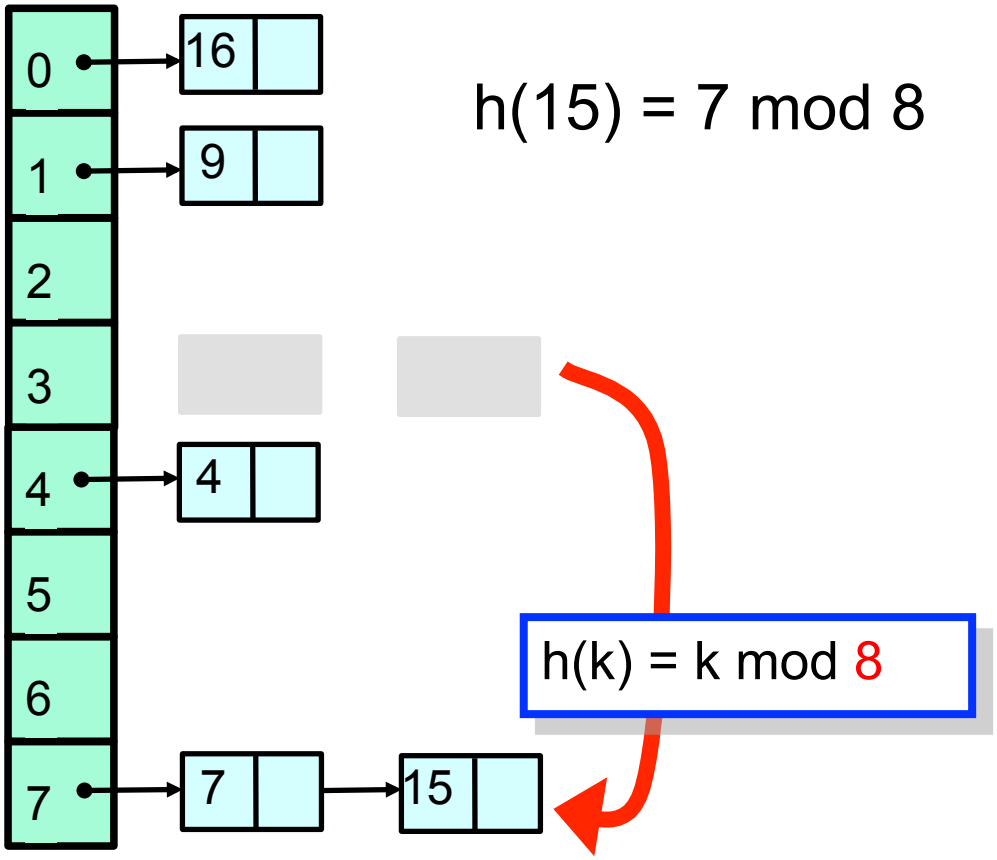
CS-206 Concurrency

Lecture 9
Concurrent
Hash Tables



Spring 2015

Prof. Babak Falsafi

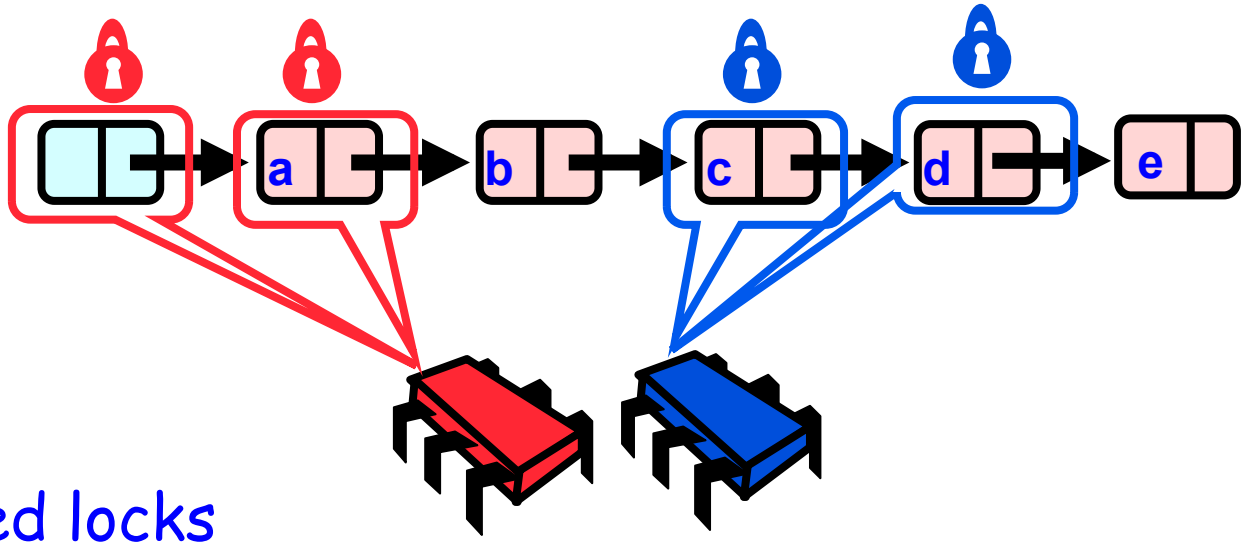
parsa.epfl.ch/courses/cs206/

Where are We?

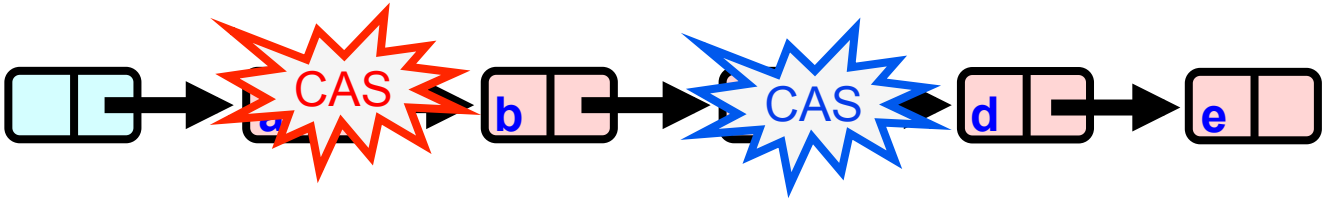
		Lecture & Lab		
M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr		29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Concurrent hash tables
  - Coarse-grained locking
  - Fine-grained locking
  - Lock-free
- Next week
  - Futures
  - GPUs

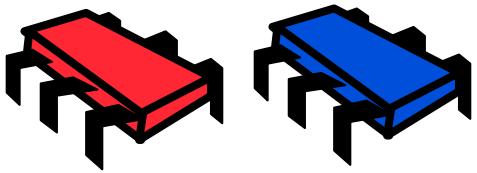
Recall: Fine-Grained locking vs. Lock-Free Lists



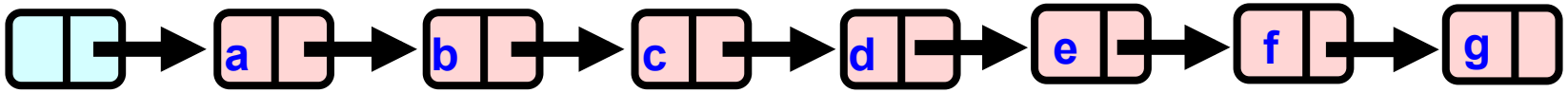
Fine-grained locks
(two locks, search then act)



Lock-free
(search and act)

