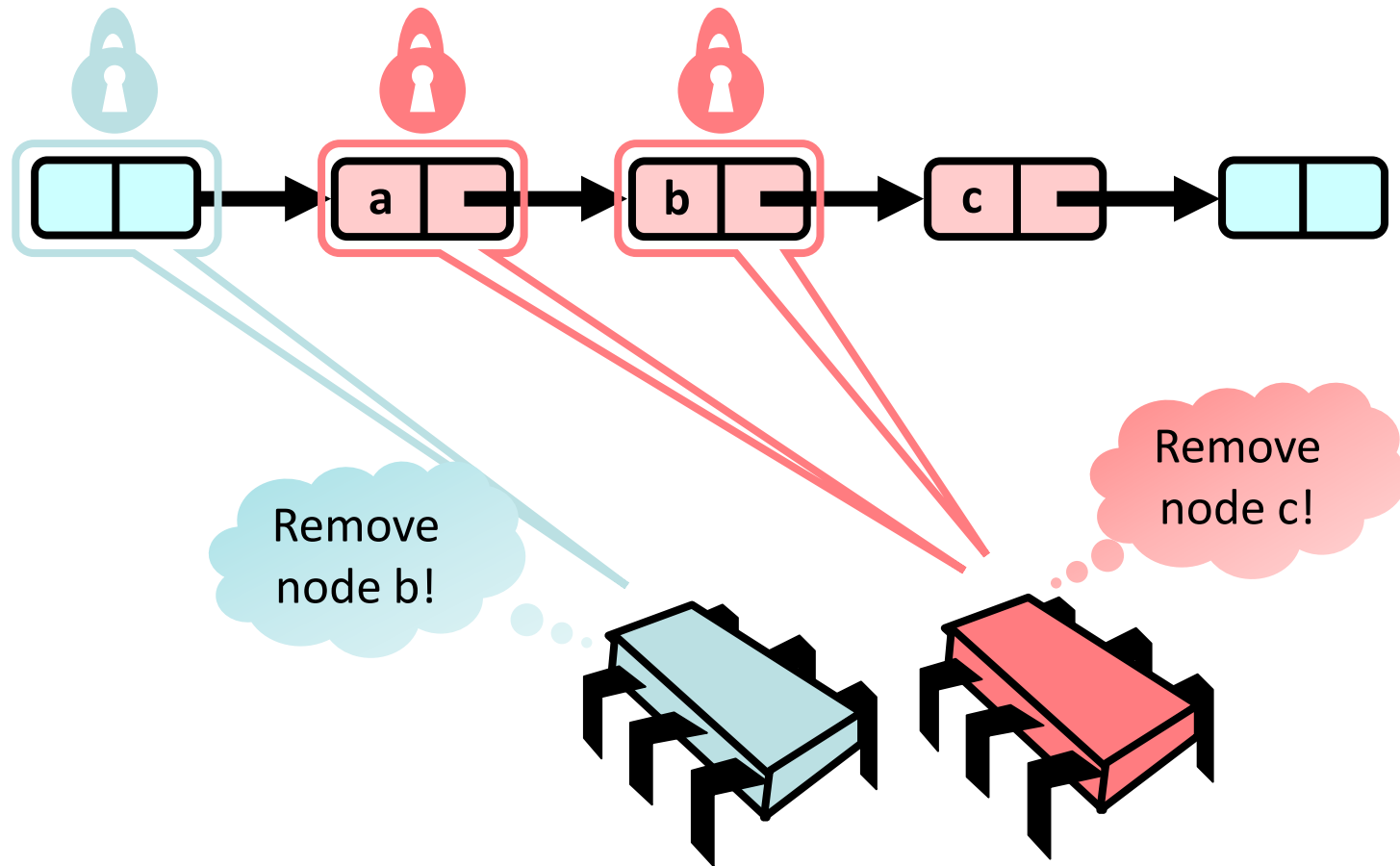
# Hand-Over-Hand Locking: Removing Nodes

- The same thread that acquired the sentinel lock can then lock the next node



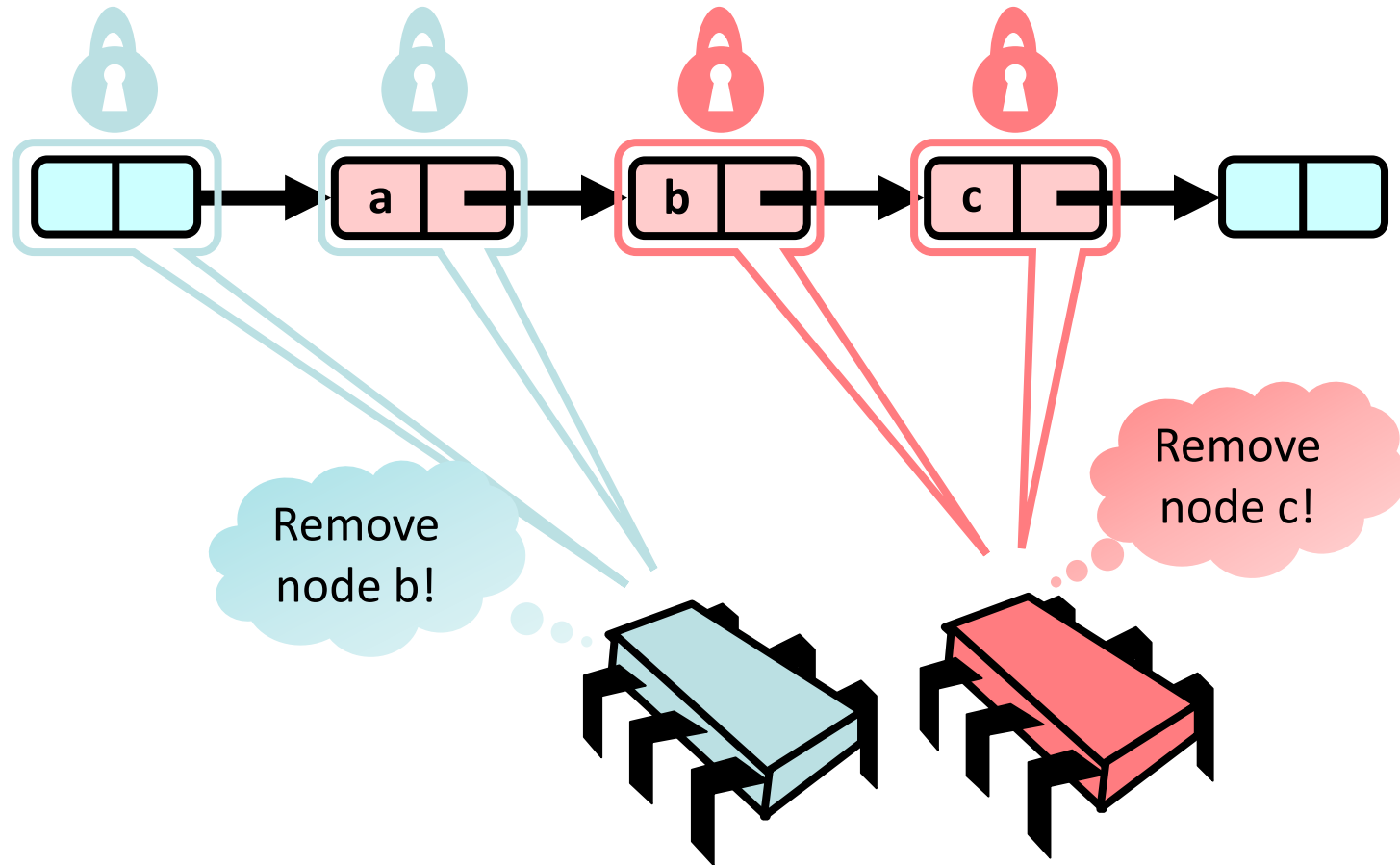
# Hand-Over-Hand Locking: Removing Nodes

- Before locking node b, the sentinel lock is released
- The other thread can now acquire the sentinel lock



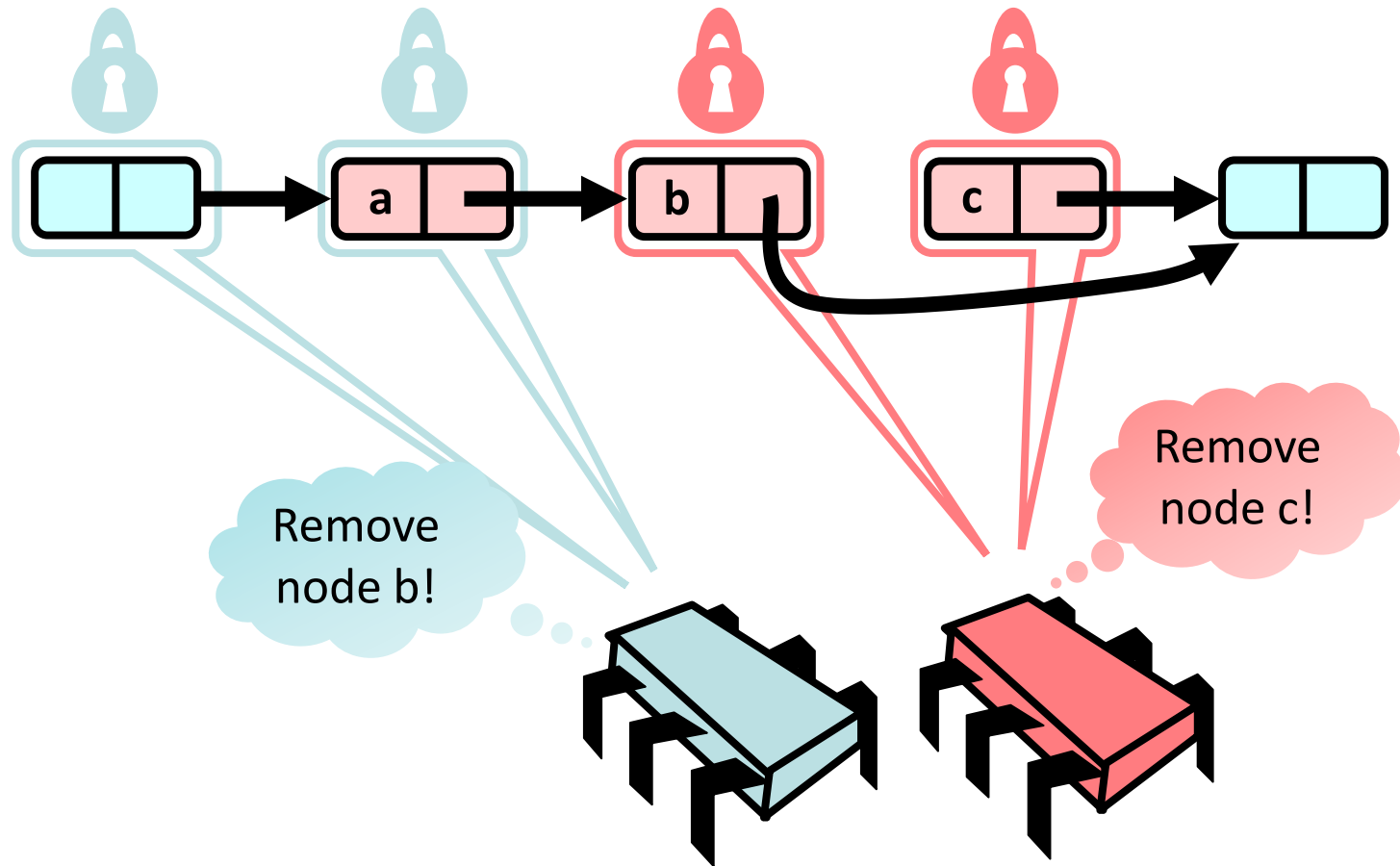
# Hand-Over-Hand Locking: Removing Nodes

- Before locking node c, the lock of node a is released
- The other thread can now lock node a



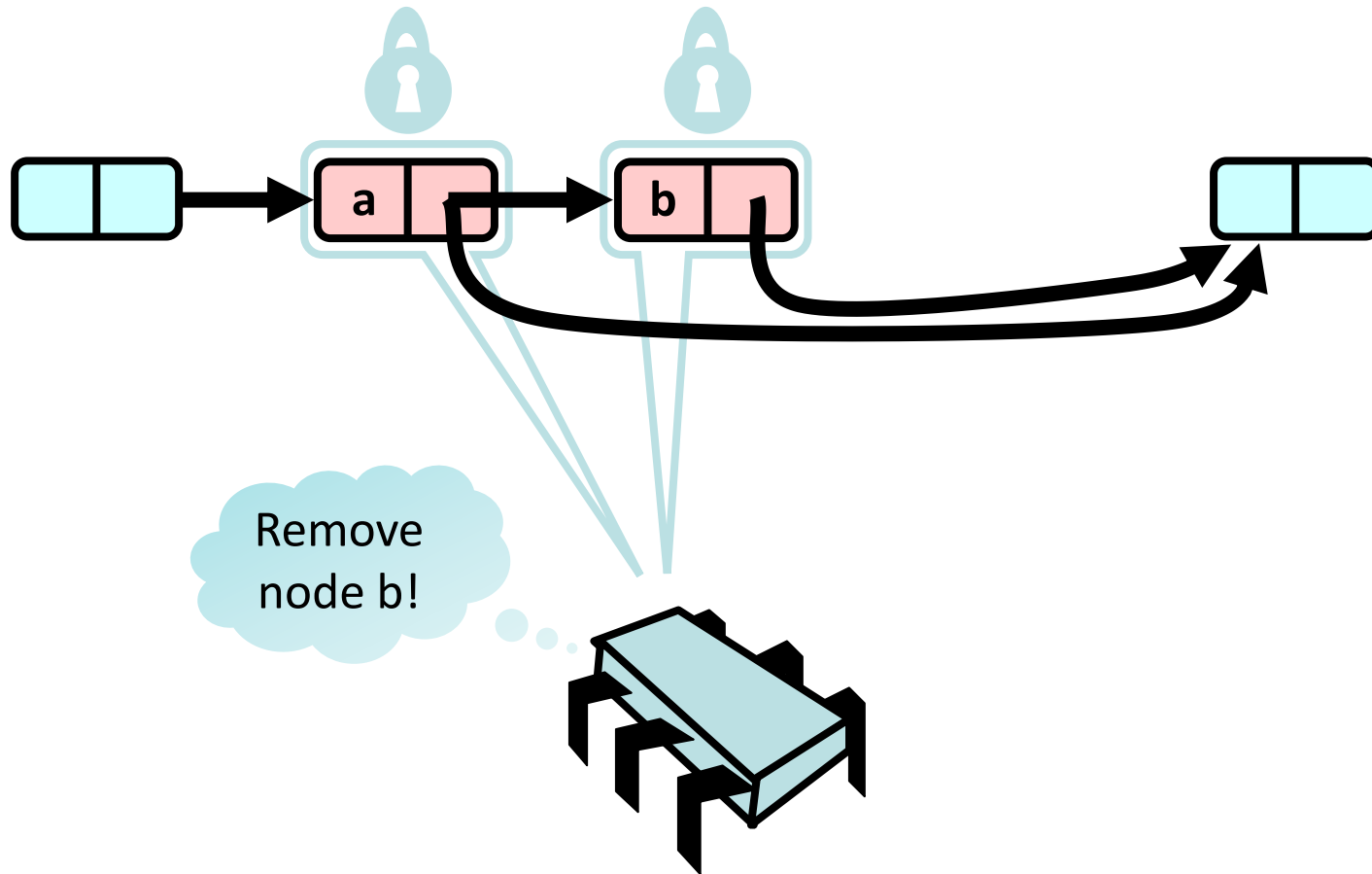
# Hand-Over-Hand Locking: Removing Nodes

- Node c can now be removed
- Afterwards, the two locks are released



# Hand-Over-Hand Locking: Removing Nodes

- The other thread can now lock node b and remove it



# List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```

**Item of interest**

**Usually a hash code**

**Reference to next node**

# Remove Method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        pred = this.head;  
        pred.lock();  
        curr = pred.next;  
        curr.lock();  
  
        ...  
  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
}
```

**Start at the head and lock it**

**Lock the current node**

**Traverse the list and  
remove the item**

**Make sure that the  
locks are released**

On the  
next slide!

## Remove Method

```
while (curr.key <= key) {
```

**Search key range**

```
  if (item == curr.item) {  
    pred.next = curr.next;  
    return true;  
  }
```

**If item found,  
remove the node**

```
  pred.unlock();  
  pred = curr;  
  curr = curr.next;  
  curr.lock();
```

**Unlock pred and  
lock the next node**