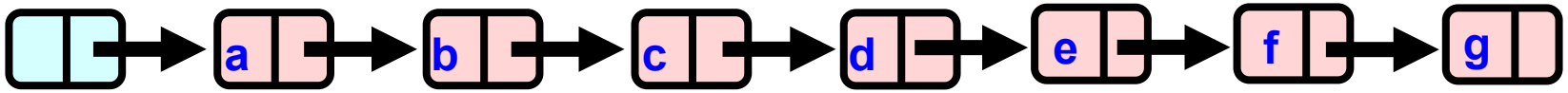


Example: Lock-free Lists



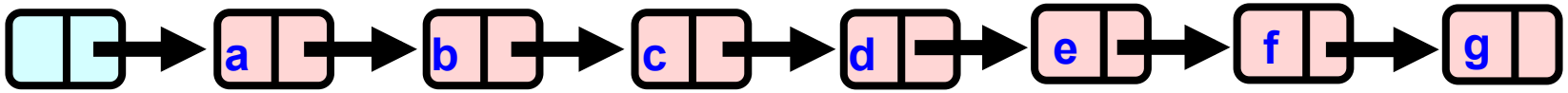
- Assume 4 threads removing a, c, e, g
  - Time to advance pointer: $T_{pointer}$ (~cache hit)
  - Time to perform a CAS: T_{CAS}
  - Time to remove: T_{remove}
- How long does it take to remove these four nodes concurrently?
- Answer: slowest thread is the one removing g
  - Worst case: $7 * T_{pointer} + T_{CAS} + T_{remove}$

Example: Fine-Grain Locked List



- Assume 4 threads removing a, c, e, g
  - Time to advance pointer: $T_{pointer}$ (~cache hit)
  - Time to grab locks: T_{lock} (~T_{CAS})
  - Time to remove node: T_{remove}
- How long does it take to remove these four nodes concurrently in the best case?
- Answer: threads traverse the list in the reverse order
  - Best case order: g, e, c, a ➔ $7 * T_{pointer} + 8*T_{CAS} + T_{remove}$

Example: Fine-Grain Locked List



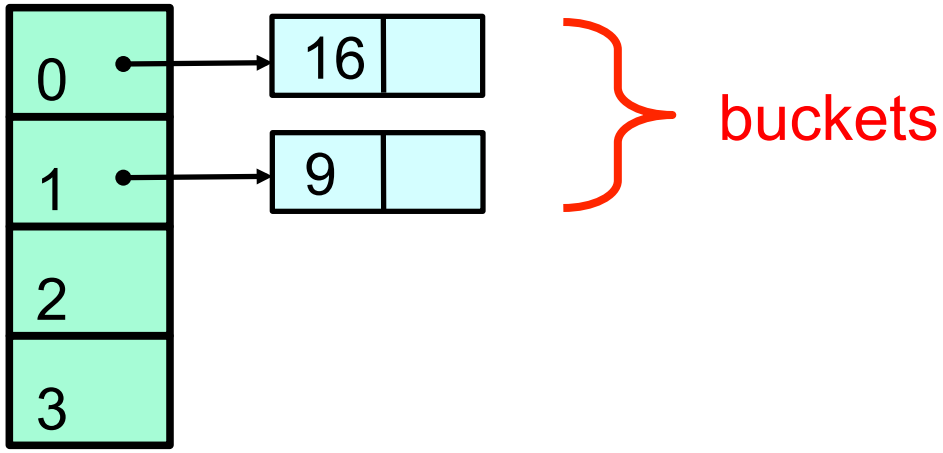
- Worst case order: a, c, e, g
  - g will wait for everyone to traverse and remove their nodes
  - a → $T_{pointer} + 2*T_{CAS} + T_{remove}$
  - c → $T_a + 2*T_{pointer} + 2*T_{CAS} + T_{remove}$
  - e → $T_c + 2*T_{pointer} + 2*T_{CAS} + T_{remove}$
  - g = Total time ~ $4*(2*(T_{pointer} + T_{CAS}) + T_{remove})$

- It gets worse with more threads and longer lists

Now, Hash Tables: Where are they used?

- Everywhere
- Earlier internet-scale peer-to-peer systems
  - Freenet, BitTorrent, Napster,…..
- Large-scale IT companies (e.g., Amazon, Google)
  - Data organized as key-value stores

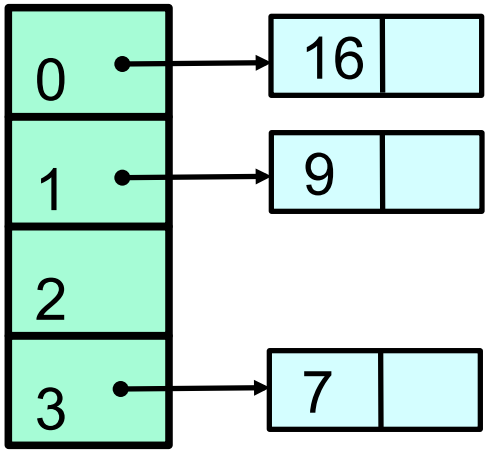
Sequential Closed Hash Map



2 Items

h(k) = k mod 4

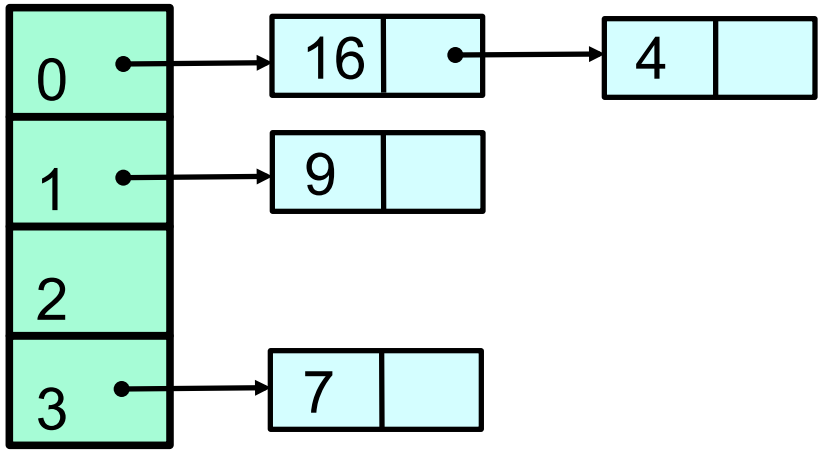
Add an Item



3 Items

h(k) = k mod 4

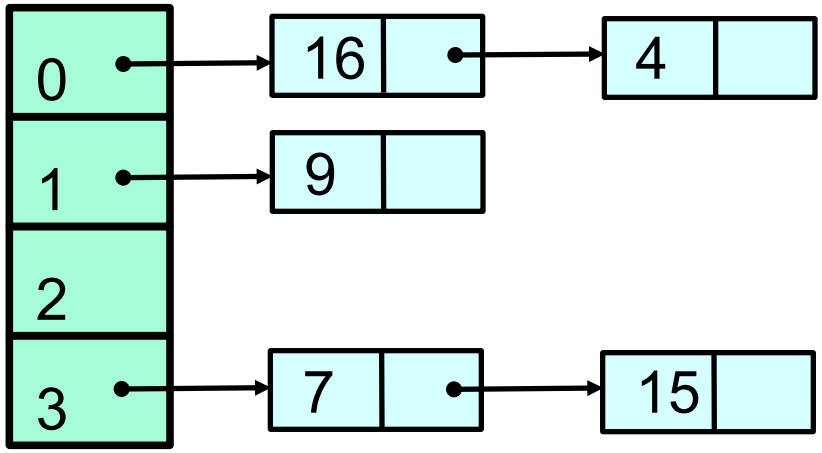
Add Another: Collision



4 Items

h(k) = k mod 4

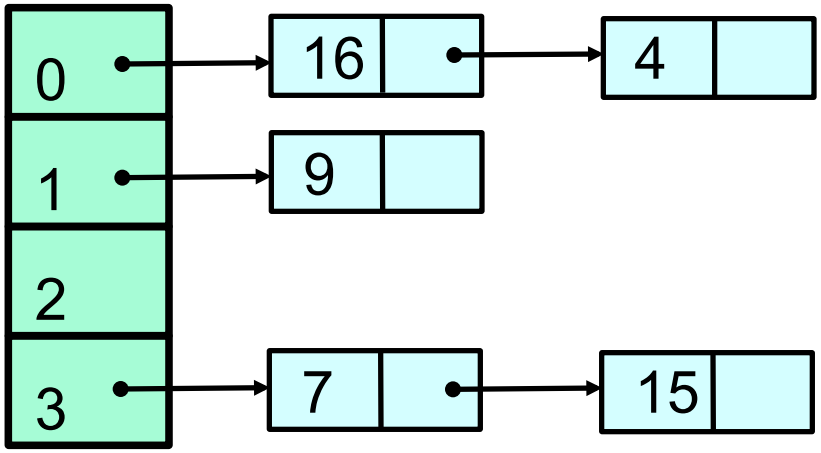
More Collisions



5 Items

h(k) = k mod 4

More Collisions

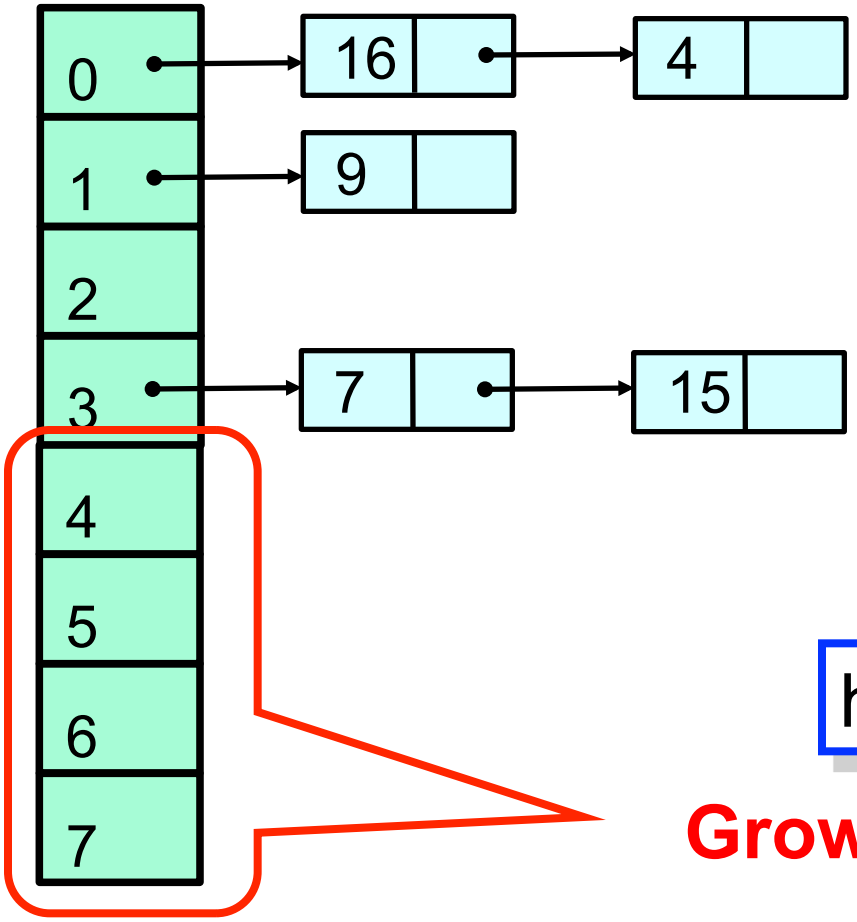


Problem:
buckets getting too long

5 Items

h(k) = k mod 4

Resizing

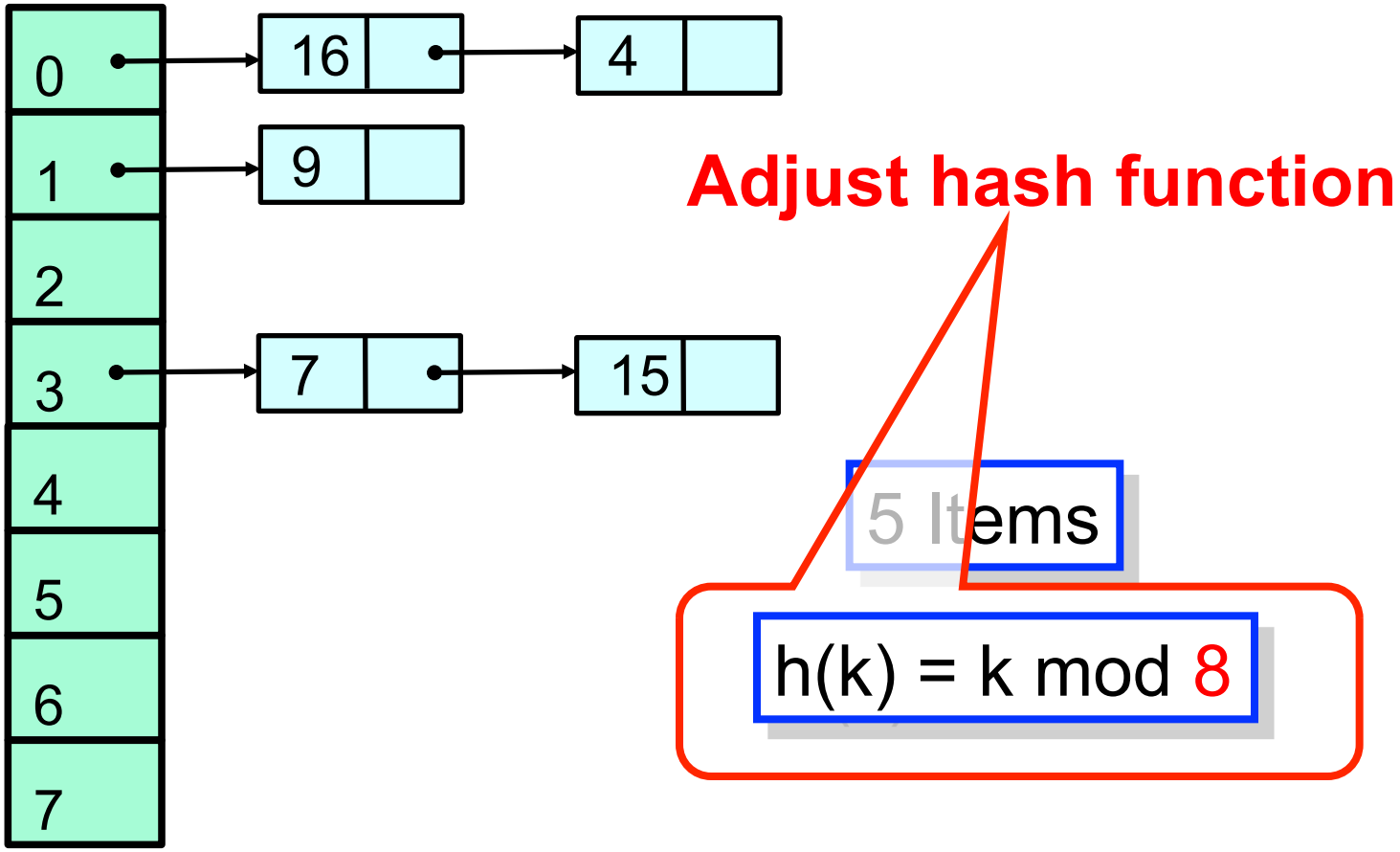


5 Items

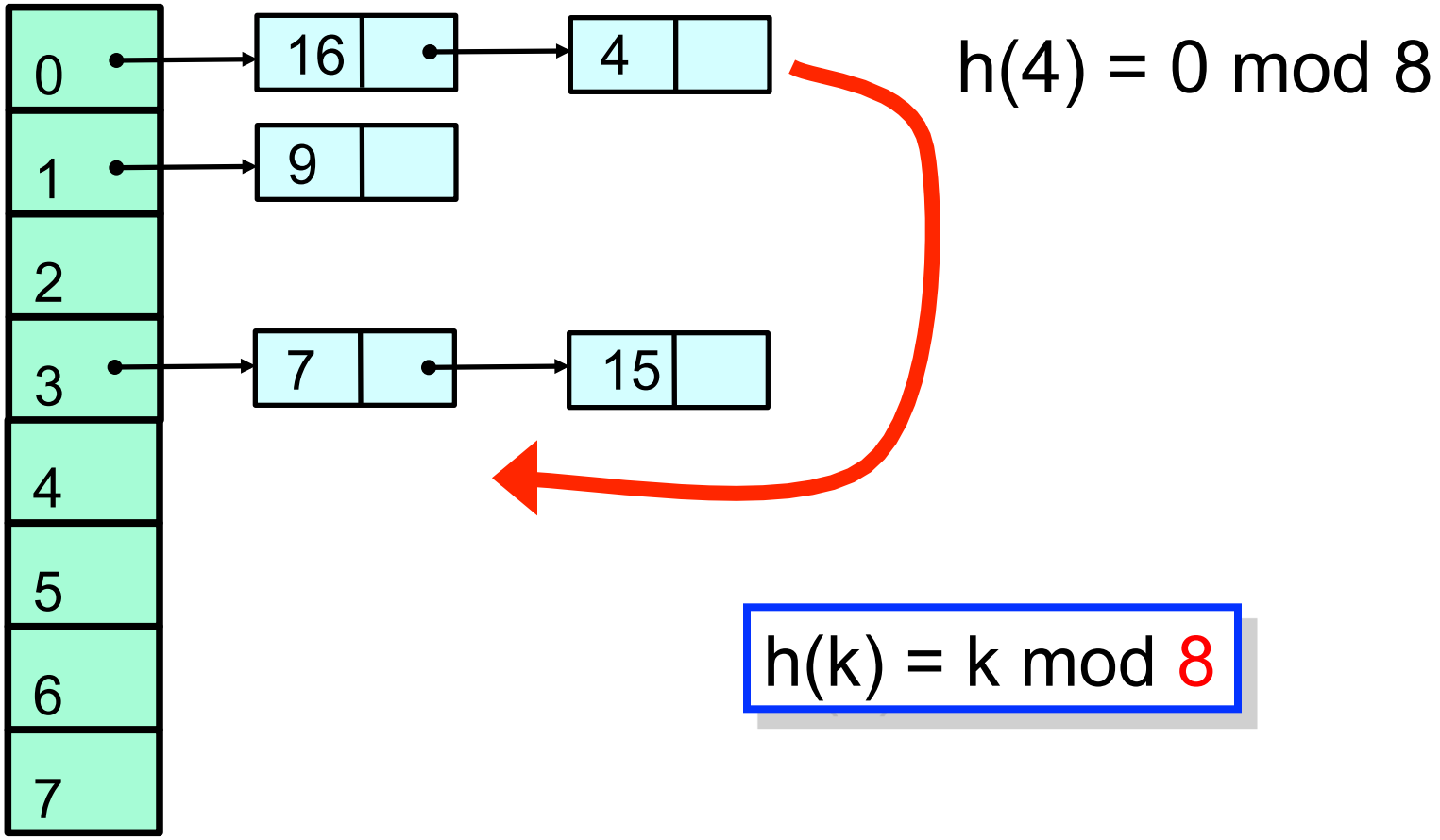
h(k) = k mod 8

Grow the array

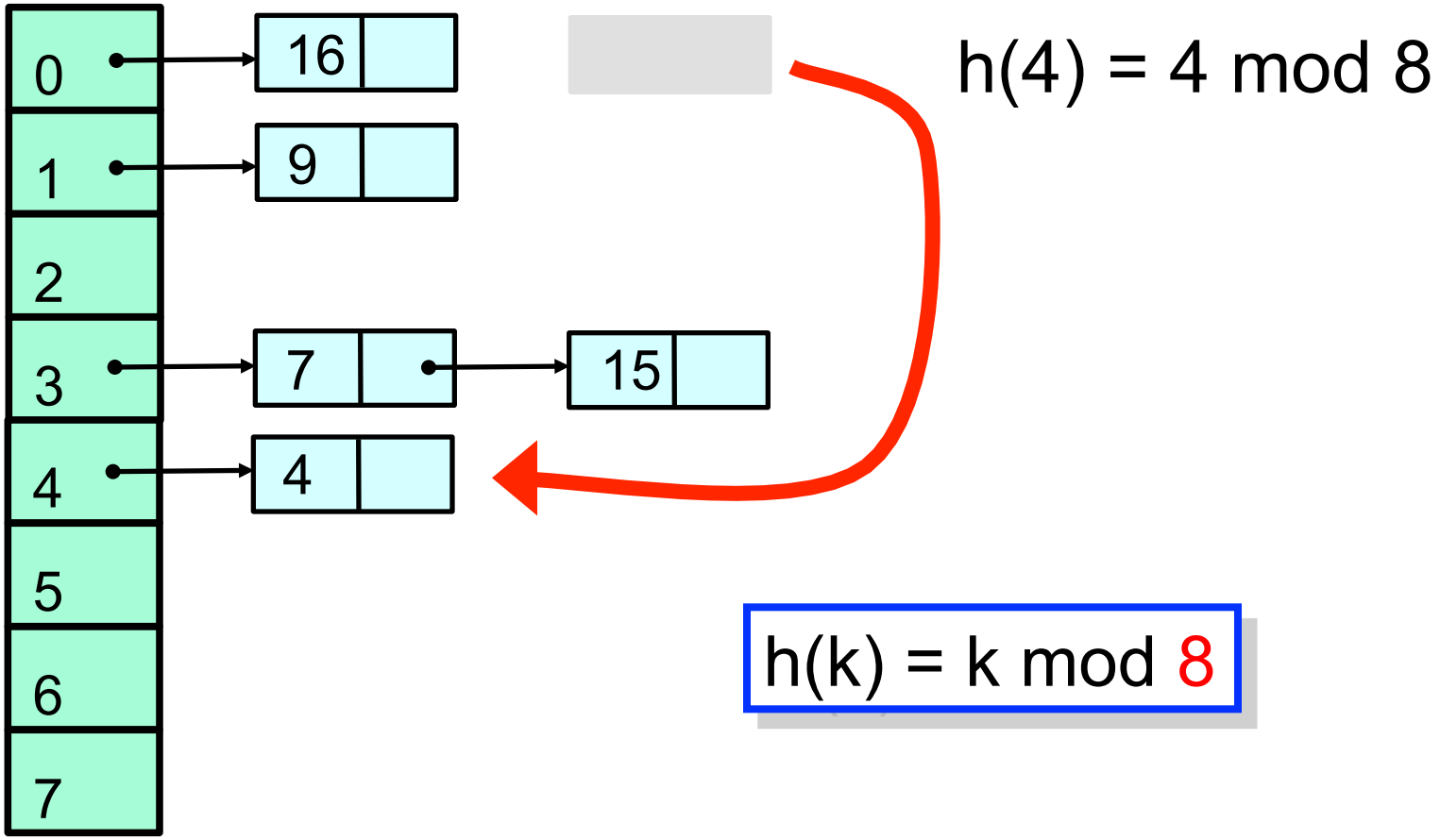
Resizing