```
}
```

```
return false;
```

**Return false if the element is not present**



# Why does this work?

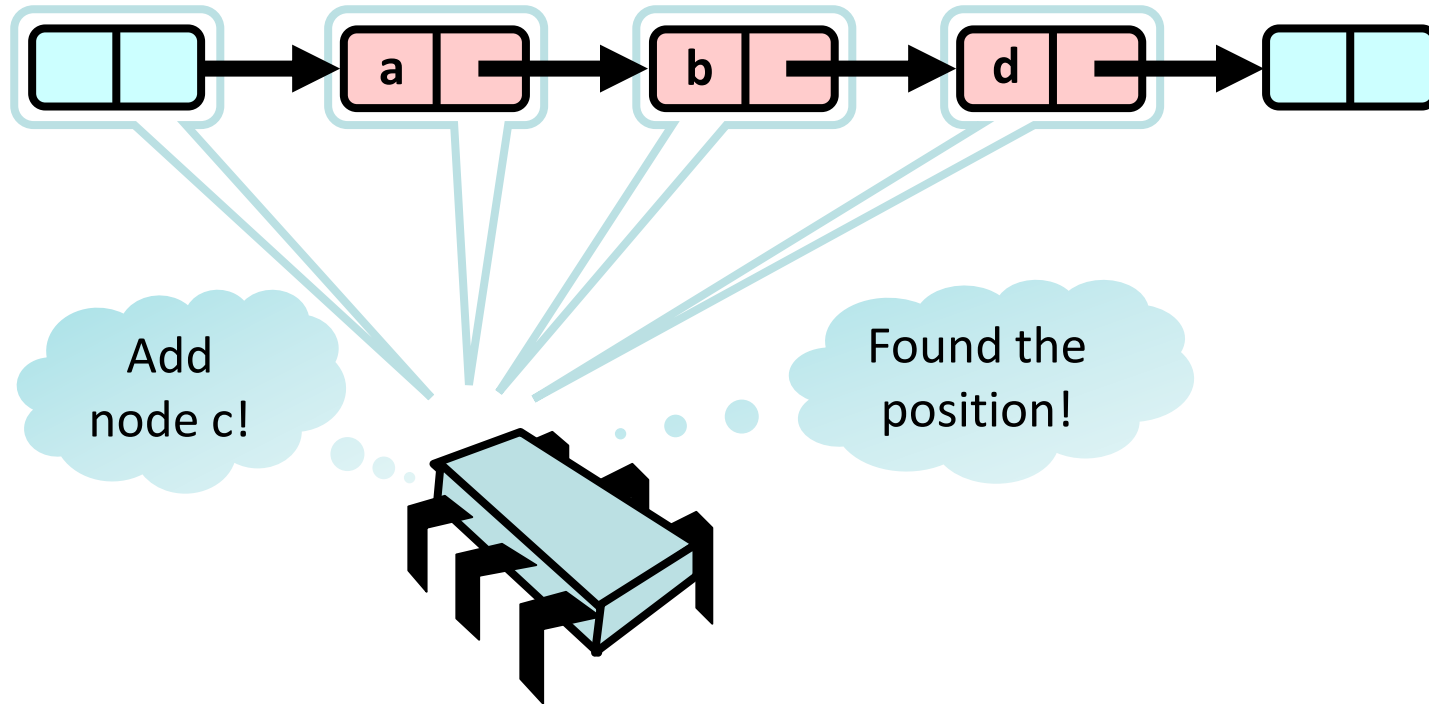
- To remove node e
  - Node e must be locked
  - Node e's predecessor must be locked
- Therefore, if you lock a node
  - It can't be removed
  - And neither can its successor
- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - Is the successor lock actually required?

# Drawbacks

- Hand-over-hand locking is sometimes better than coarse-grained locking
  - Threads can traverse in parallel
  - Sometimes, it's worse!
- However, it's certainly not ideal
  - Inefficient because many locks must be acquired and released
- How can we do better?

# Optimistic Synchronization

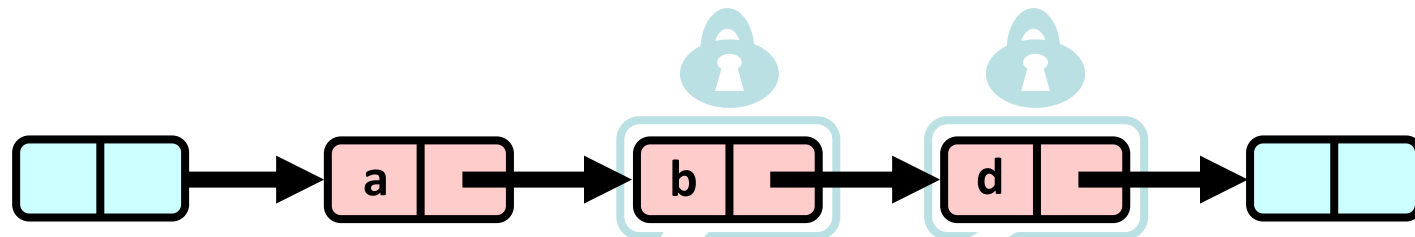
- Traverse the list without locking!



# Optimistic Synchronization: Traverse without Locking

- Once the nodes are found, try to lock them
- Check that everything is ok

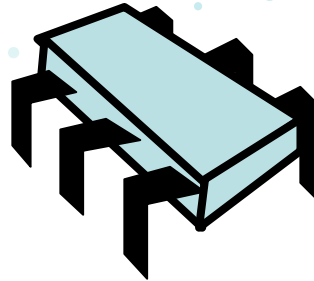
What could go wrong...?



Add node c!

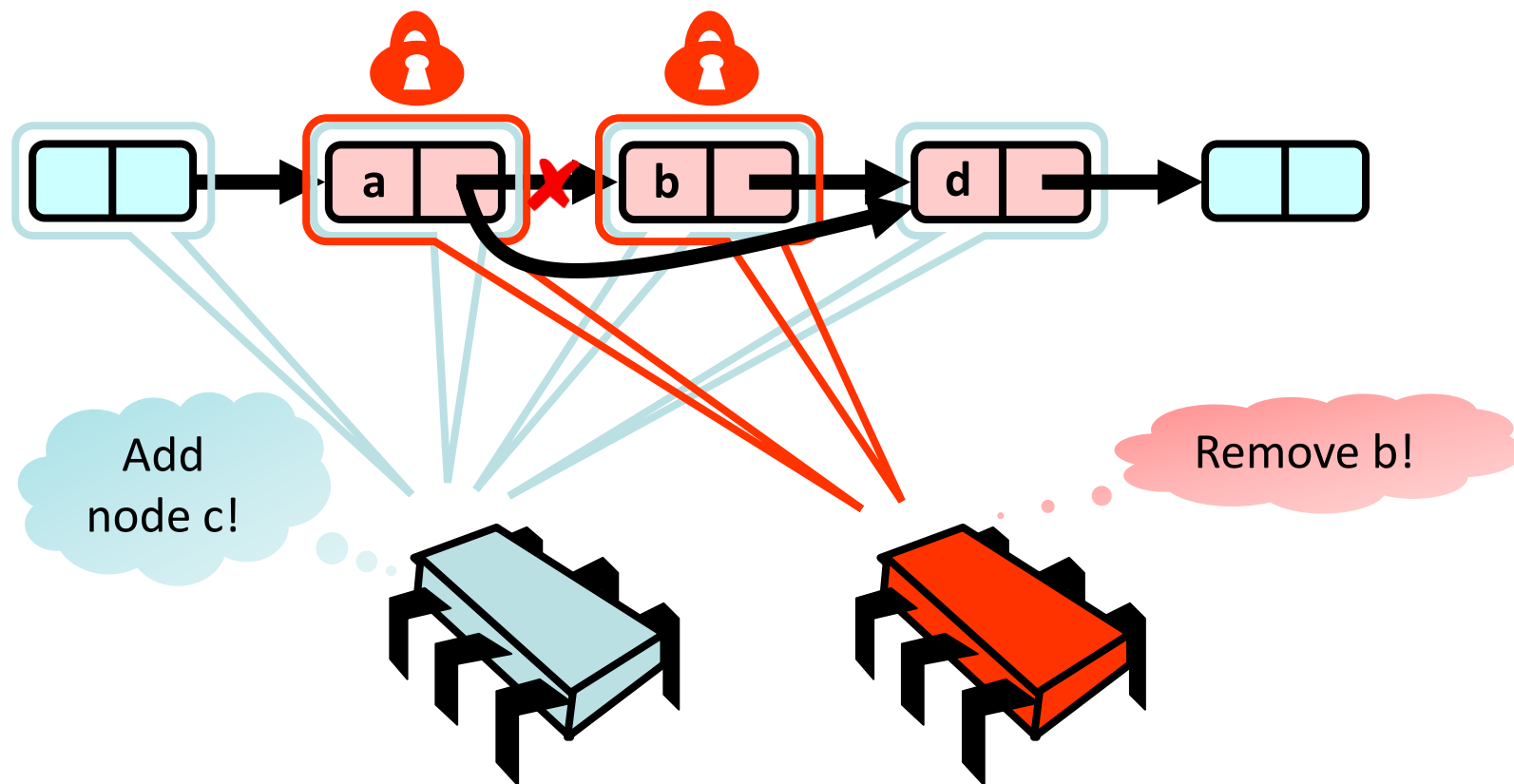
Lock them!

Is everything ok?



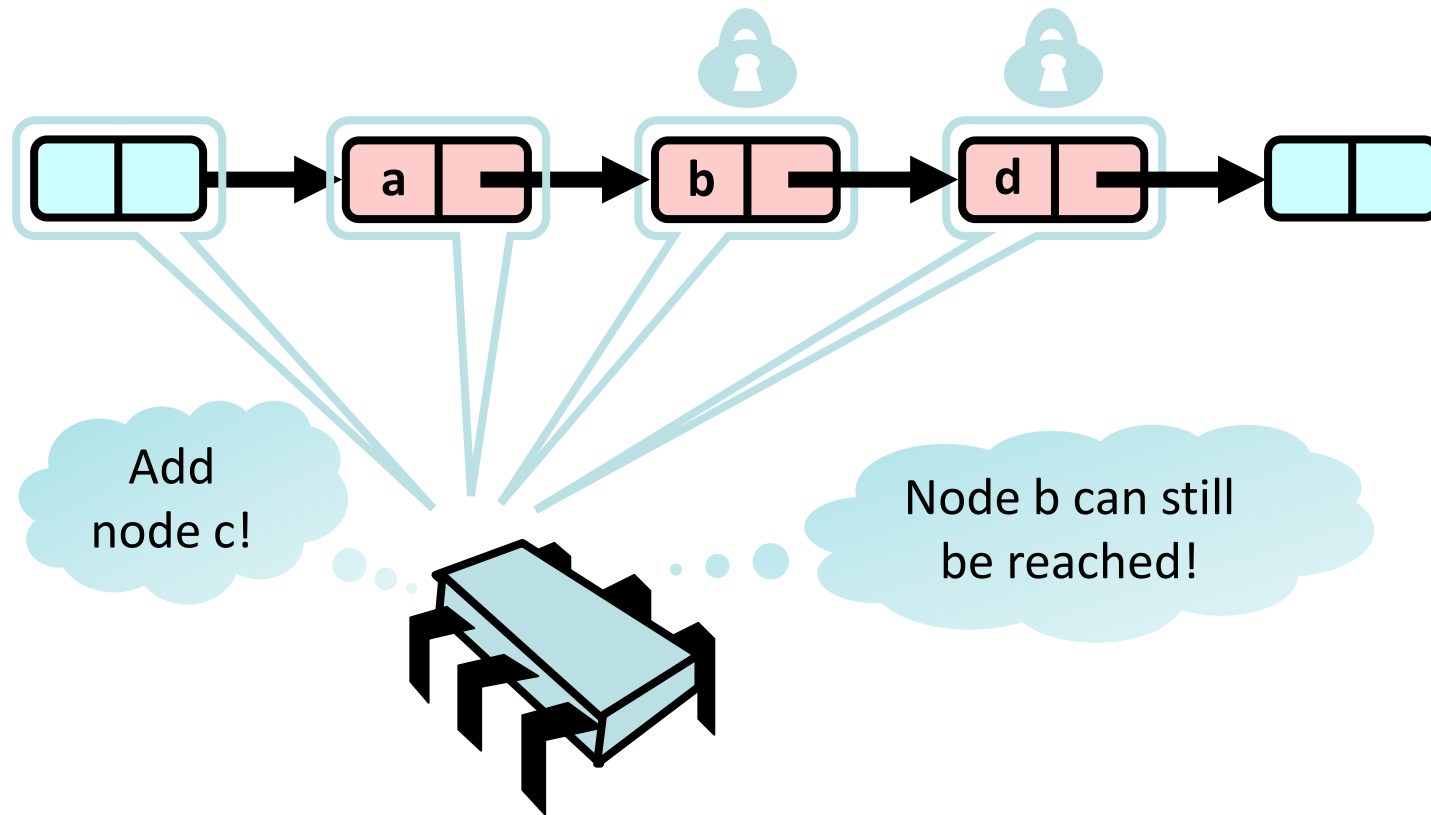
# Optimistic Synchronization: What Could Go Wrong?

- Another thread may lock nodes a and b and remove b before node c is added → If the pointer from node b is set to node c, then node c is not added to the list!



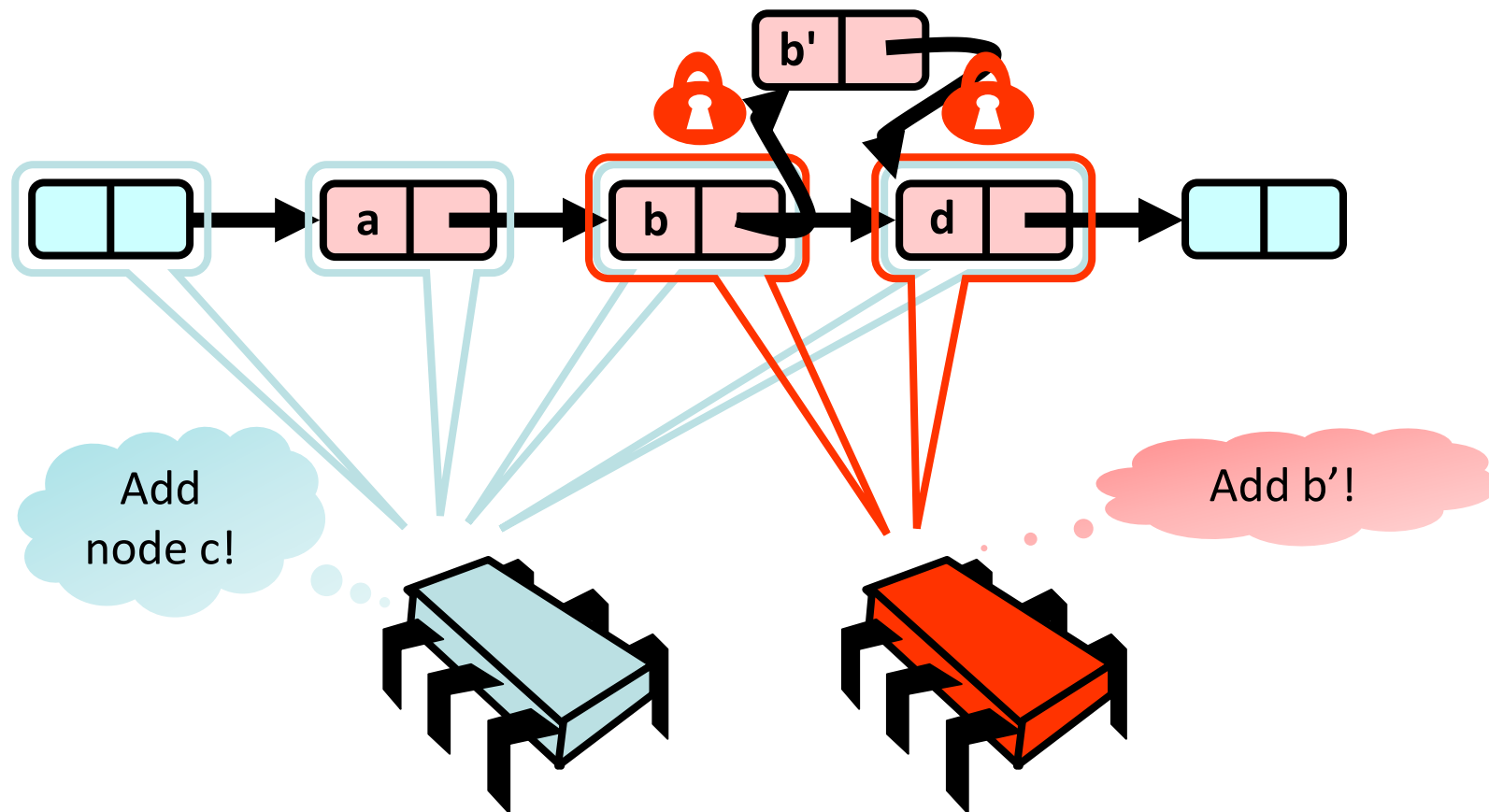
# Optimistic Synchronization: Validation #1

- How can this be fixed?
- After locking node b and node d, traverse the list again to verify that b is still reachable



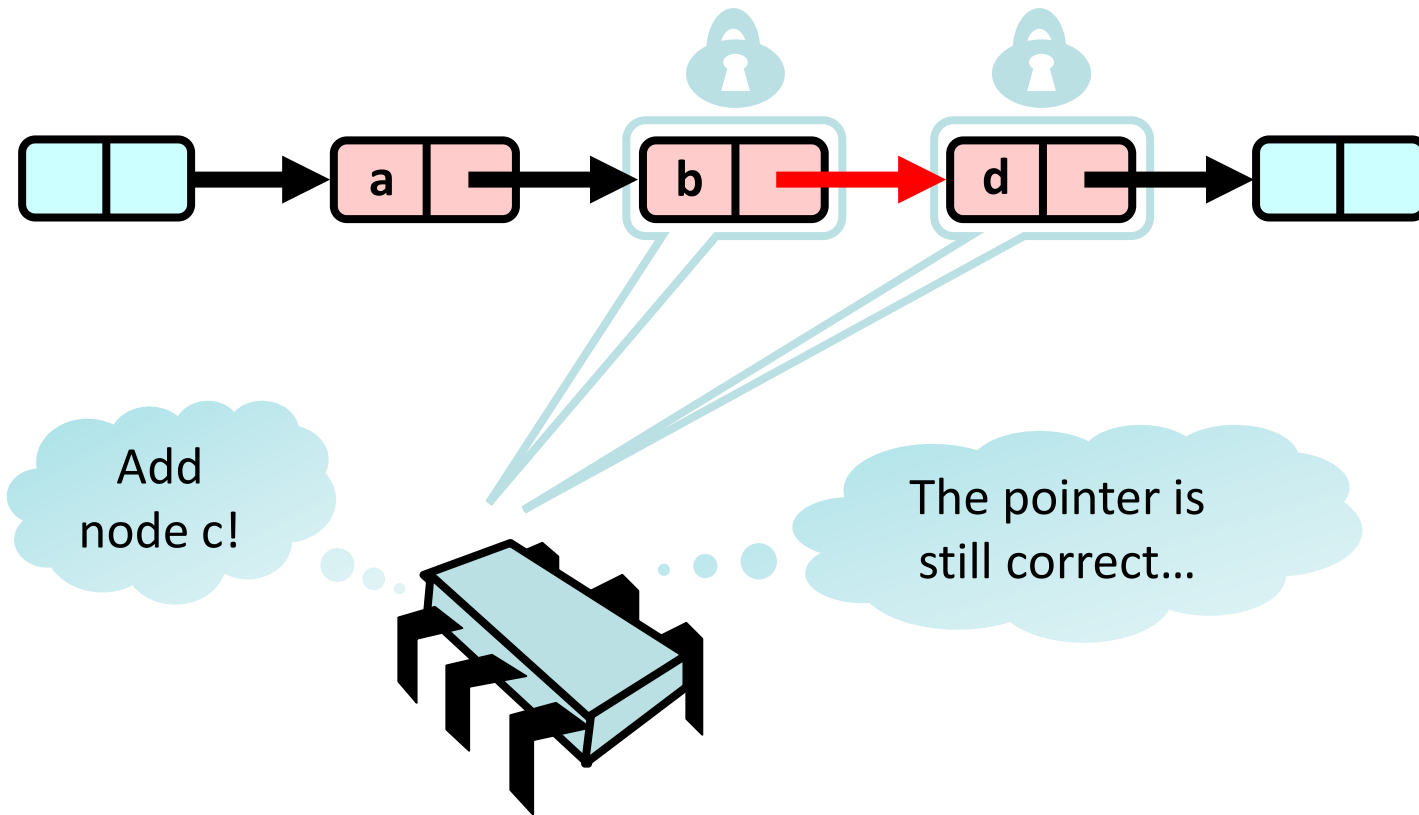
# Optimistic Synchronization: What Else Could Go Wrong?

- Another thread may lock nodes b and d and add a node b' before node c is added → By adding node c, the addition of node b' is undone!



## Optimistic Synchronization: Validation #2

- How can this be fixed?
- After locking node b and node d, also check that node b still points to node d!





## Optimistic Synchronization: Validation

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

**If pred is reached,  
test if the  
successor is curr**

**Predecessor not reachable**

## Optimistic Synchronization: Remove

```
private boolean remove(Item item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        }  
        ...  
    }  
}
```

**Retry on synchronization  
conflict**

**Stop if we find the item**

# Optimistic Synchronization: Remove