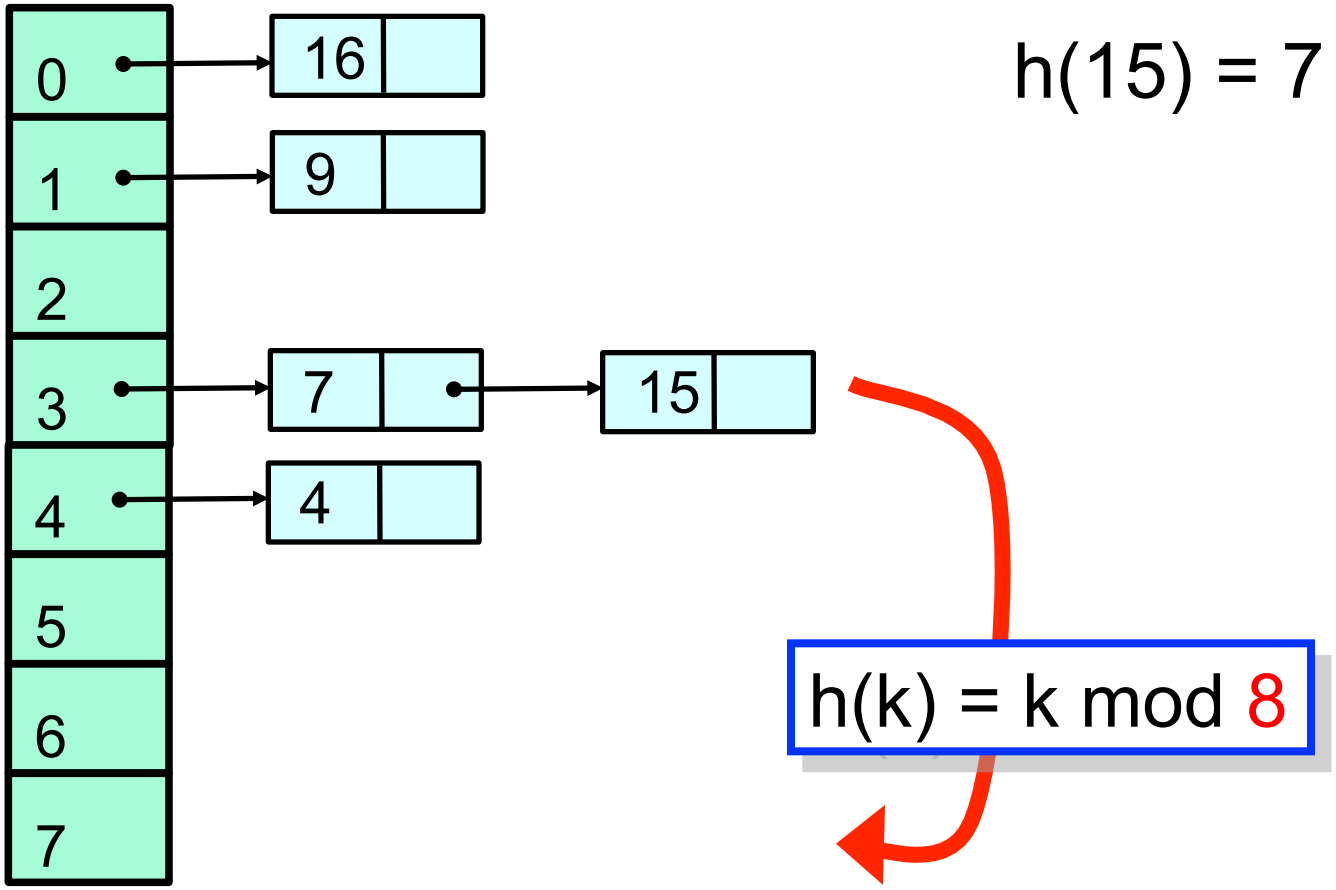
Resizing



Resizing

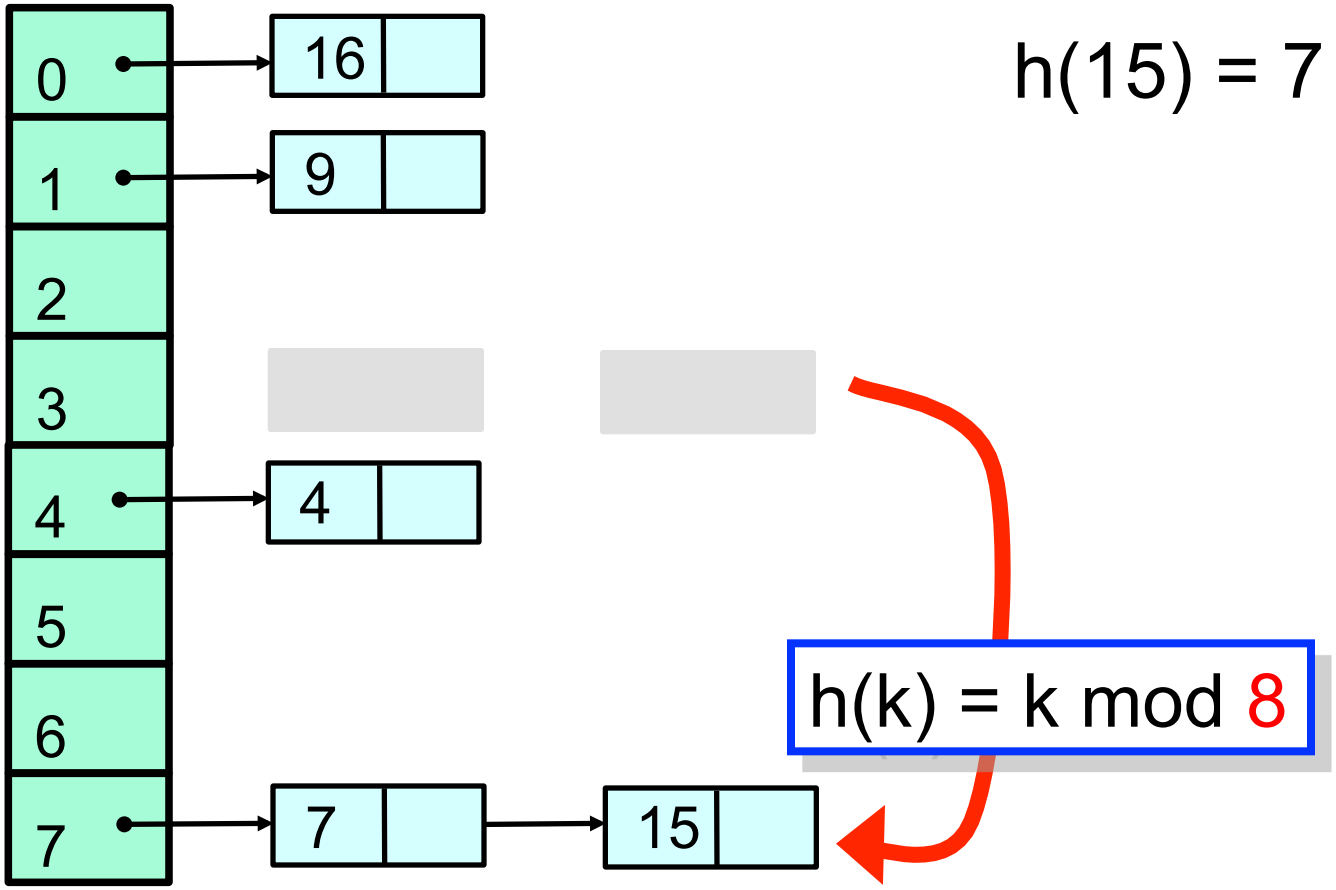


Resizing



h(15) = 7 mod 8

Resizing



Fields

```
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();
  }
…
```

Array of lock-free lists

Constructor

```
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();
  }
…
```

Initial size

Constructor

```
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();
  }
…
```

Allocate memory

Constructor

```
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();
  }
…
```

Initialization

Add Method

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

Add Method

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

Use object hash code to pick a bucket

Add Method

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

Call bucket's add() method

No Brainer?

- We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation
- What's not to like?

No Brainer?

- We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation
- What's not to like?

We don't know how to resize …

Is Resizing Necessary?

- Constant-time method calls require
  - Constant-length buckets
  - Table size proportional to set size
  - As set grows, must be able to resize

Set Method Mix

- Typical load
  - 90% contains()
  - 9% add ()
  - 1% remove()
- Growing is important
- Shrinking not so much