```
...
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    }
} finally {
    pred.unlock();
    curr.unlock();
}
}
```

**Lock both nodes**

**Check for synchronization conflicts**

**Remove node if target found**

**Always unlock the nodes**

# Optimistic Synchronization

- Why is this correct?
  - If nodes b and c are both locked, node b still accessible, and node c still the successor of node b, then neither b nor c will be deleted by another thread
  - This means that it's ok to delete node c!
- Why is it good to use optimistic synchronization?
  - Limited hot-spots: no contention on traversals
  - Fewer lock acquisitions and releases
- When is it good to use optimistic synchronization?
  - When the cost of scanning twice without locks is less than the cost of scanning once with locks
- Can we do better?
  - It would be better to traverse the list only once...

# Lazy Synchronization

- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”
- How can we remove nodes “lazily”?
  - First perform a logical delete: Mark current node as removed (new!)



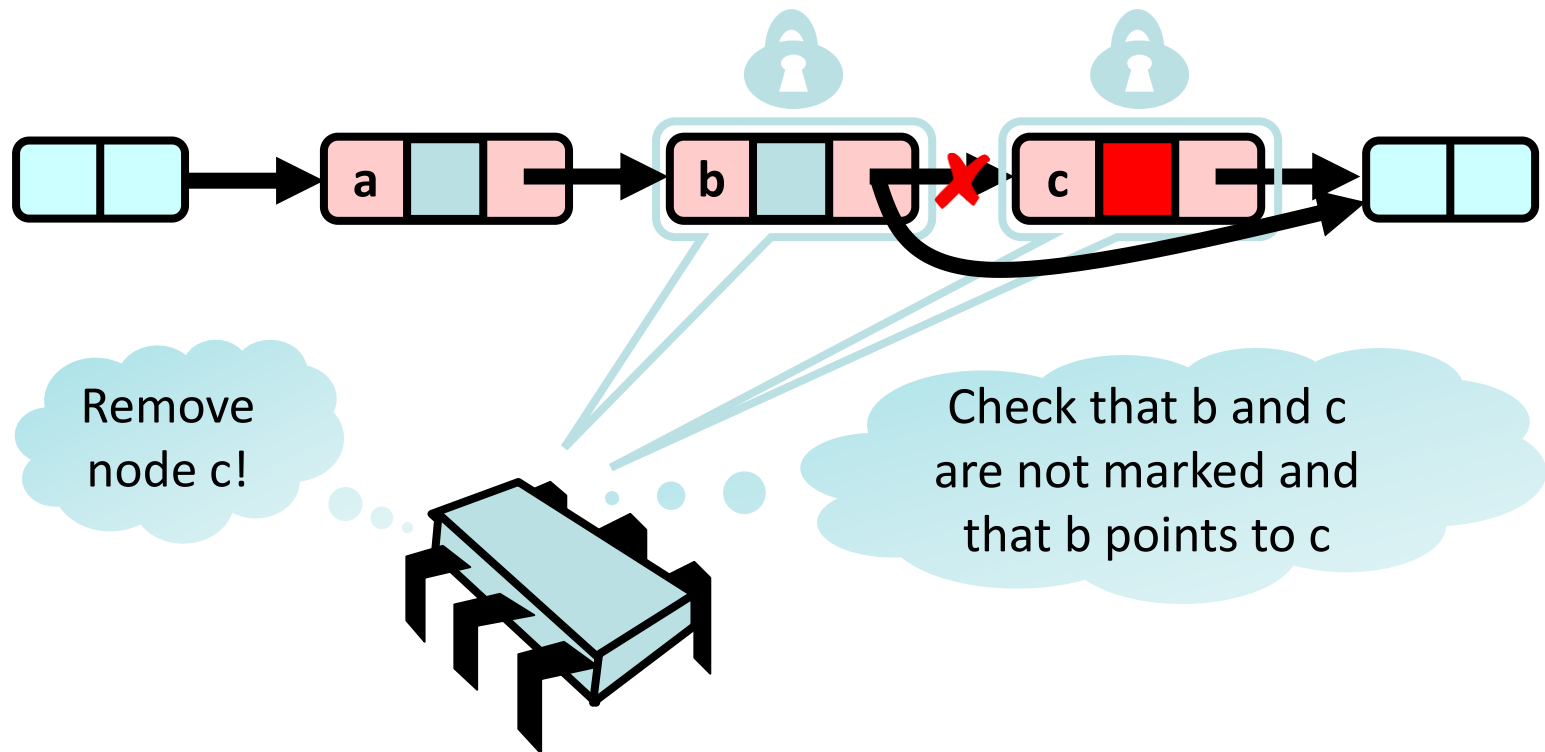
- Then perform a physical delete: Redirect predecessor’s next (as before)

# Lazy Synchronization

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls...
- Note that we must still lock pred and curr nodes!
- How does validation work?
  - Check that neither pred nor curr are marked
  - Check that pred points to curr

# Lazy Synchronization

- Traverse the list and then try to lock the two nodes
- Validate!
- Then, mark node c and change the predecessor's next pointer



## Lazy Synchronization: Validation

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

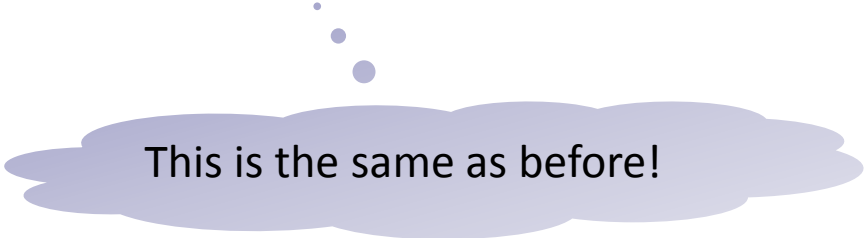
**Nodes are not  
logically removed**

**Predecessor still  
points to current**



## Lazy Synchronization: Remove

```
public boolean remove(Item item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
        ...
    }
}
```



This is the same as before!

## Lazy Synchronization: Remove

```
...
try {
    pred.lock(); curr.lock();
    if (validate(pred,curr)) {
        if (curr.item == item) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    }
} finally {
    pred.unlock();
    curr.unlock();
}
}
```

**Check for  
synchronization conflicts**

**If the target is found,  
mark the node and  
remove it**

## Lazy Synchronization: Contains

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

**Traverse without locking  
(nodes may have been  
removed)**

**Is the element present and not marked?**

# Evaluation

- Good
  - The list is traversed only once without locking
  - Note that contains() doesn't lock at all!
  - This is nice because typically contains() is called much more often than add() or remove()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays
- Traffic jam?
  - If one thread gets the lock and experiences a cache miss/page fault, every other thread that needs the lock is stuck!
  - We need to trust the scheduler....

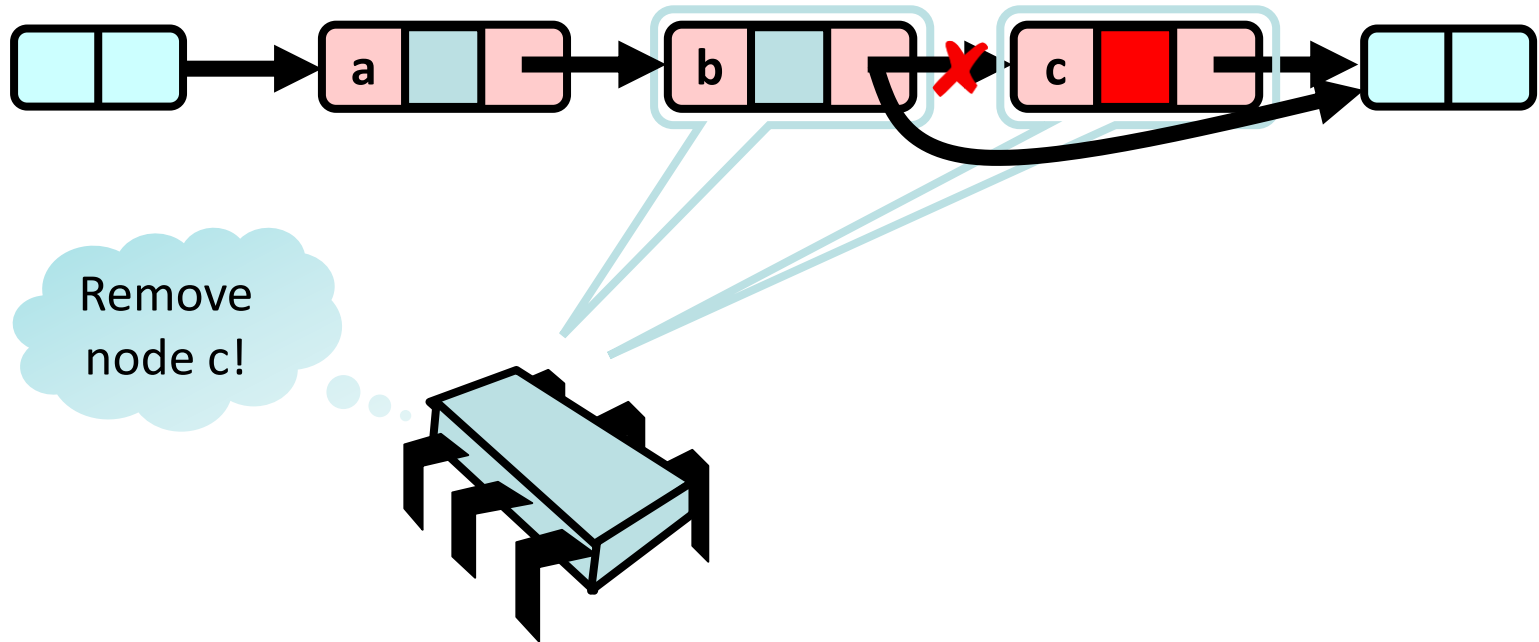
# Lock-Free Data Structures

- If we want to guarantee that some thread will eventually complete a method call, even if other threads may halt at malicious times, then the implementation cannot use locks!
- Next logical step: Eliminate locking entirely!
- Obviously, we must use some sort of RMW method
- Let's use CompareAndSet() (CAS)!



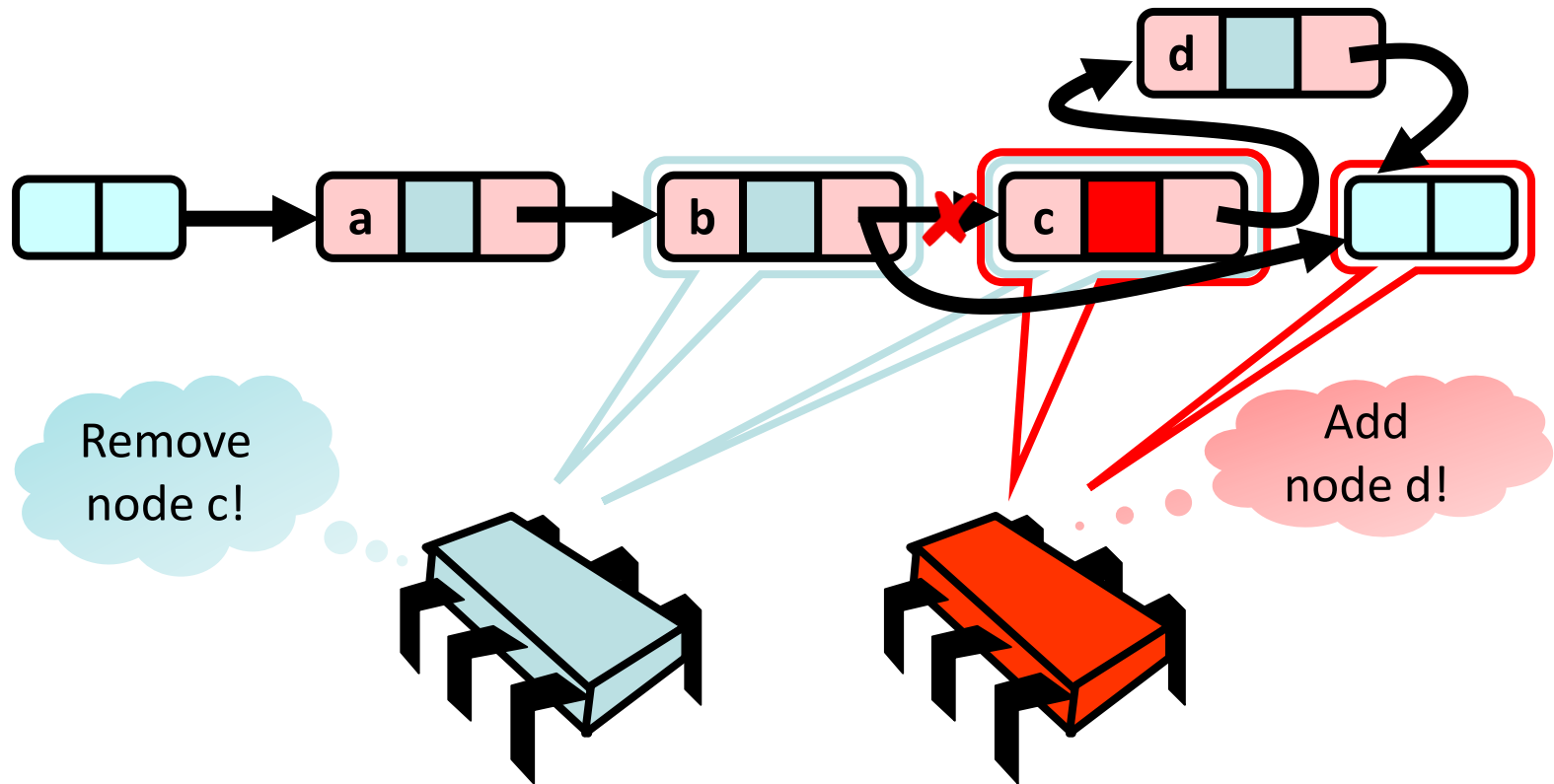
# Remove Using CAS

- First, remove the node logically (i.e., mark it)
- Then, use CAS to change the next pointer
- Does this work...?



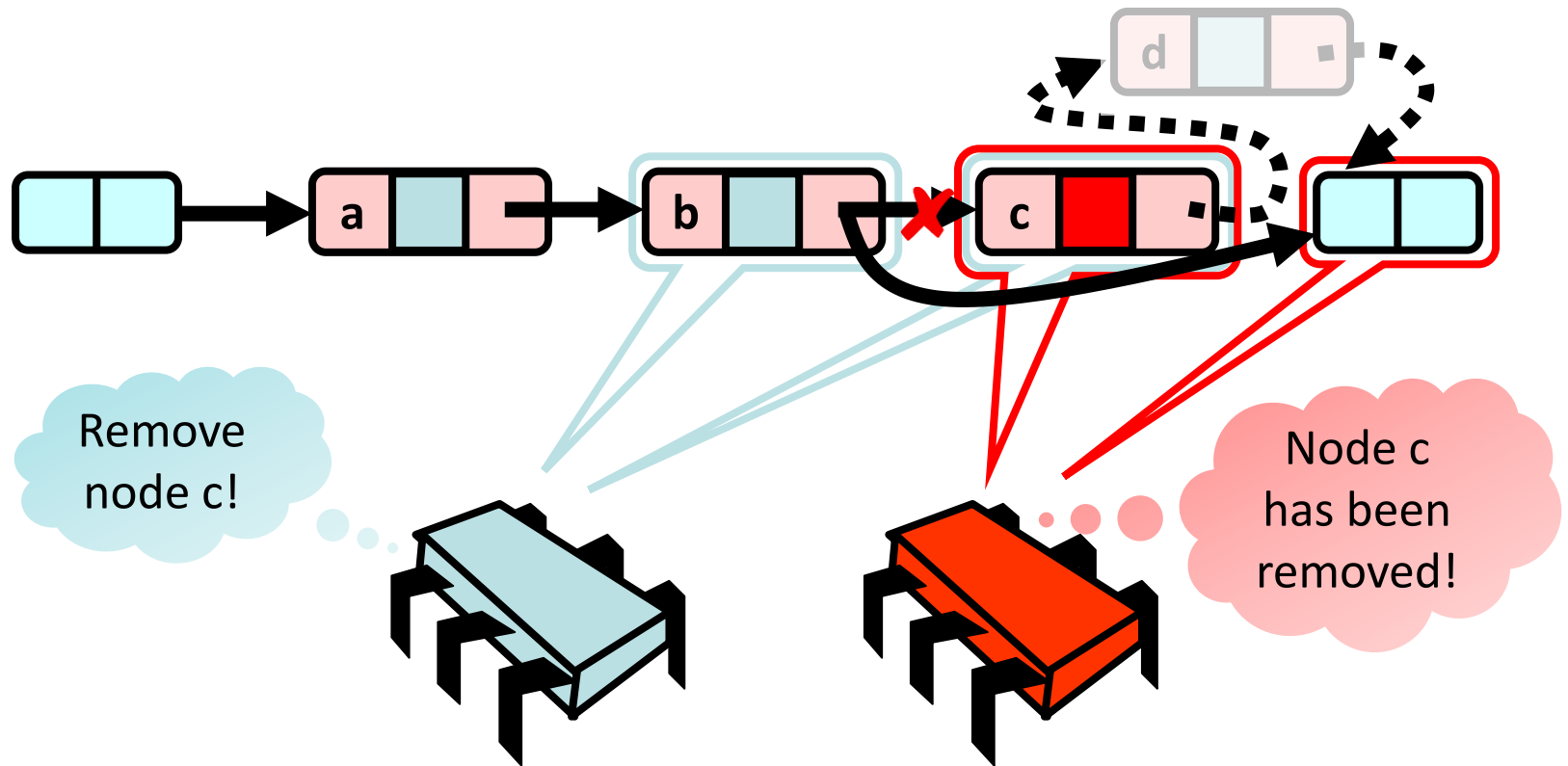
# Remove Using CAS: Problem

- Unfortunately, this doesn't work!
- Another node d may be added before node c is physically removed
- As a result, node d is not added to the list...



# Solution

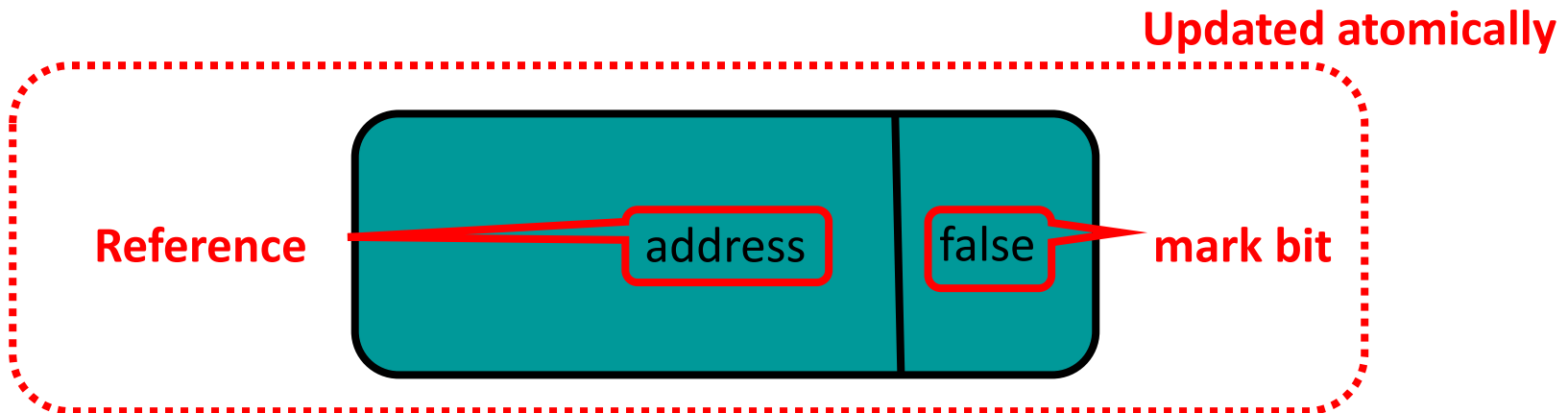
- Mark bit and next pointer are “CASed together”
- This atomic operation ensures that no node can cause a conflict by adding (or removing) a node at the same position in the list





# Solution

- Such an operation is called an **atomic markable reference**
  - Atomically update the mark bit and redirect the predecessor's next pointer
- In Java, there's an AtomicMarkableReference class
  - In the package `Java.util.concurrent.atomic` package



## Changing State