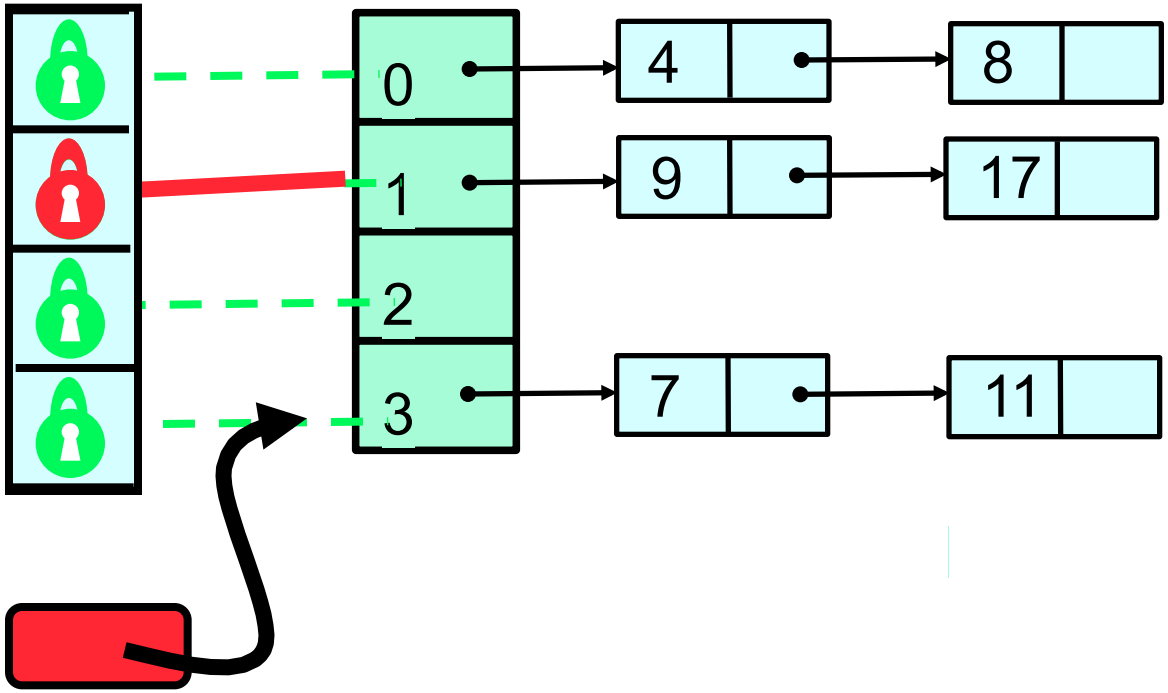
When to Resize?

- Many reasonable policies. Here's one.
- Pick a threshold on num of items in a bucket
- Global threshold
  - When ≥ ¼ buckets exceed this value
- Bucket threshold
  - When any bucket exceeds this value

Coarse-Grained Locking

- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck

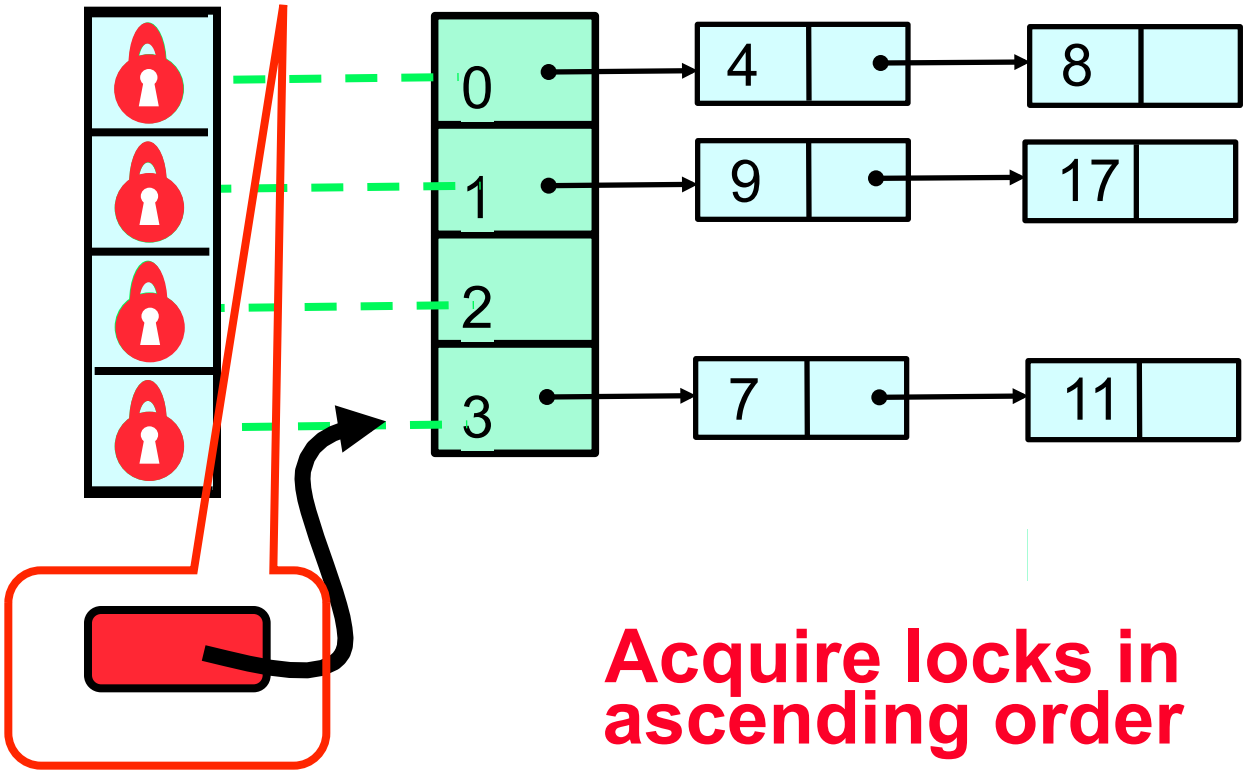
Fine-grained Locking



Each lock associated with one bucket

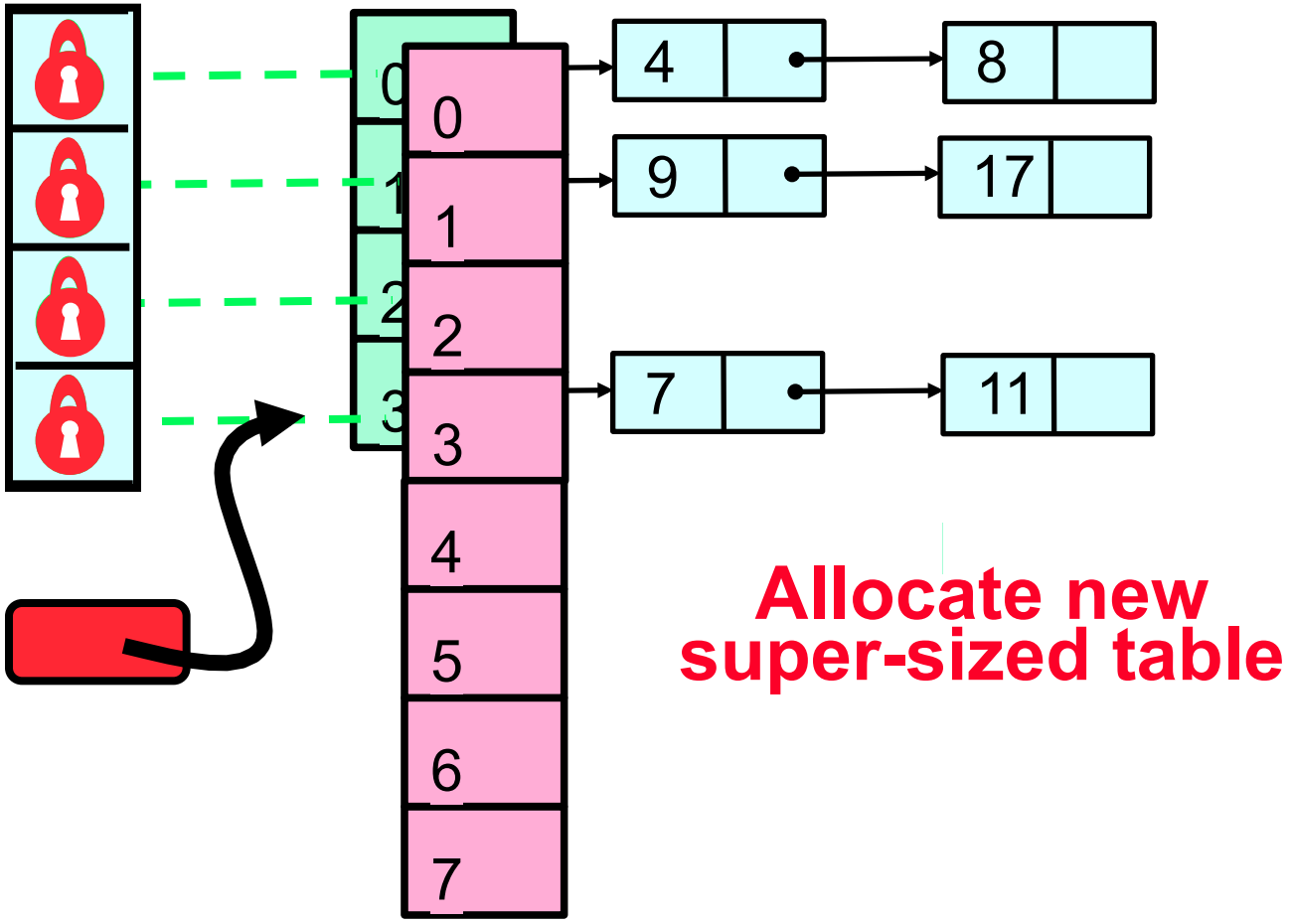
Resize This

Make sure table reference didn't change between resize decision and lock acquisition