```
private Object ref;  
private boolean mark;
```

**The reference to the next  
Object and the mark bit**

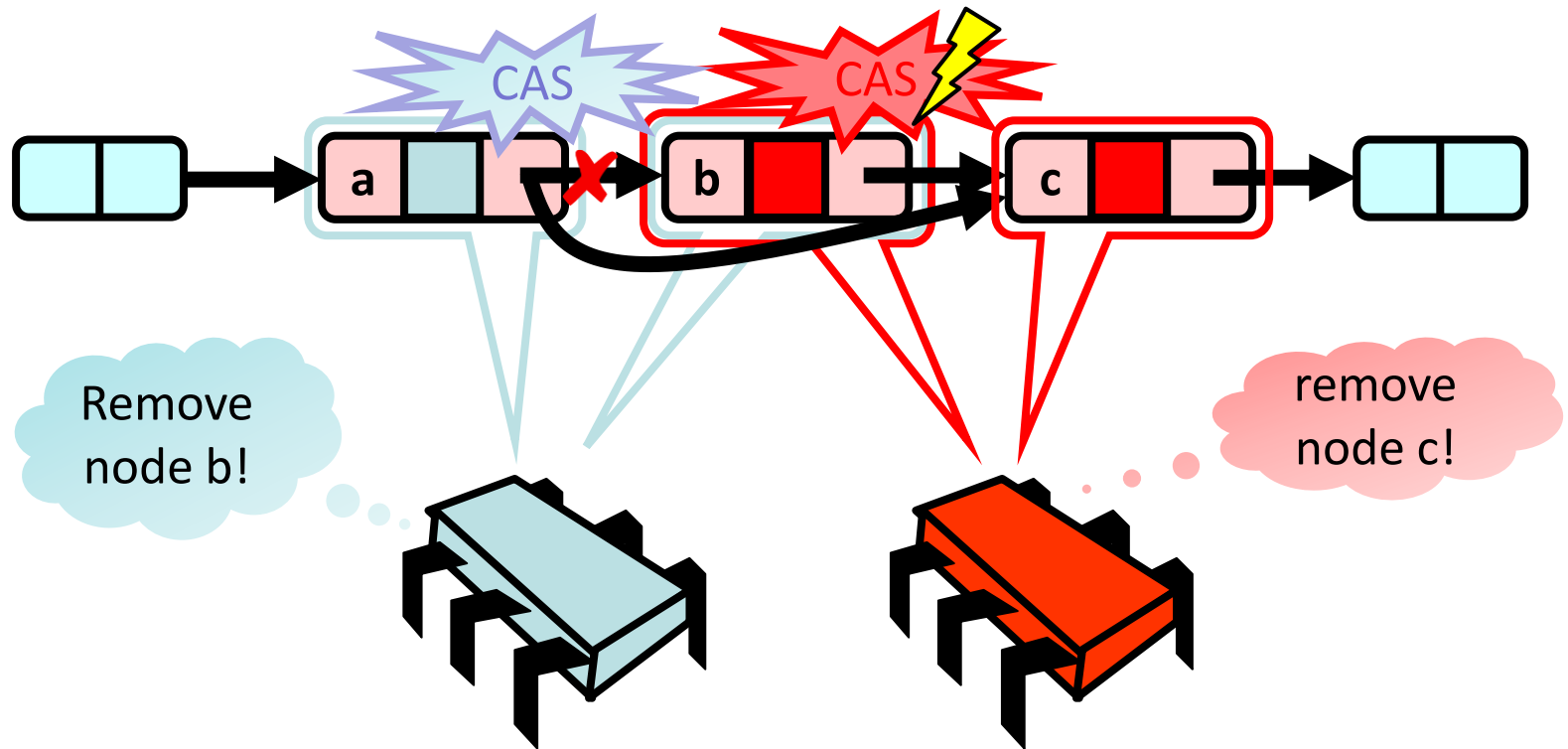
```
public synchronized boolean compareAndSet(  
Object expectedRef, Object updateRef,  
boolean expectedMark, boolean updateMark) {
```

```
    if (ref == expectedRef && mark == expectedMark) {  
        ref = updateRef;  
        mark = updateMark;  
    }  
}
```

**If the reference and the mark are as  
expected, update them atomically**

# Removing a Node

- If two threads want to delete the nodes b and c, both b and c are marked
- The CAS of the red thread fails because node b is marked!
- (If node b is not marked, then b is removed first and there is no conflict)



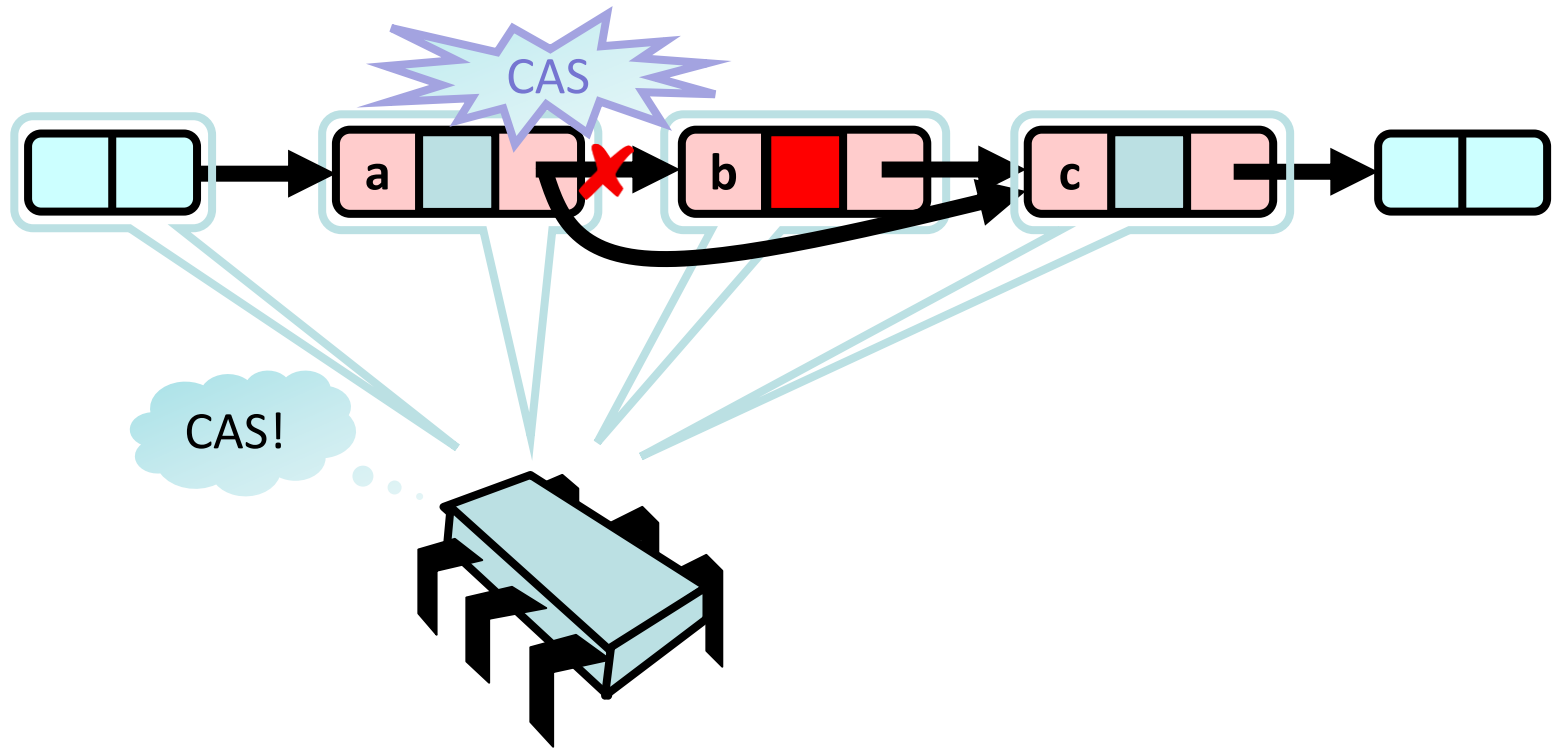
# Traversing the List

- Question: What do you do when you find a “logically” deleted node in your path when you’re traversing the list?



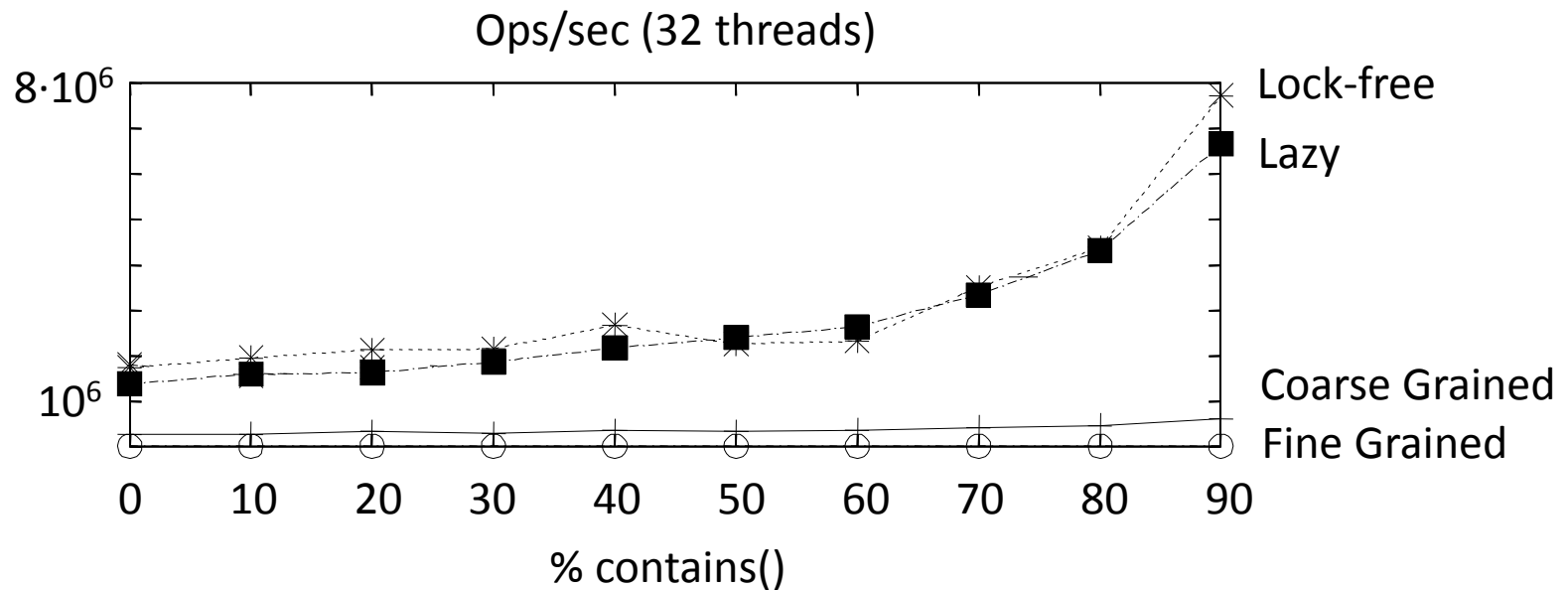
# Lock-Free Traversal

- If a logically deleted node is encountered, CAS the predecessor's next field and proceed (repeat as needed)



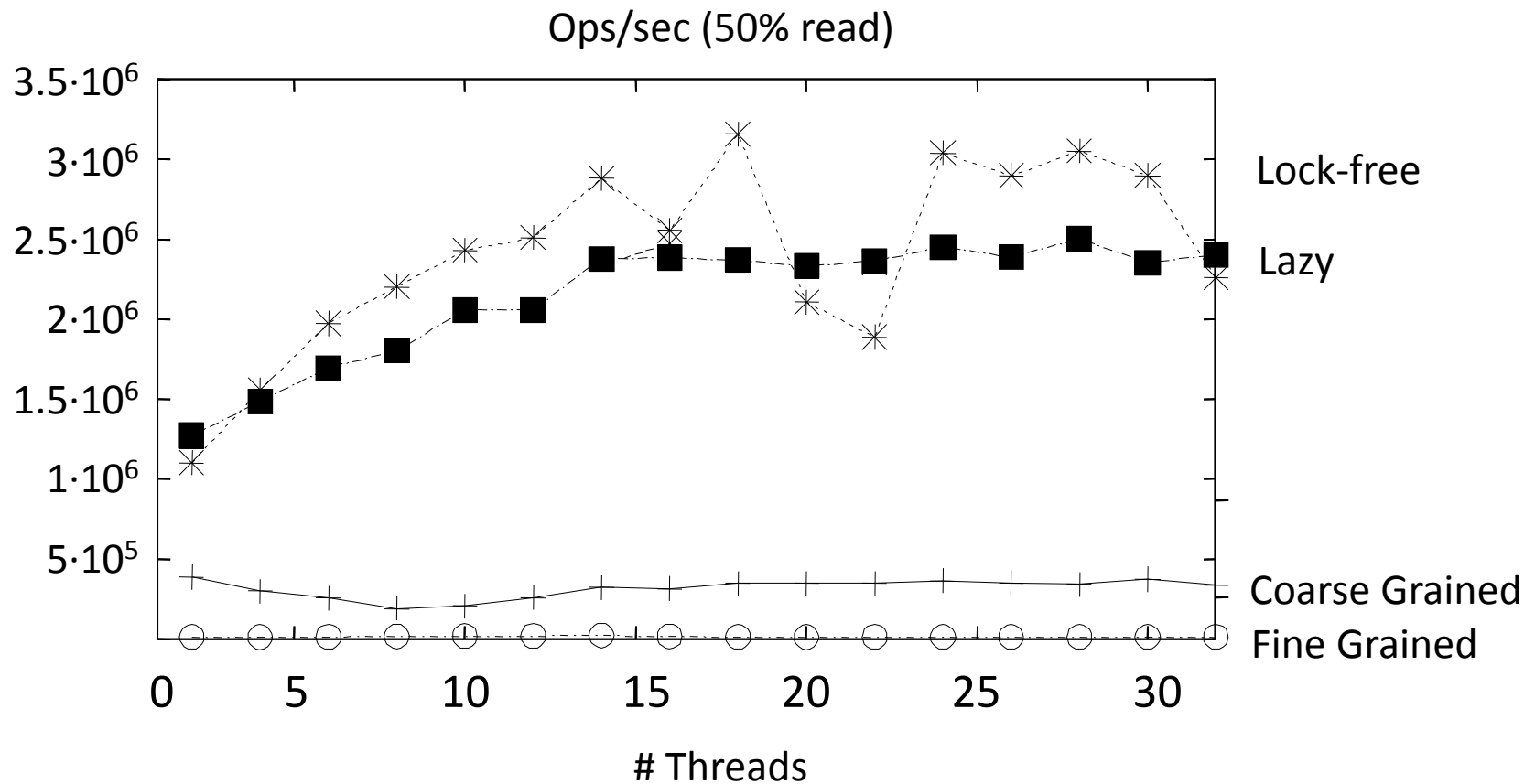
# Performance

- The throughput of the presented techniques has been measured for a varying percentage of contains() method calls
  - Using a benchmark on a 16 node shared memory machine



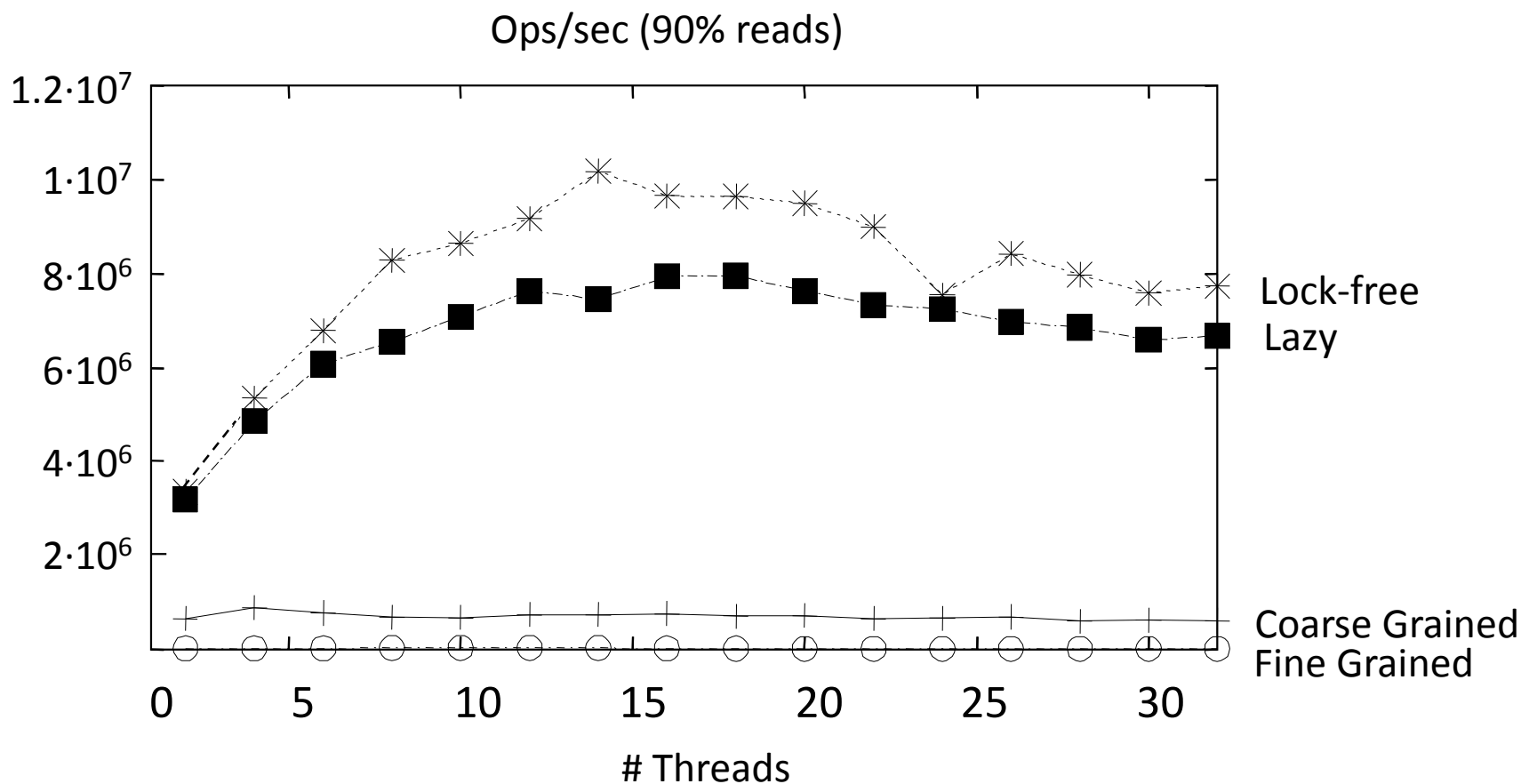
## Low Ratio of contains()

- If the ratio of contains() is low, the lock-free linked list and the linked list with lazy synchronization perform well even if there are many threads



## High Ratio of contains()

- If the ratio of contains() is high, again both the lock-free linked list and the linked list with lazy synchronization perform well even if there are many threads






## “To Lock or Not to Lock”

- Locking vs. non-blocking: Extremist views on both sides
- It is nobler to compromise by combining locking and non-blocking techniques
  - Example: Linked list with lazy synchronization combines blocking `add()` and `remove()` and a non-blocking `contains()`
  - Blocking/non-blocking is a property of a method

# Linear-Time Set Methods

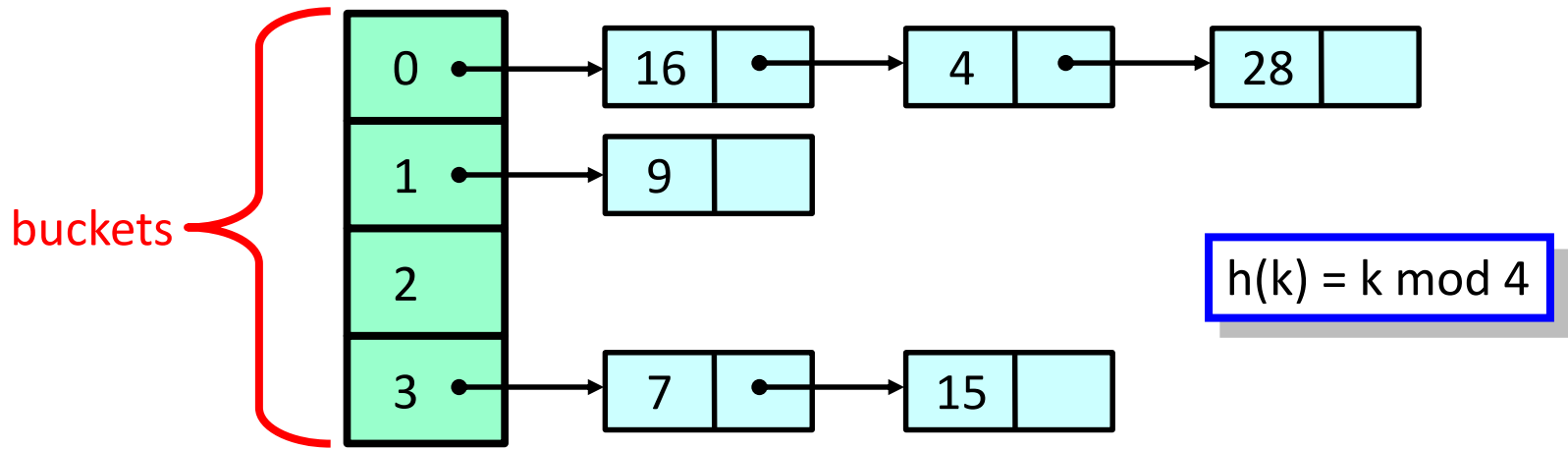
- We looked at a number of ways to make highly-concurrent list-based sets
  - Fine-grained locks
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- What's not so great?
  - `add()`, `remove()`, `contains()` take time **linear in the set size**
- We want constant-time methods! ···  How...?
  - At least on average...

# Hashing

- A hash function maps the items to integers
  - $h: \text{items} \rightarrow \text{integers}$
- Uniformly distributed
  - Different items “most likely” have different hash values
- In Java there is a `hashCode()` method

# Sequential Hash Map

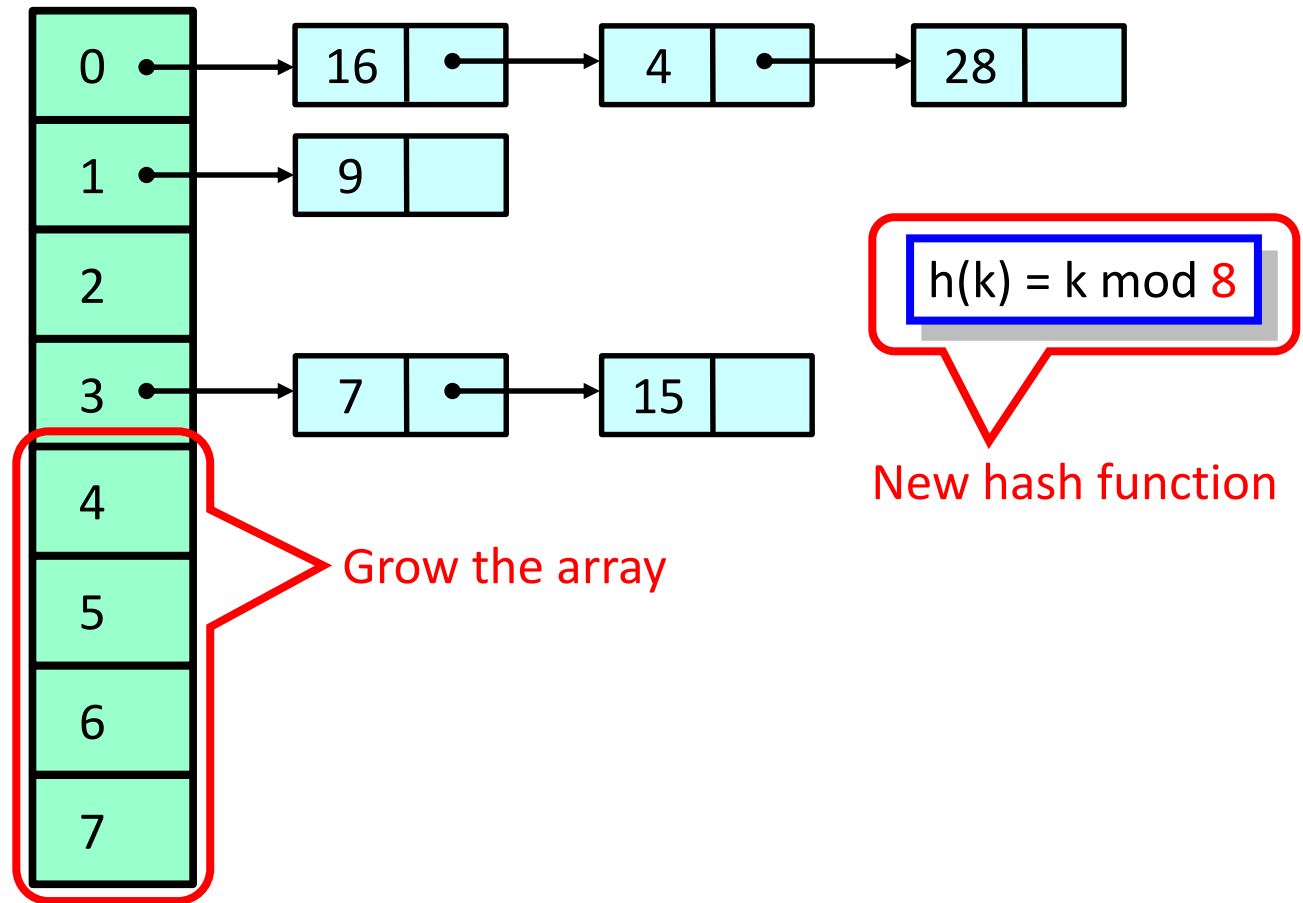
- The hash table is implemented as an array of buckets, each pointing to a list of items



- Problem: If many items are added, the lists get long → Inefficient lookups!
- Solution: Resize!

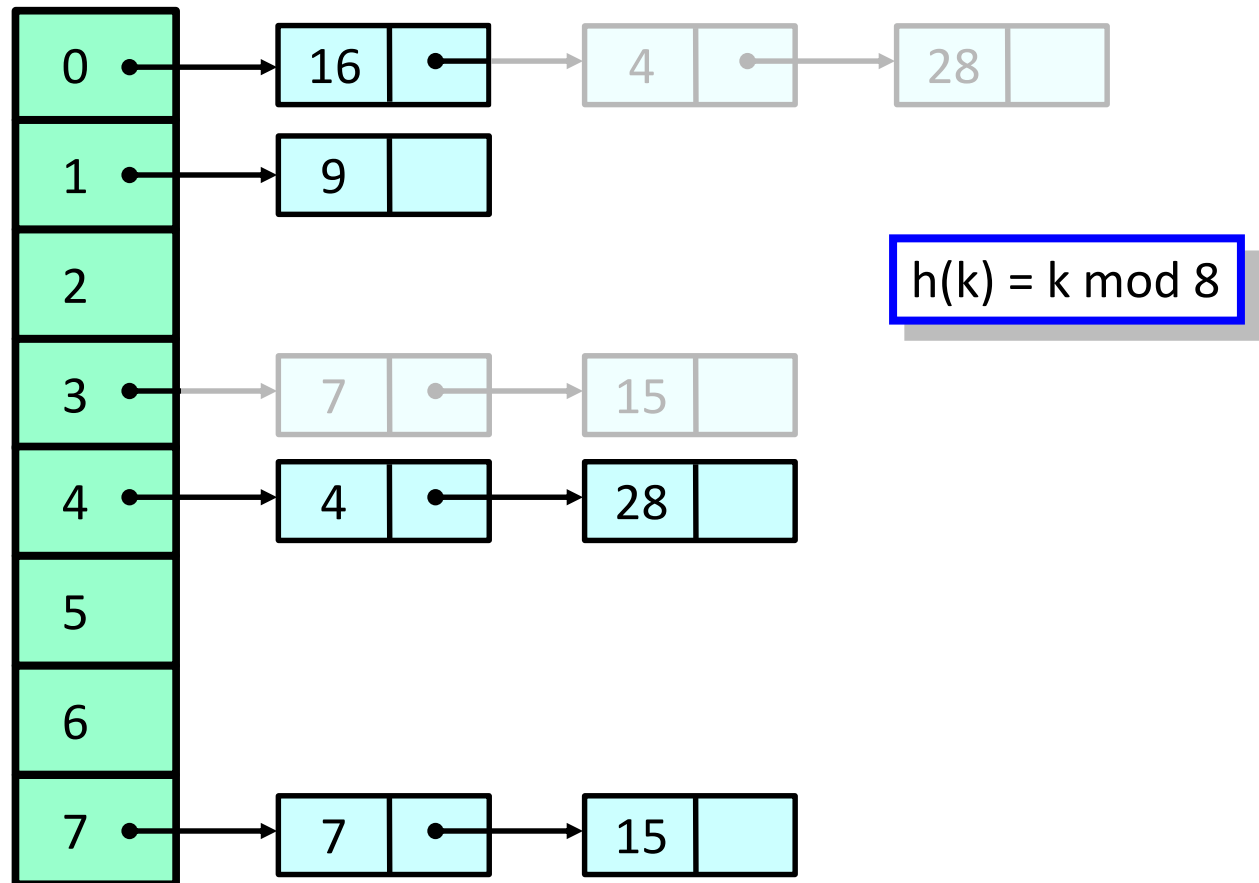
# Resizing

- The array size is doubled and the hash function adjusted



# Resizing

- Some items have to be moved to different buckets!



# Hash Sets

- A hash set implements a set object
  - Collection of items, no duplicates
  - `add()`, `remove()`, `contains()` methods
- More coding ahead!



# Simple Hash Set