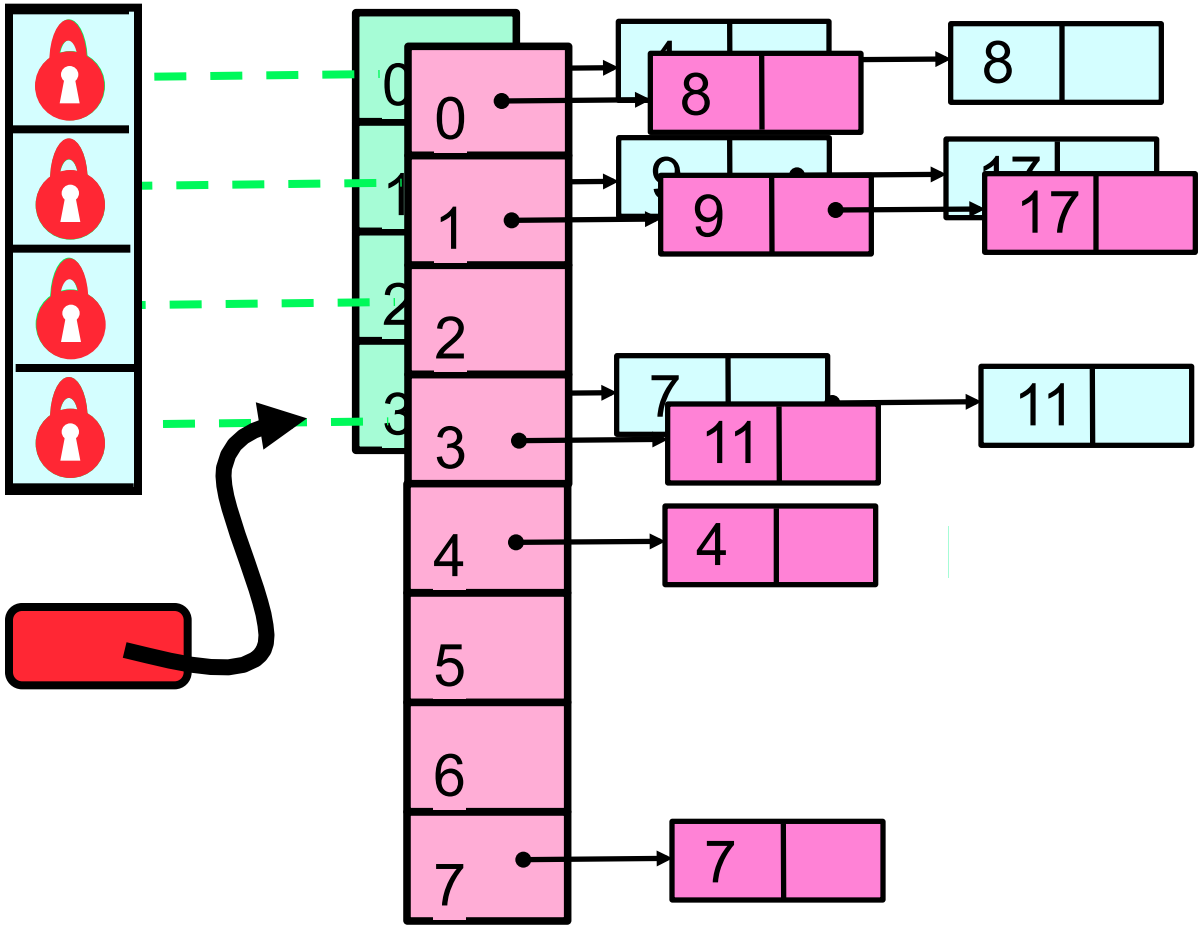


Acquire locks in ascending order

Resize This

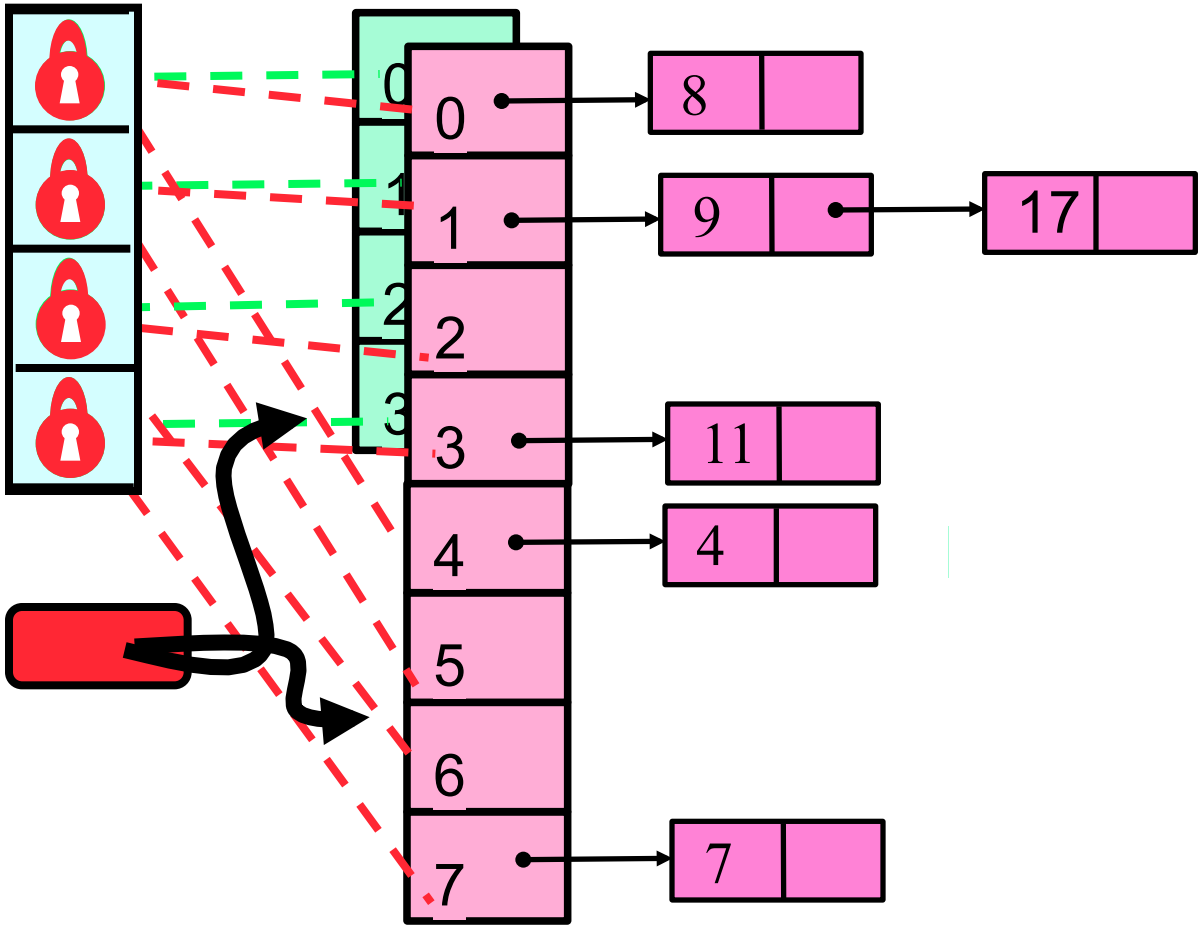


Resize This



Resize This

Striped Locks: each lock now associated with two buckets



Observations

- We grow the table, but not locks
  - Resizing lock array is tricky …
- We use sequential lists
  - Not **LockFreeList** lists
  - If we're locking anyway, why pay?