```
public class SimpleHashSet {  
    protected LockFreeList[] table;  
  
    public SimpleHashSet(int capacity) {  
        table = new LockFreeList[capacity];  
        for (int i = 0; i < capacity; i++) {  
            table[i] = new LockFreeList();  
        }  
    }  
  
    public boolean add(Object key) {  
        int hash = key.hashCode() % table.length;  
        return table[hash].add(key);  
    }  
    ...  
}
```

Array of lock-free lists

Initial size

Initialization

Use hash of object to pick a bucket  
and call bucket's add() method



# Simple Hash Set: Evaluation

- We just saw a
  - Simple
  - Lock-free
  - Concurrenthash-based set implementation
- But we don't know **how to resize...**
- Is Resizing really necessary?
  - Yes, since constant-time method calls require **constant-length buckets** and a **table size proportional to the set size**
  - As the set grows, we must be able to resize

# Set Method Mix

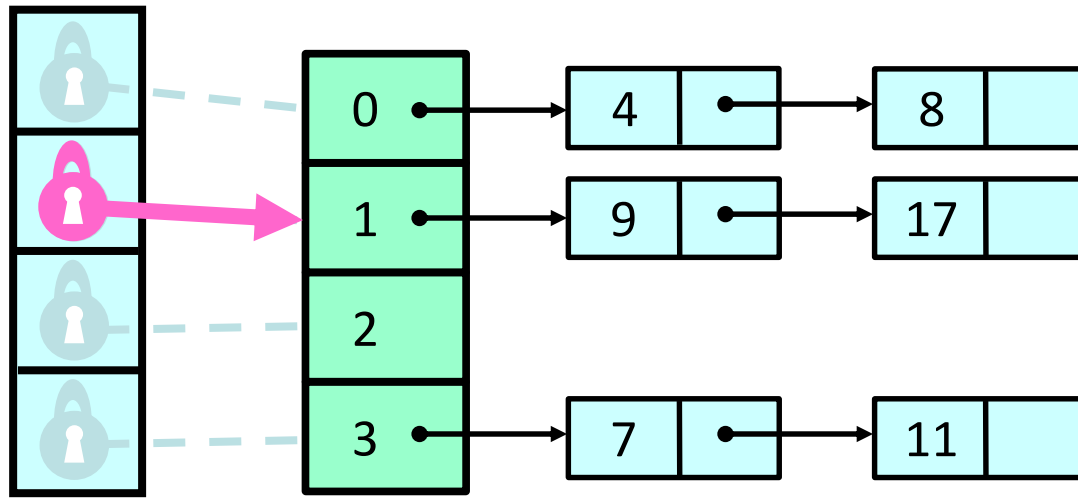
- Typical load
  - 90% contains()
  - 9% add ()
  - 1% remove()
- Growing is important, shrinking not so much
- When do we resize?
- There are many reasonable policies, e.g., pick a threshold on the number of items in a bucket
- Global threshold
  - When, e.g.,  $\geq \frac{1}{4}$  buckets exceed this value
- Bucket threshold
  - When any bucket exceeds this value

# Coarse-Grained Locking

- If there are concurrent accesses, how can we **safely** resize the array?
- As with the linked list, a straightforward solution is to use coarse-grained locking: lock the entire array!
- This is very simple and correct
- However, we again get a sequential bottleneck...
- How about fine-grained locking?

# Fine-Grained Locking

- Each lock is associated with one bucket

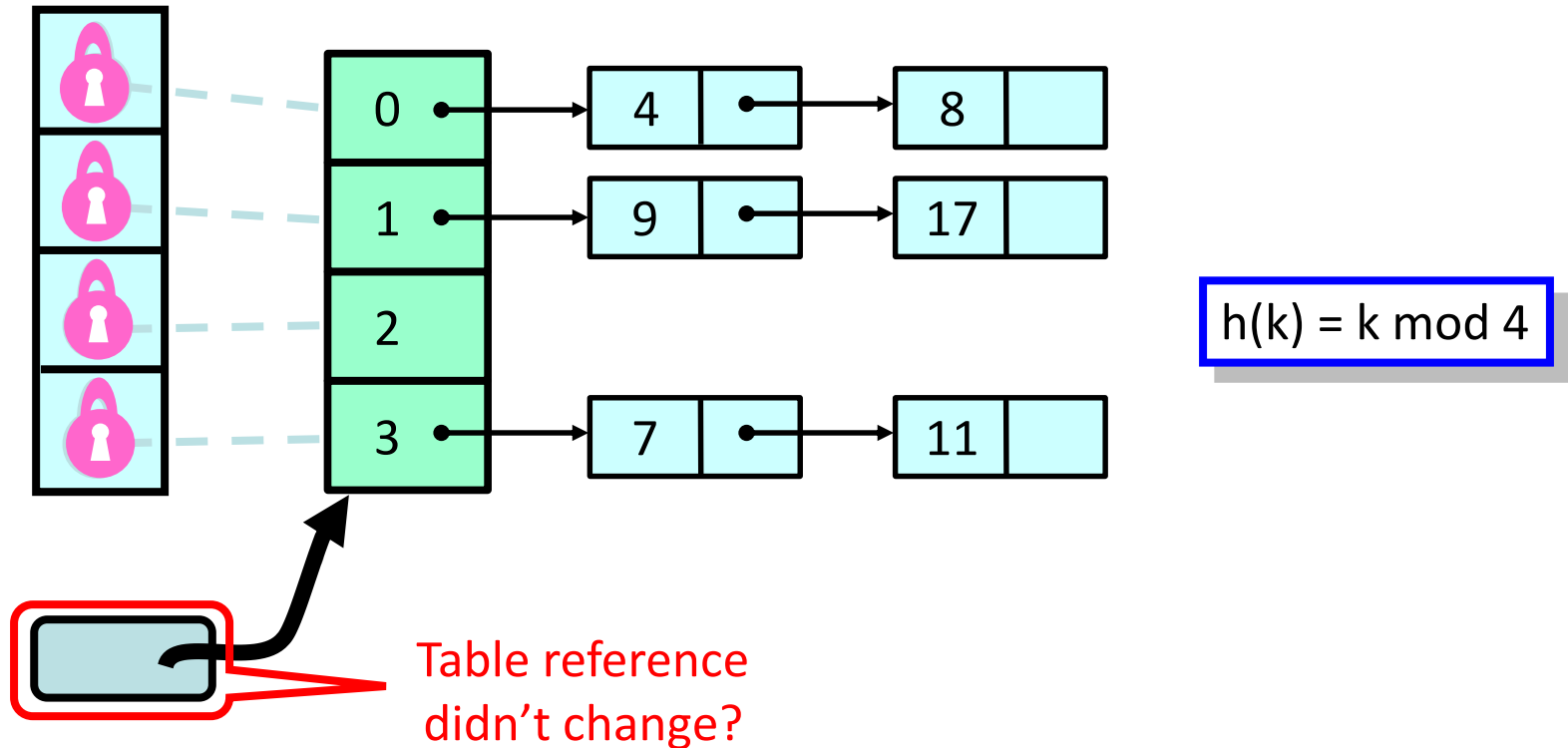


$$h(k) = k \bmod 4$$

- After acquiring the lock of the list, insert the item in the list!

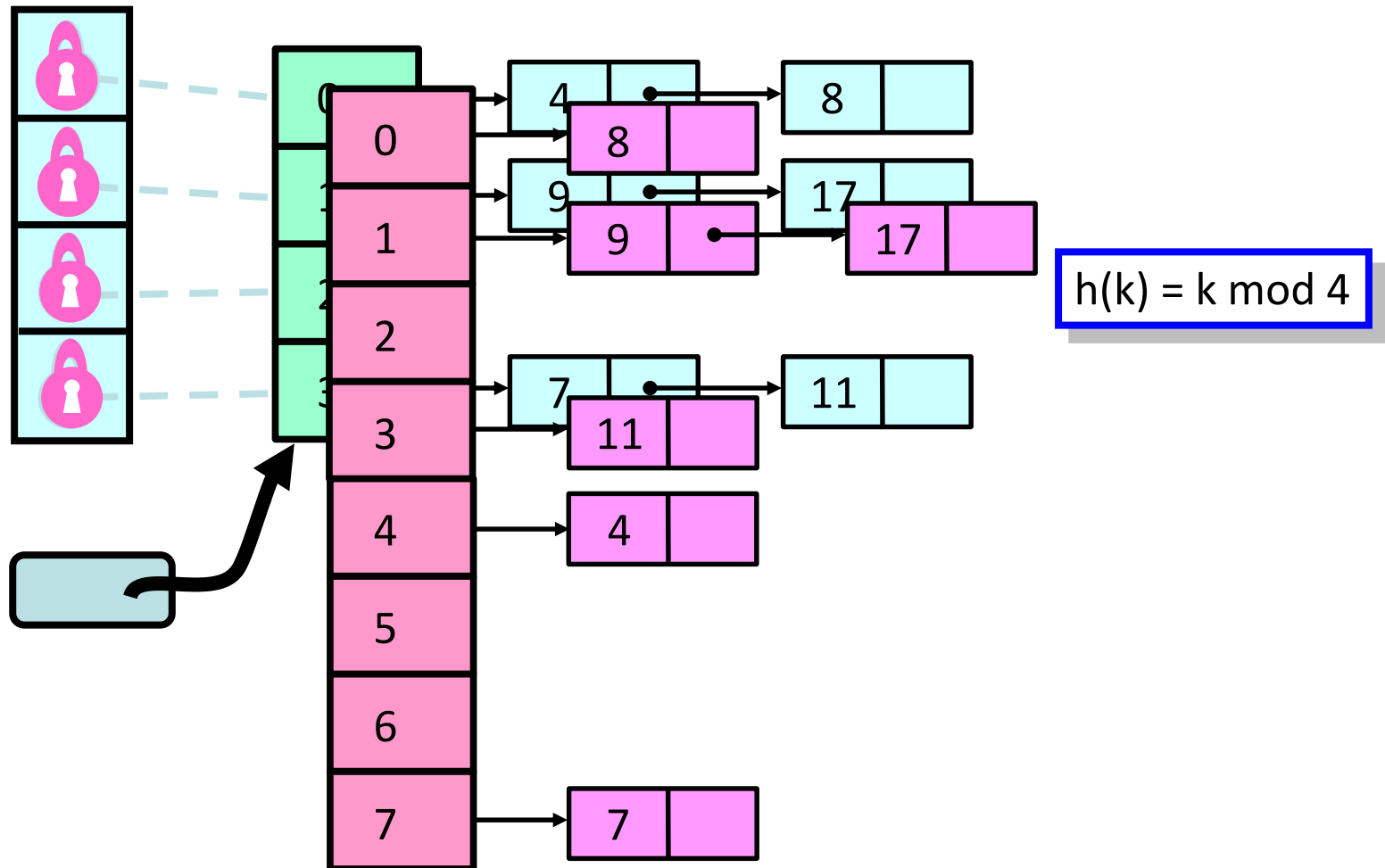
# Fine-Grained Locking: Resizing

- Acquire all locks in ascending order and make sure that the table reference didn't change between resize decision and lock acquisition!



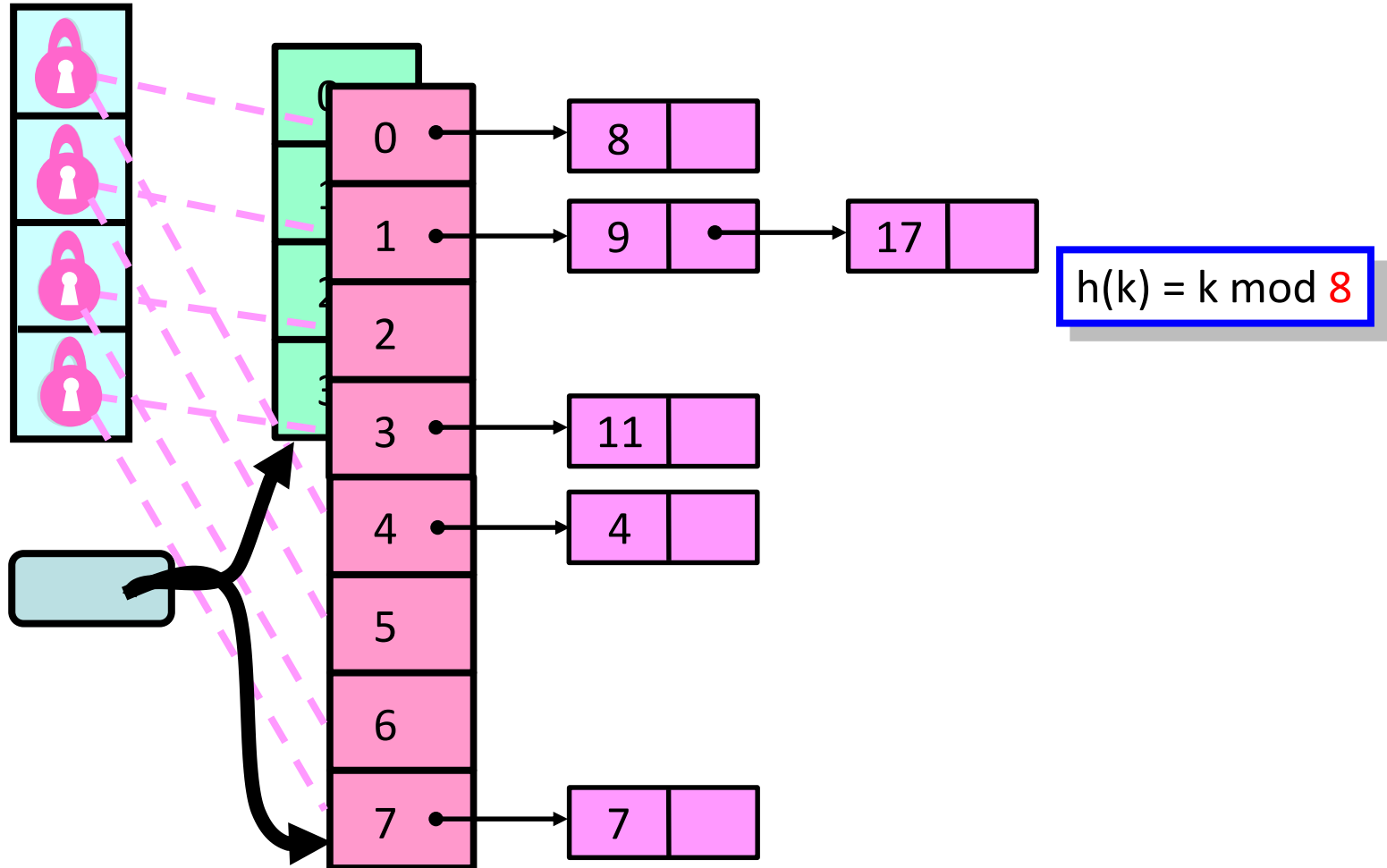
# Fine-Grained Locking: Resizing

- Allocate a new table and copy all elements



# Fine-Grained Locking: Resizing

- Stripe the locks: Each lock is now associated with two buckets
- Update the hash function and the table reference



# Observations

- We grow the table, but we don't increase the number of locks
  - Resizing the lock array is possible, but tricky...
- We use sequential lists (coarse-grained locking)
  - No lock-free list
  - If we're locking anyway, why pay?



## Fine-Grained Hash Set

```
public class FGHashSet {
```

```
    protected RangeLock[] lock;
```

```
    protected List[] table;
```

Array of locks

Array of buckets

```
    public FGHashSet(int capacity) {
```

```
        table = new List[capacity];
```

```
        lock = new RangeLock[capacity];
```

```
        for (int i = 0; i < capacity; i++) {
```

```
            lock[i] = new RangeLock();
```

```
            table[i] = new LinkedList();
```

```
        }
```

```
    }
```

Initially the same  
number of locks  
and buckets

## Fine-Grained Hash Set: Add Method

```
public boolean add(Object key) {  
    int keyHash = key.hashCode() % lock.length;  
    synchronized (lock[keyHash]) {  
        int tableHash = key.hashCode() % table.length;  
        return table[tableHash].add(key);  
    }  
}
```

Acquire the  
right lock

Call the add() method of  
the right bucket

## Fine-Grained Hash Set: Resize Method

```
public void resize(int depth, List[] oldTable) {  
    synchronized (lock[depth]) {  
        if (oldTable == this.table) {  
            int next = depth + 1;  
            if (next < lock.length)  
                resize(next, oldTable);  
            else  
                sequentialResize();  
        }  
    }  
}
```

**Resize() calls  
resize(0,this.table)**

**Acquire the next  
lock and check  
that no one else  
has resized**

**Recursively acquire  
the next lock**

**Once the locks are  
acquired, do the work**

## Fine-Grained Locks: Evaluation

- We can resize the table, but not the locks
- It is debatable whether method calls are constant-time in presence of contention ...
- Insight: The `contains()` method does not modify any fields
  - Why should concurrent `contains()` calls conflict?

# Read/Write Locks

```
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

**Return the associated read lock**

**Return the associated write lock**

# Lock Safety Properties

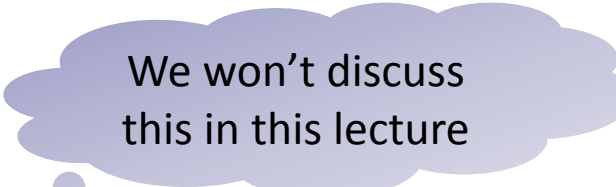
- No thread may acquire the write lock
  - while any thread holds the write lock
  - or the read lock
- No thread may acquire the read lock
  - while any thread holds the write lock
- Concurrent read locks OK
  
- This satisfies the following safety properties
  - If  $\text{readers} > 0$  then  $\text{writer} == \text{false}$
  - If  $\text{writer} = \text{true}$  then  $\text{readers} == 0$

# Read/Write Lock: Liveness

- How do we guarantee liveness?
  - If there are lots of readers, the writers may be locked out!
- Solution: FIFO Read/Write lock
  - As soon as a writer requests a lock, no more readers are accepted
  - Current readers “drain” from lock and the writers acquire it eventually

# Optimistic Synchronization

- What if the contains() method scans without locking...?
- If it finds the key
  - It is ok to return true!
  - Actually requires a proof...
- What if it doesn't find the key?
  - It may be a victim of resizing...
  - Get a **read lock** and try again!
  - This makes sense if it is expected(?) that the key is there and resizes are rare.
  - Better: Check if the table size is the same before and after the method call!



We won't discuss this in this lecture



# Stop The World Resizing

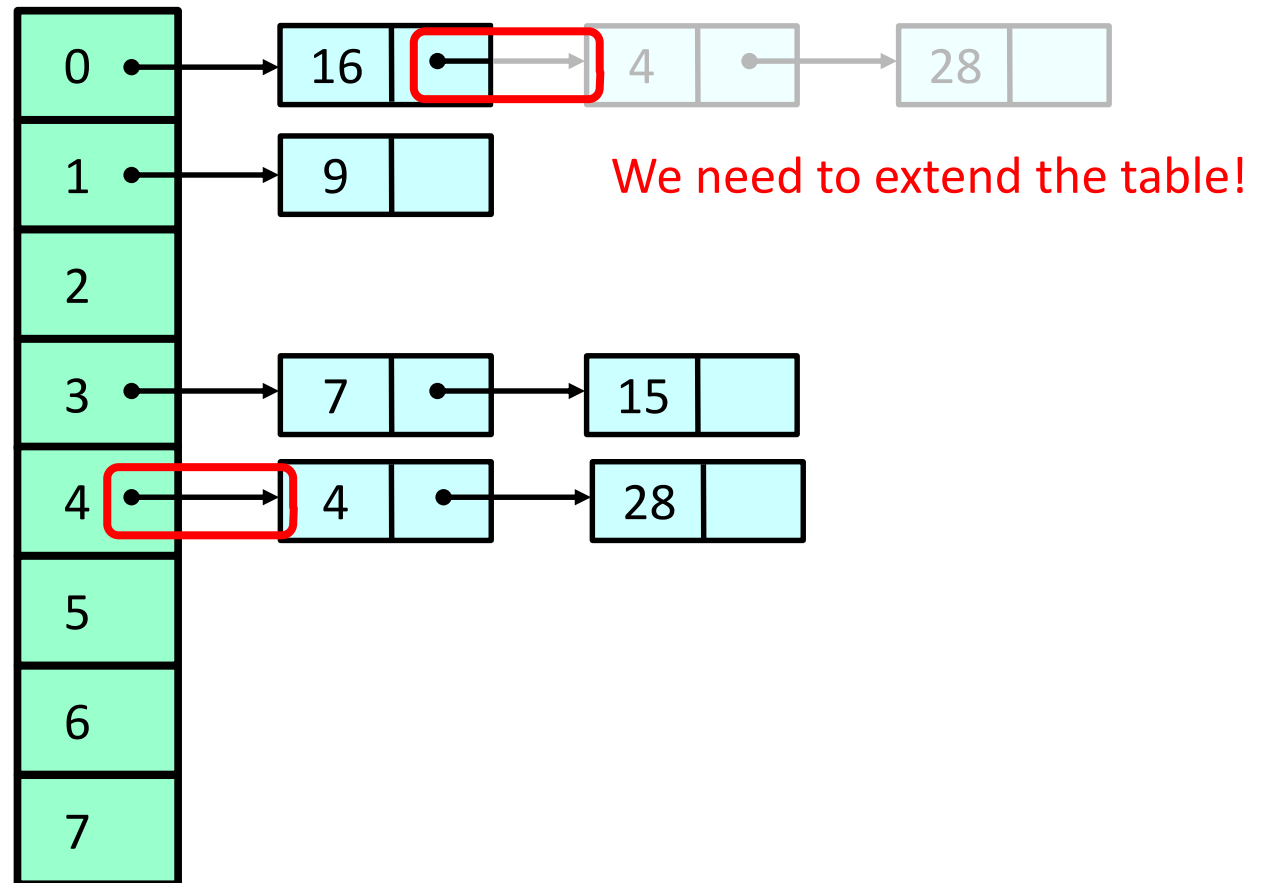
- The resizing we have seen up till now stops all concurrent operations
- Can we design a resize operation that will be incremental?
- We need to avoid locking the table...
- We want a **lock-free table** with **incremental resizing!**

A light blue thought bubble with a white question mark inside, connected to the end of the fourth bullet point by a thin line of three small circles.

How...?

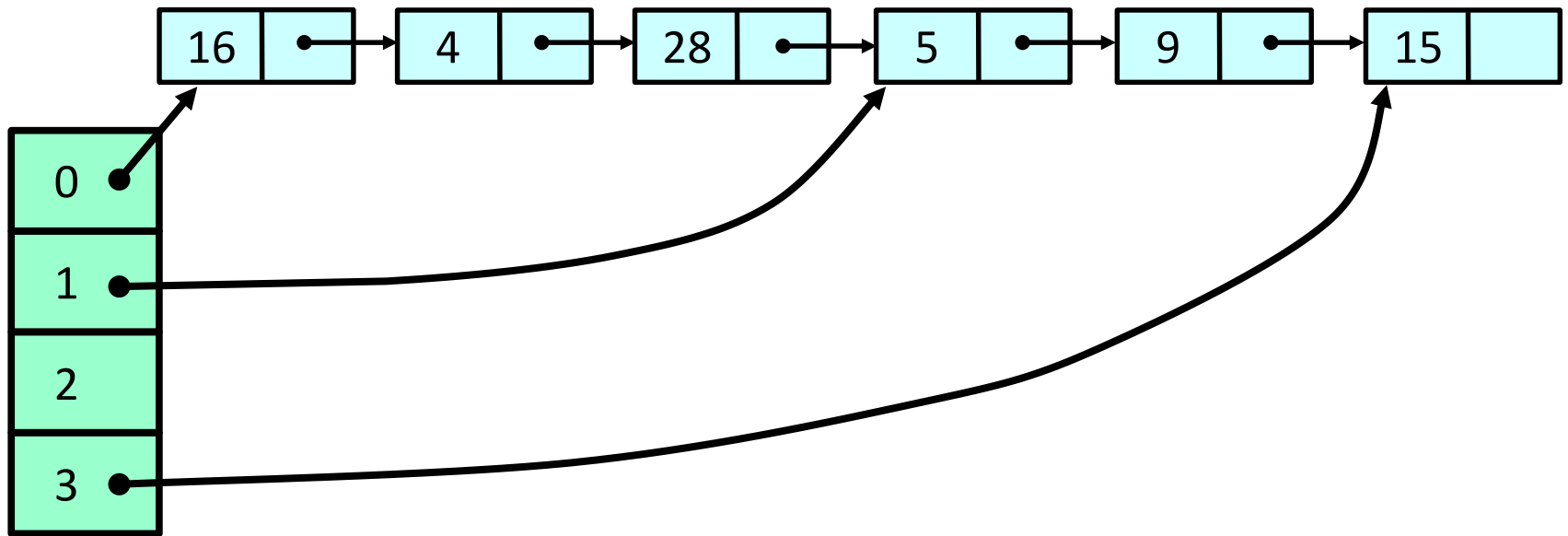
# Lock-Free Resizing Problem

- In order to remove and then add even a single item, “single location CAS” is not enough...



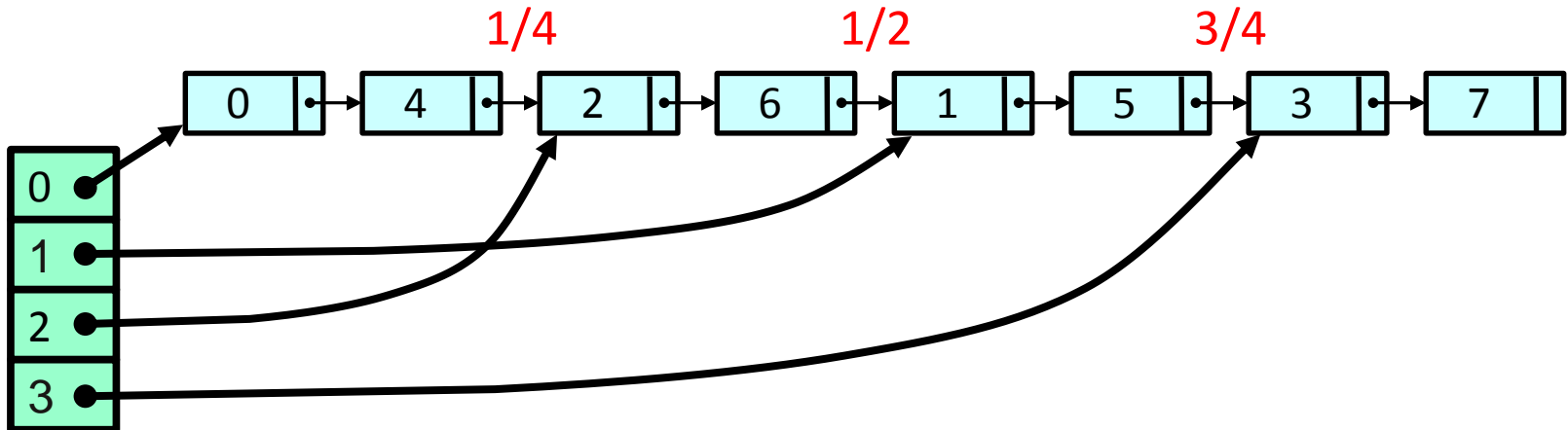
## Idea: Don't Move the Items

- Move the buckets instead of the items!
- Keep all items in a single lock-free list
- Buckets become “shortcut pointers” into the list



# Recursive Split Ordering

- Example: The items 0 to 7 need to be hashed into the table
- Recursively split the buckets in half:

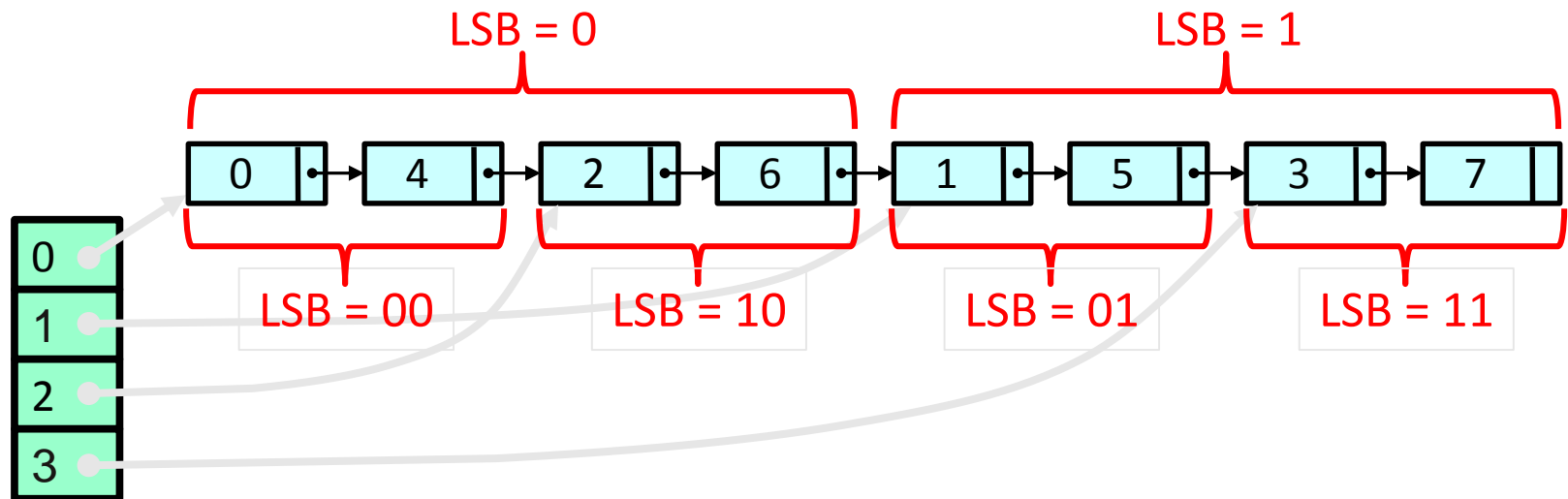


- The list entries are sorted in an order that allows recursive splitting



# Recursive Split Ordering

- Note that the least significant bit (LSB) is 0 in the first half and 1 in the other half! The second LSB determines the next pointers etc.

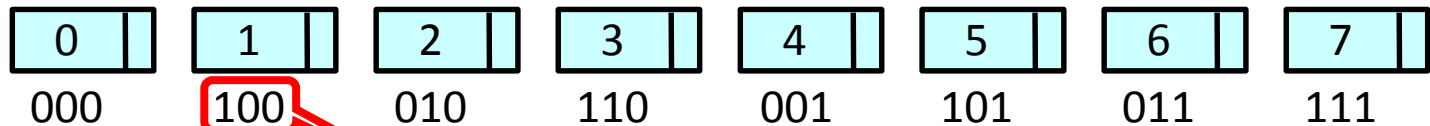


# Split-Order

- If the table size is  $2^i$ :
  - Bucket  $b$  contains keys  $k = b \bmod 2^i$
  - The bucket index consists of the key's  $i$  least significant bits
- When the table splits:
  - Some keys stay ( $b = k \bmod 2^{i+1}$ )
  - Some keys move ( $b+2^i = k \bmod 2^{i+1}$ )
- Whether a key moves is determined by the  $(i+1)^{\text{st}}$  bit
  - counting backwards

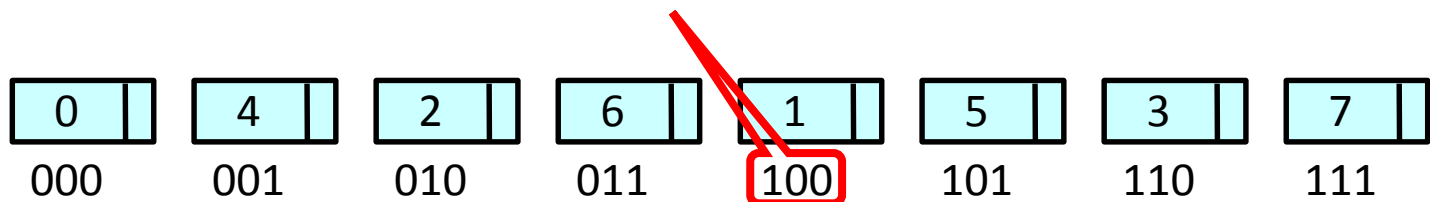
# A Bit of Magic

- We need to map the real keys to the split-order
- Look at the reversed binary representation of the keys and the indices
- The real keys:



- Split-order:

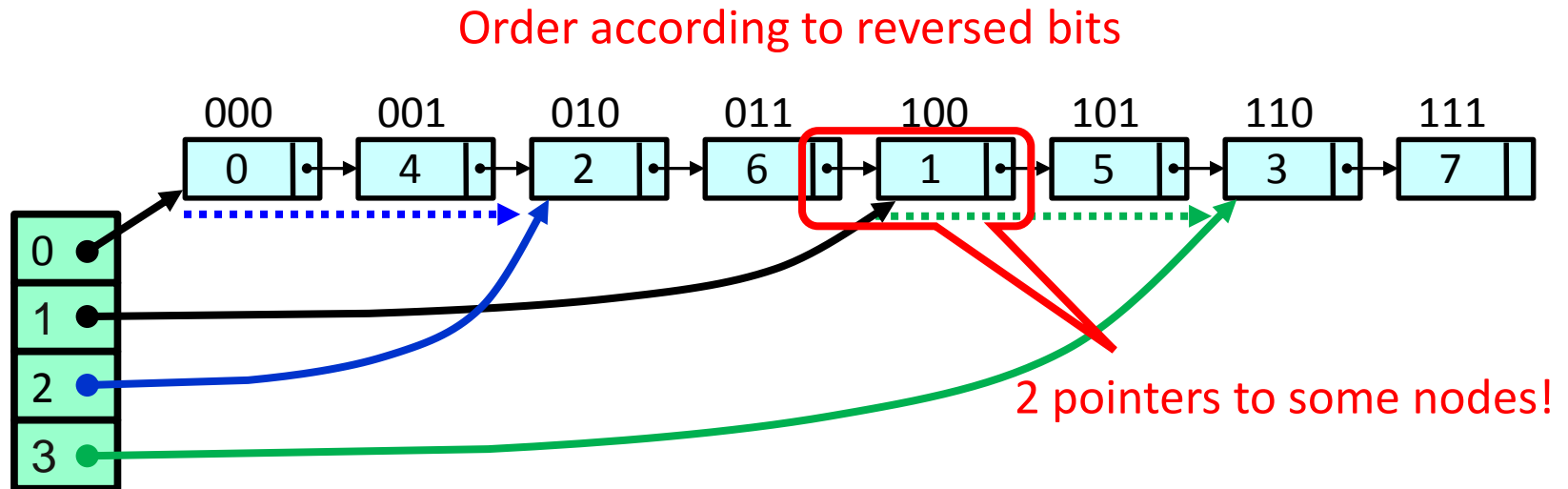
Real key 1 is at index 4!



- Just reverse the order of the key bits in order to get the index!

# Split Ordered Hashing

- After a resize, the new pointers are found by searching for the right index

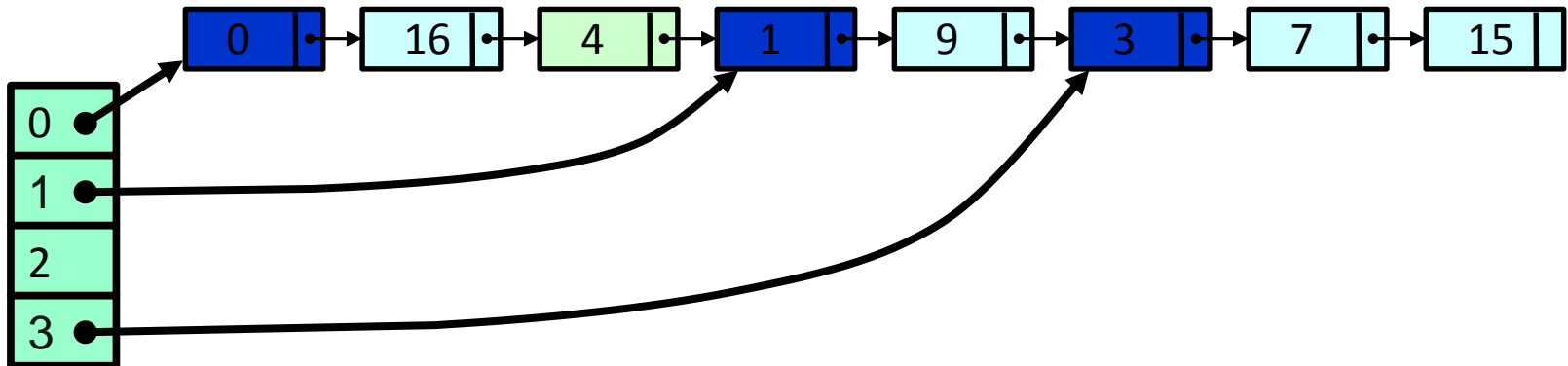


- A problem remains: How can we remove a node by means of a CAS if two sources point to it?



# Sentinel Nodes

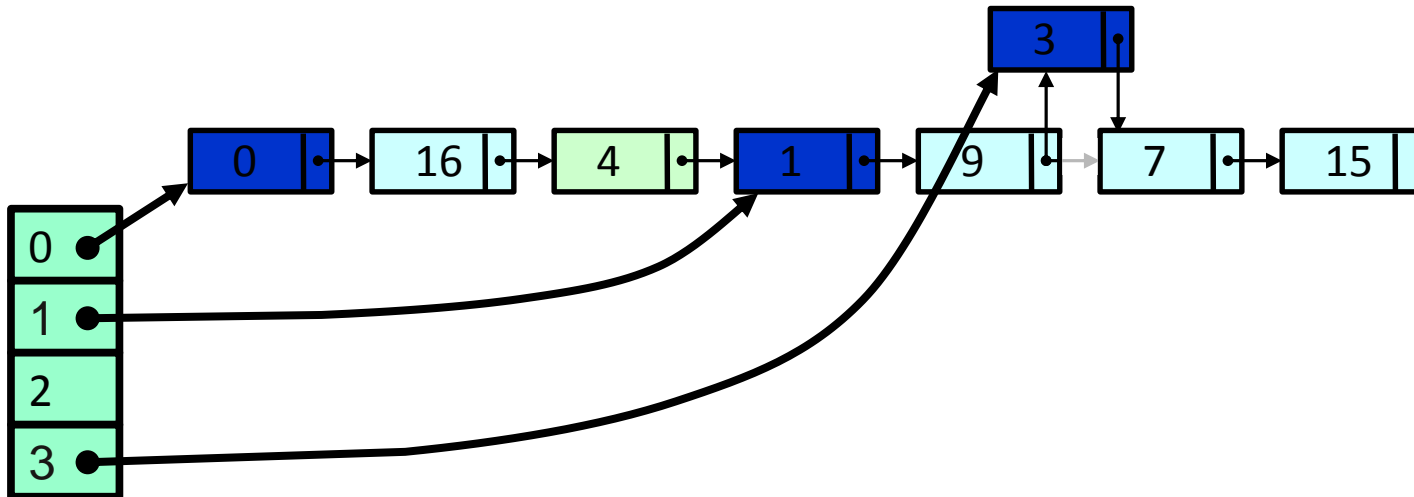
- Solution: Use a **sentinel node** for each bucket



- We want a sentinel key for  $i$ 
  - before all keys that hash to bucket  $i$
  - after all keys that hash to bucket  $(i-1)$

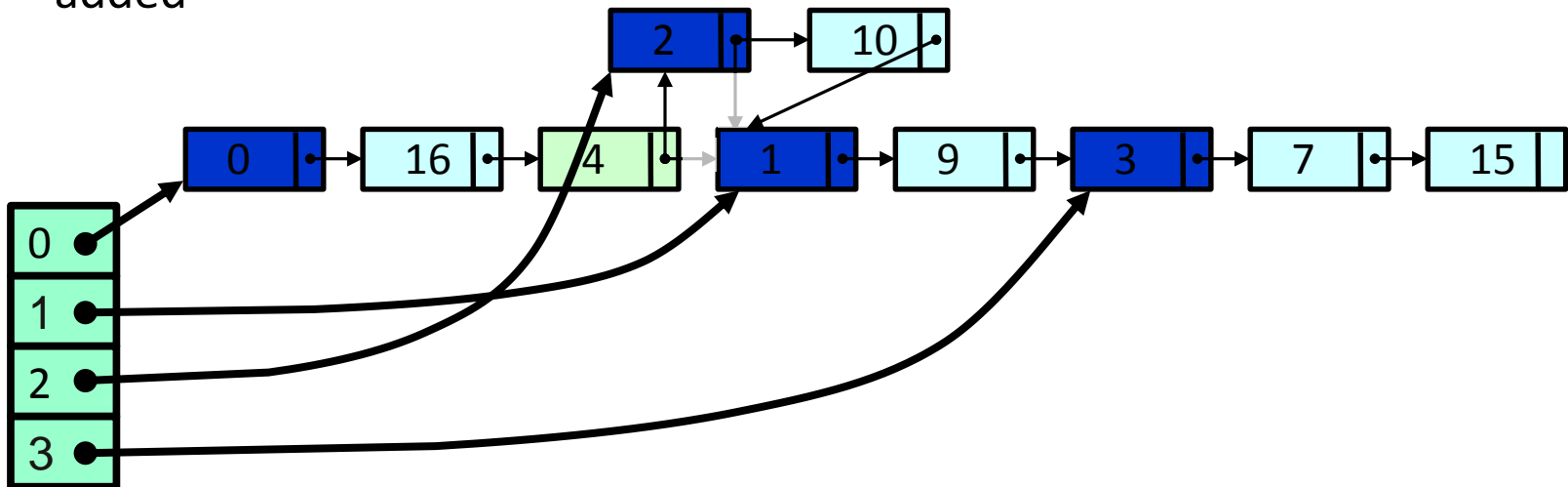
# Initialization of Buckets

- We can now split a bucket in a lock-free manner using two CAS() calls
- Example: We need to initialize bucket 3 to split bucket 1!



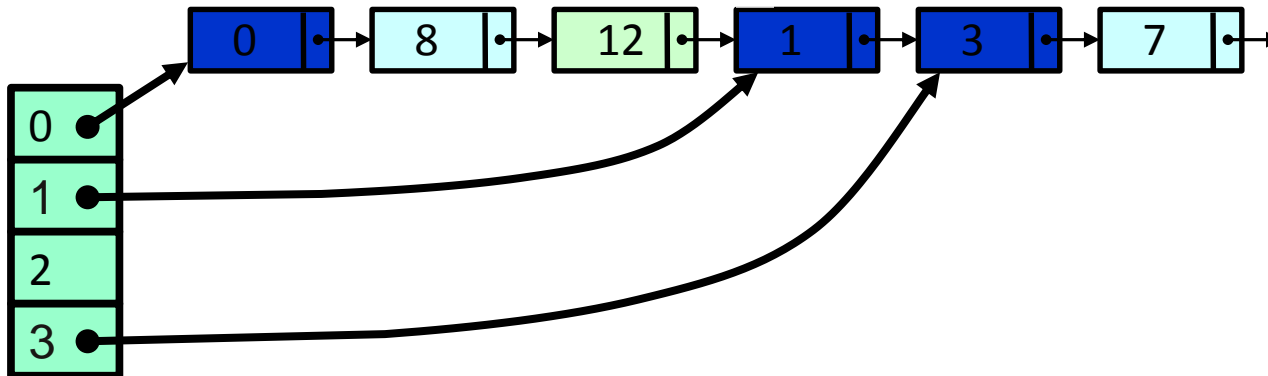
# Adding Nodes

- Example: Node 10 is added
- First, bucket 2 ( $= 10 \bmod 4$ ) must be initialized, then the new node is added



# Recursive Initialization

- It is possible that buckets must be initialized recursively
- Example: When node 7 is added, bucket 3 ( $= 7 \bmod 4$ ) is initialized and then bucket 1 ( $= 3 \bmod 2$ ) is also initialized



$n = \text{number of nodes}$

- Note that  $\approx \log n$  empty buckets may be initialized if one node is added, but the expected depth is **constant**!

## Lock-Free List

```
private int makeRegularKey(int key) {  
    return reverse(key | 0x80000000);  
}
```

**Set high-order bit  
to 1 and reverse**

```
private int makeSentinelKey(int key) {  
    return reverse(key);  
}
```

**Simply reverse  
(high-order bit is 0)**

# Split-Ordered Set

```
public class S0Set{  
    protected LockFreeList[] table;  
    protected AtomicInteger tableSize;  
    protected AtomicInteger setSize;  
  
    public S0Set(int capacity) {  
        table = new LockFreeList[capacity];  
        table[0] = new LockFreeList();  
        tableSize = new AtomicInteger(1);  
        setSize = new AtomicInteger(0);  
    }  
}
```

This is the lock-free list with minor modifications

Track how much of the table is used and the set size so that we know when to resize

Initially use 1 bucket and the size is 0

## Split-Ordered Set: Add

```
public boolean add(Object object) {  
    int hash = object.hashCode();  
    int bucket = hash % tableSize.get();  
    int key = makeRegularKey(hash);  
    LockFreeList list = getBucketList(bucket);  
    if (!list.add(object, key))  
        return false;  
    resizecheck();  
    return true;  
}
```

**Pick a bucket**  
**Non-sentinel**  
**split-ordered key**

**Get pointer to bucket's sentinel, initializing if necessary**

**Try to add with reversed key**

**Resize if necessary**

# Recall: Resizing & Initializing Buckets

- Decision to Resize
  - Divide the set size by the total number of buckets
  - If the quotient exceeds a threshold, double the table size up to a fixed limit
- Initializing Buckets
  - Buckets are originally null
  - If you encounter a null bucket, initialize it
  - Go to bucket's parent (earlier nearby bucket) and recursively initialize if necessary
  - Constant expected work per bucket!



## Split-Ordered Set: Initialize Bucket

```
public void initializeBucket(int bucket) {  
    int parent = getParent(bucket);  
    if (table[parent] == null)  
        initializeBucket(parent);  
    int key = makeSentinelKey(bucket);  
    table[bucket] = new  
        LockFreeList(table[parent], key);  
}
```

**Find parent,  
recursively  
initialize if needed**

**Prepare key for  
new sentinel**

**Insert sentinel if not present and  
return reference to rest of list**

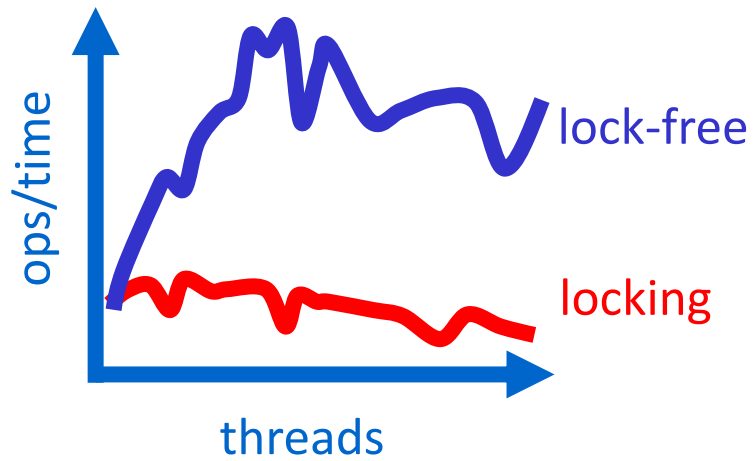
# Correctness

- Split-ordered set is a correct, linearizable, concurrent set implementation
- Constant-time operations!
  - It takes no more than  $O(1)$  items between two dummy nodes on average
  - Lazy initialization causes at most  $O(1)$  expected recursion depth in `initializeBucket()`

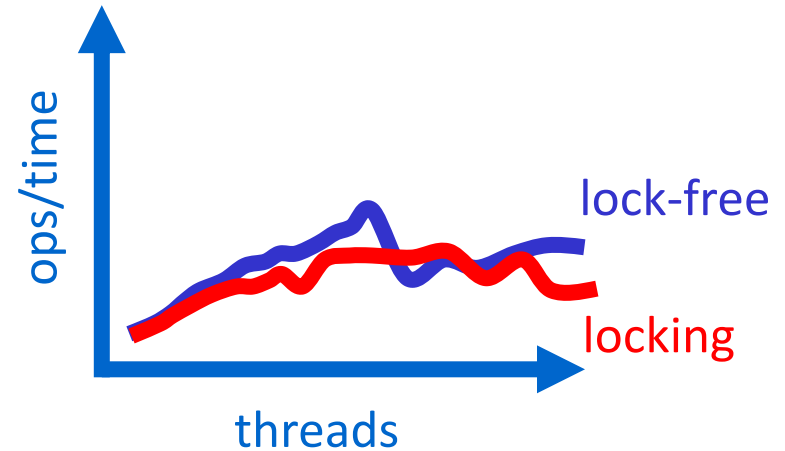
# Empirical Evaluation

- Evaluation has been performed on a 30-processor Sun Enterprise 3000
- Lock-Free vs. fine-grained optimistic locking (“Lea”)
- $10^6$  operations: 88% contains(), 10% add(), 2% remove()

Low load:

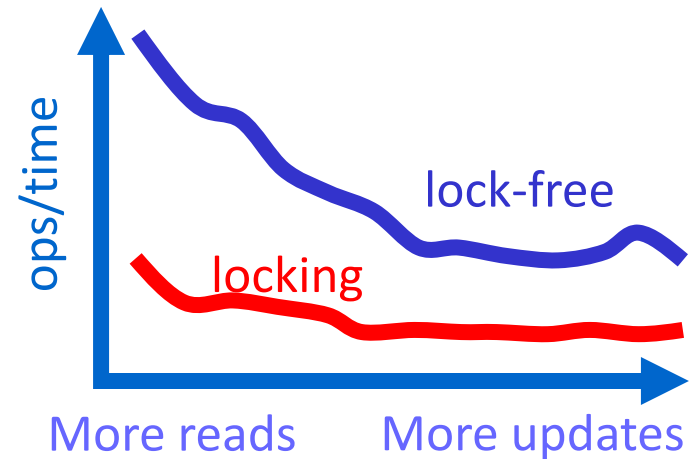
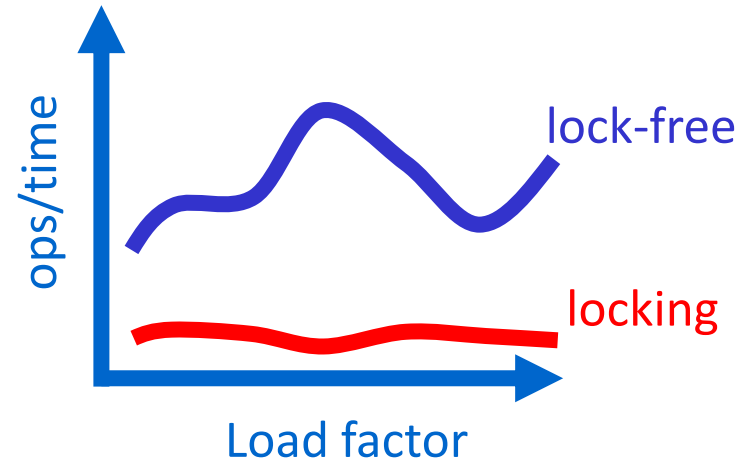


High load:



# Empirical Evaluation

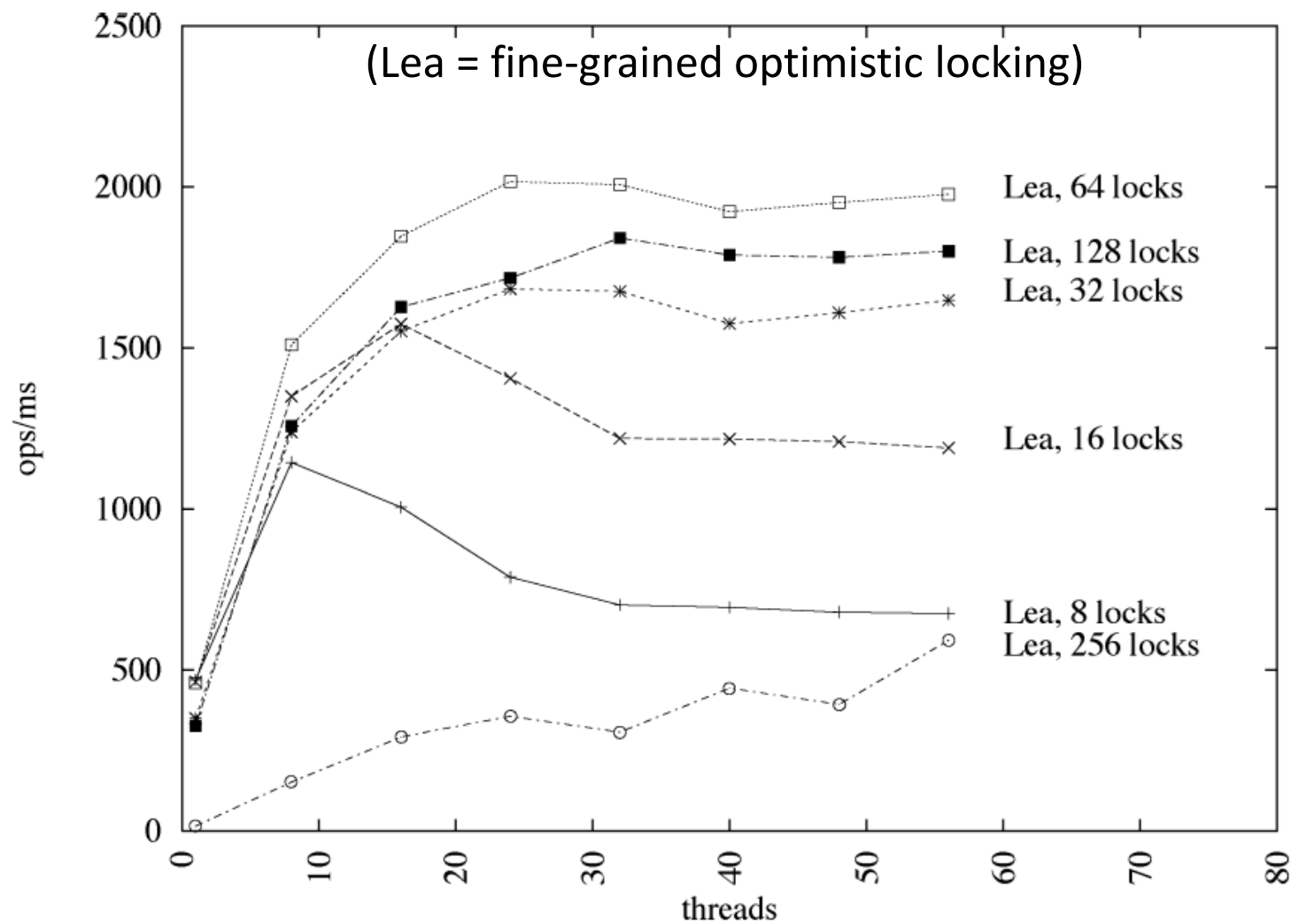
- Expected bucket length
  - The load factor is the capacity of the individual buckets
  
- Varying The Mix
  - Increasing the number of updates



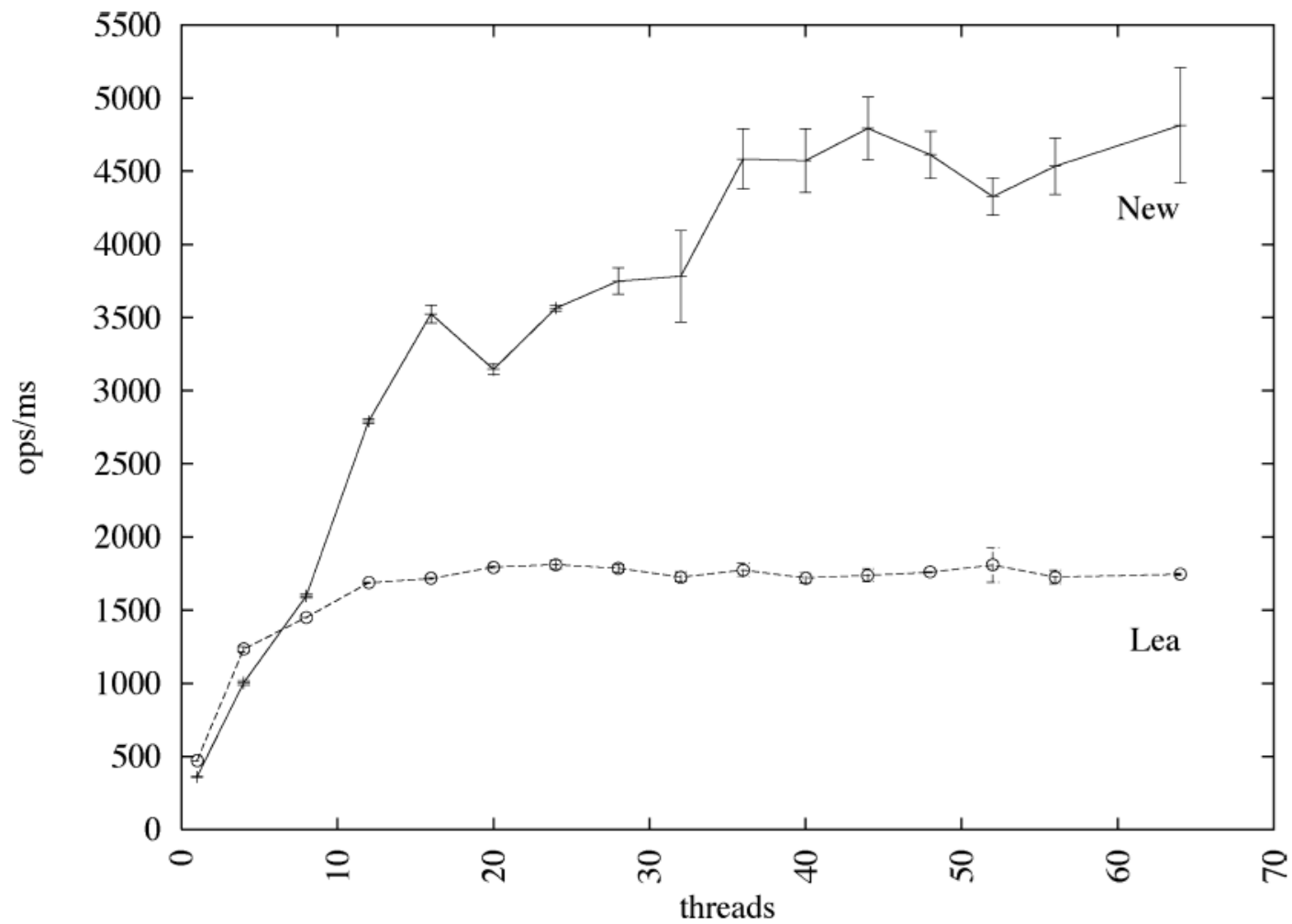
# Additional Performance

- Additionally, the following parameters have been analyzed:
  - The effects of the choice of locking granularity
  - The effects of the bucket size

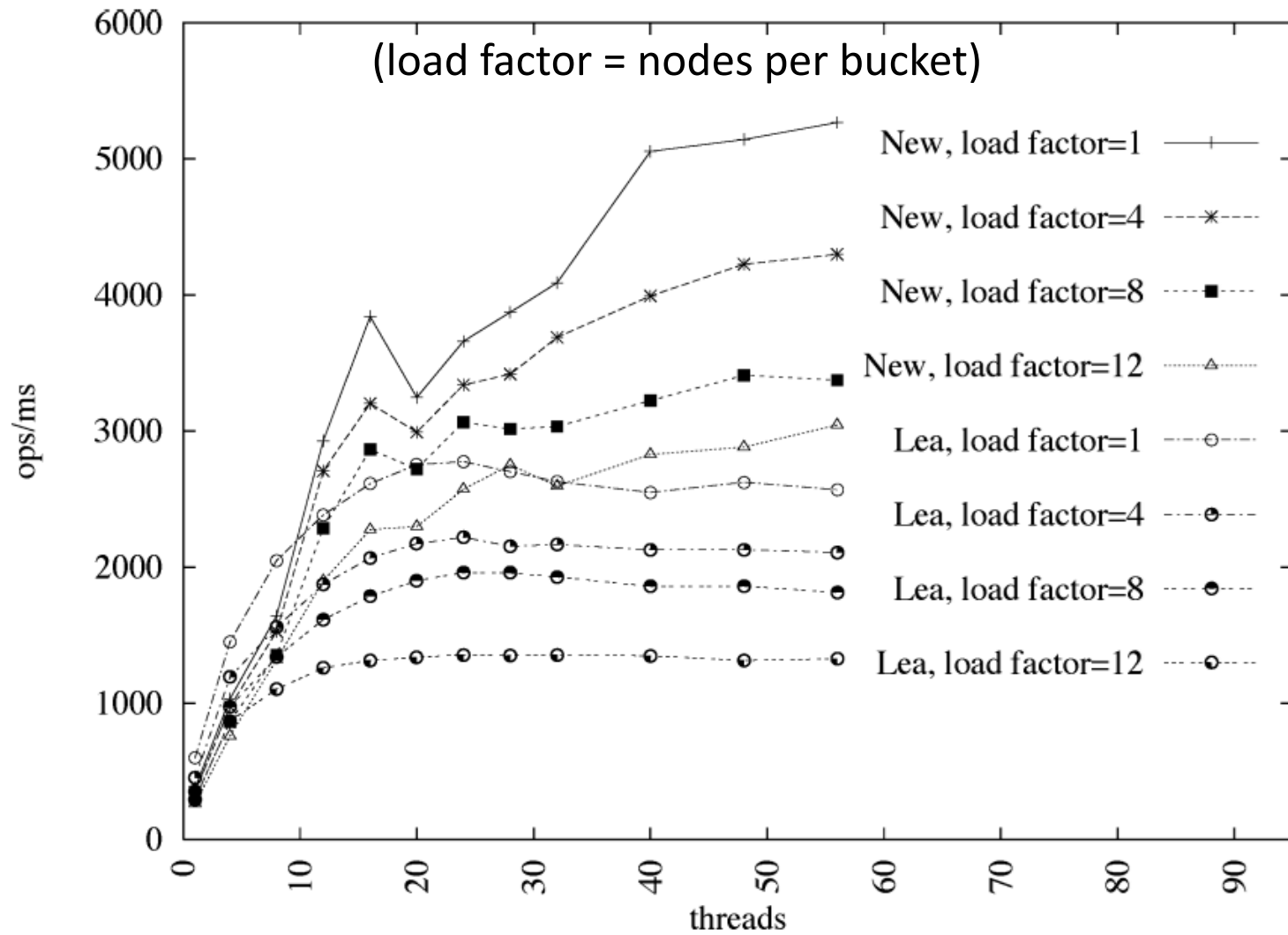
# Number of Fine-Grain Locks



# Lock-free vs. Locks

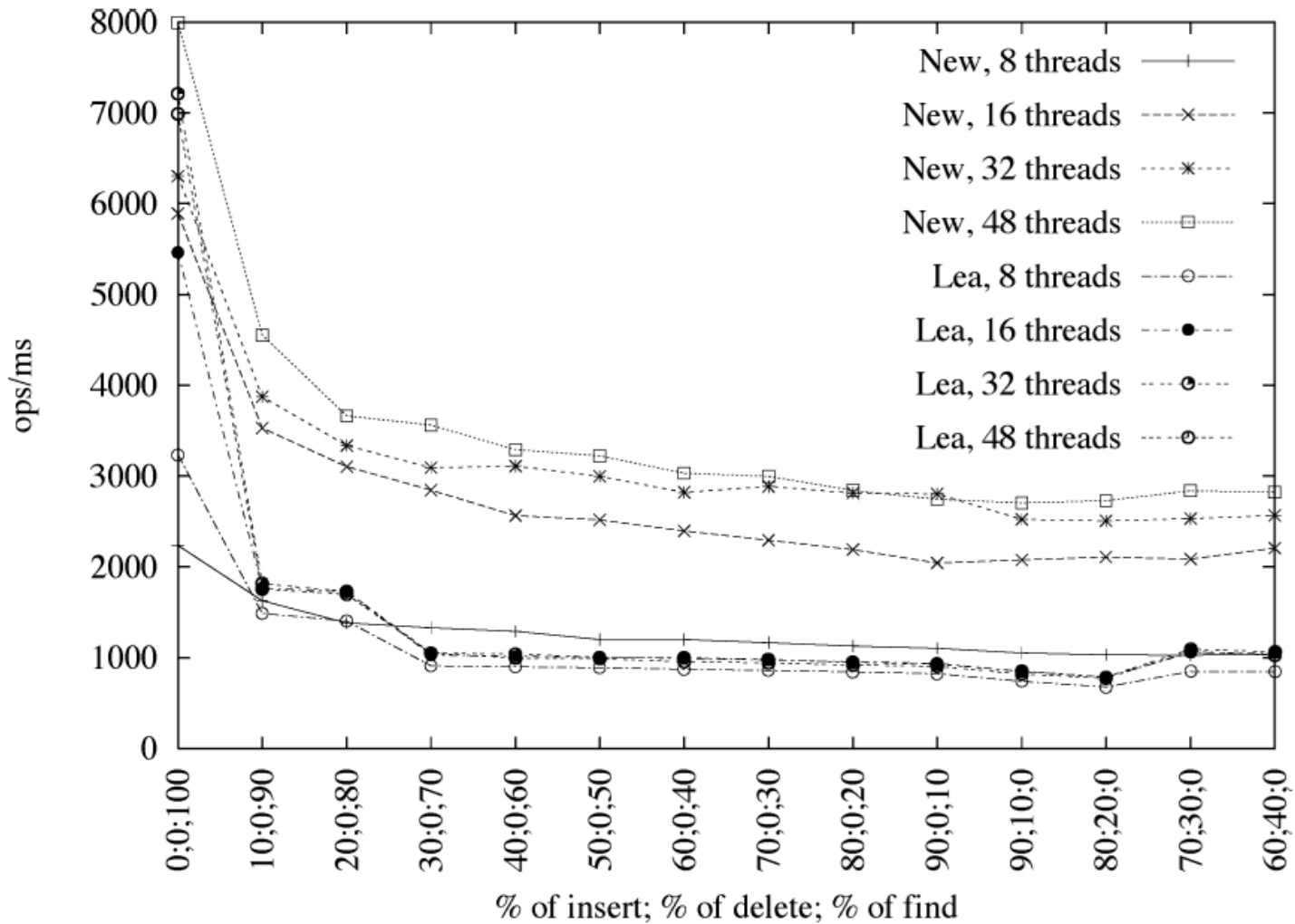


# Hash Table Load Factor





# Varying Operations



# Summary

- We talked about techniques to deal with concurrency in linked lists
  - Hand-over-hand locking
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- Then we talked about hashing
  - Fine-grained locking
  - Recursive split ordering

# Credits

- The first lock-free list algorithms are credited to John Valois, 1995.
- The lock-free list algorithm discussed in this lecture is a variation of algorithms proposed by Harris, 2001, and Michael, 2002.
- The lock-free hash set based on split-ordering is by Shalev and Shavit, 2006.

# *That's all!*

*Questions & Comments?*



*Roger Wattenhofer*