Example

Assume

- Hash Table of N buckets (stripes)
- Hash function results on average bucket length L
- We quadruple the table (number of buckets)

Scenario #1: Start with N buckets, we keep the total number of entries in the table, hash function is ok, reducing bucket length by 3x

- What is the average number of entries locked per stripe?
- Is the new table faster? By how much?

Example

Scenario #2: Start with N buckets, we also quadruple the number of entries in the table, the hash function is good, keeping bucket length

- ▷ What is the average number of entries locked per stripe?
- ▷ Is the new table faster?

Example

Scenario #2: Start with N buckets, we also quadruple the number of entries in the table, the hash function is good, keeping bucket length

- ▷ What is the average number of entries locked per stripe?
- ▷ Is the new table faster?

Example

Scenario #3: We start with N buckets, we quadruple the number of entries in the table, but hash function is bad, increasing list length by 4x

- ▷ What is the average number of entries locked per stripe?
- ▷ Is there a difference in speed?

Fine-Grained Hash Set

```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 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();
  }} …
```

Fine-Grained Hash Set

```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 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();
  }} …
```

Array of locks

Fine-Grained Hash Set

```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 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();
  }} …
```

Array of buckets

Fine-Grained Hash Set

```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 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 *same* number of locks and buckets

The add() method

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

Fine-Grained Locking

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

Which lock?

The add() method

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

Acquire the lock

Fine-Grained Locking

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

Which bucket?

The add() method

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

Call that bucket's add() method

Fine-Grained Locking

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

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

Insight

- The contains() method
  - Does not modify any fields
  - Why should concurrent **contains()** calls conflict?

A Different Locking Scheme

- **`add`**, **`remove`**, **`contains`**
  - ▷ Lock table in *shared* mode
- **`resize`**
  - ▷ Locks table in *exclusive* mode

Read/Write Locks

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

Read/Write Locks

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

Returns associated read lock

Read/Write Locks

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

Returns associated read lock

Returns associated write lock

Lock Safety Properties

- Read lock:
  - Locks out writers
  - Allows concurrent readers
- Write lock
  - Locks out writers
  - Locks out readers

Read/Write Lock

- Safety
  - If **readers > 0** then **writer == false**
  - If **writer == true** then **readers == 0**
- Liveness?
  - Will a continual stream of readers …
  - lock out writers?

FIFO R/W Lock

- As soon as a writer requests a lock
- No more readers accepted
- Current readers "drain" from lock
- Writer gets in

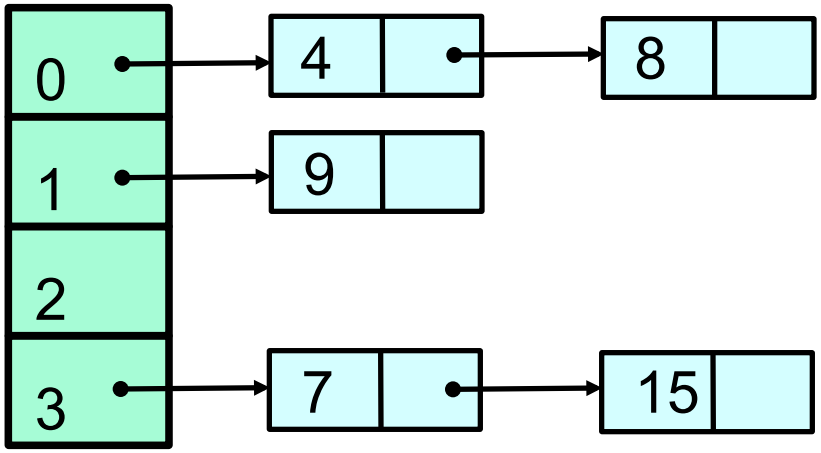
The Story So Far

- Resizing is the hard part
- Fine-grained locks
  - Striped locks cover a range (not resized)
- Read/Write locks
  - FIFO property tricky

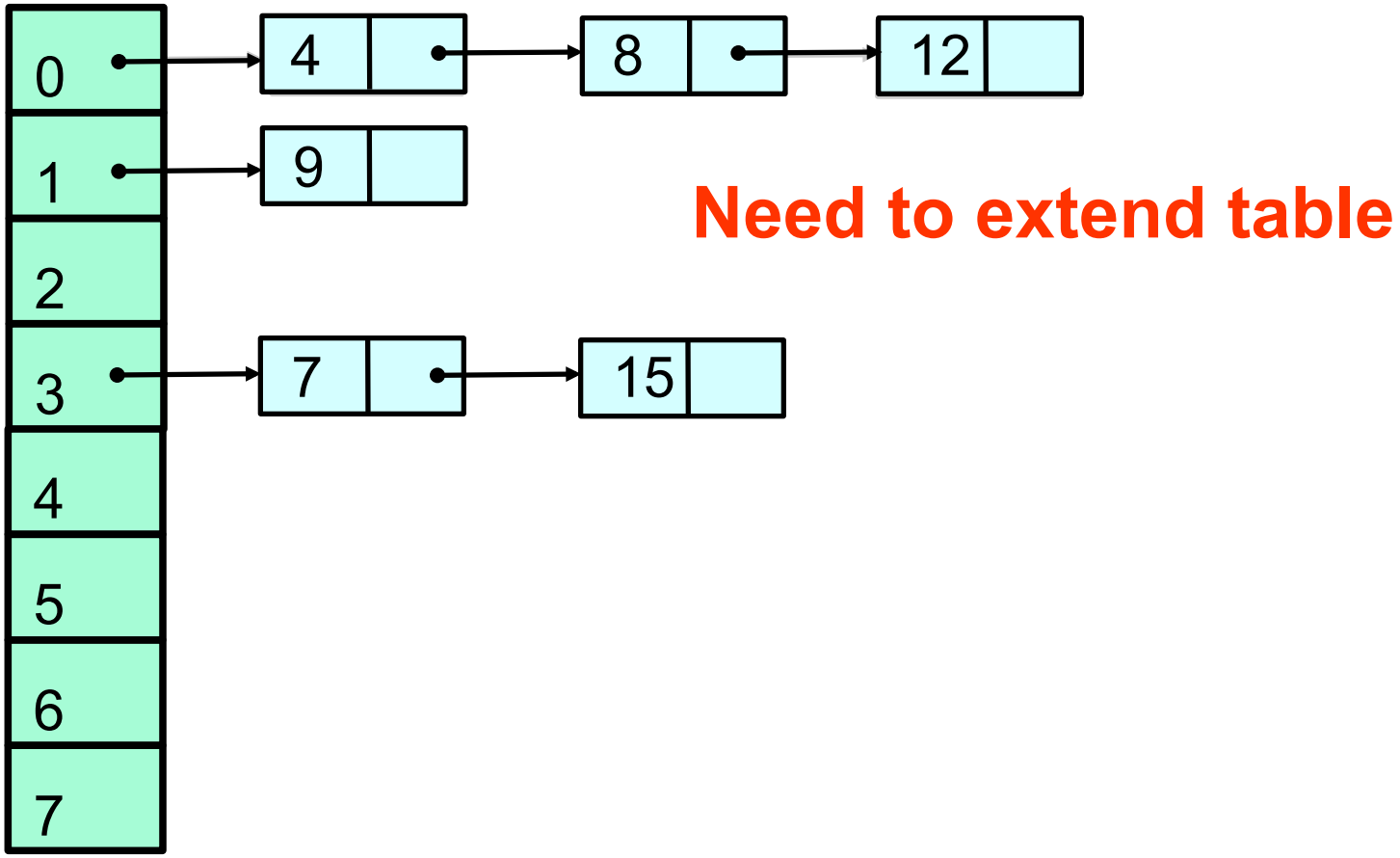
Stop The World Resizing

- Resizing stops all concurrent operations
- What about an incremental resize?
- Must avoid locking the table
- A lock-free table + incremental resizing?

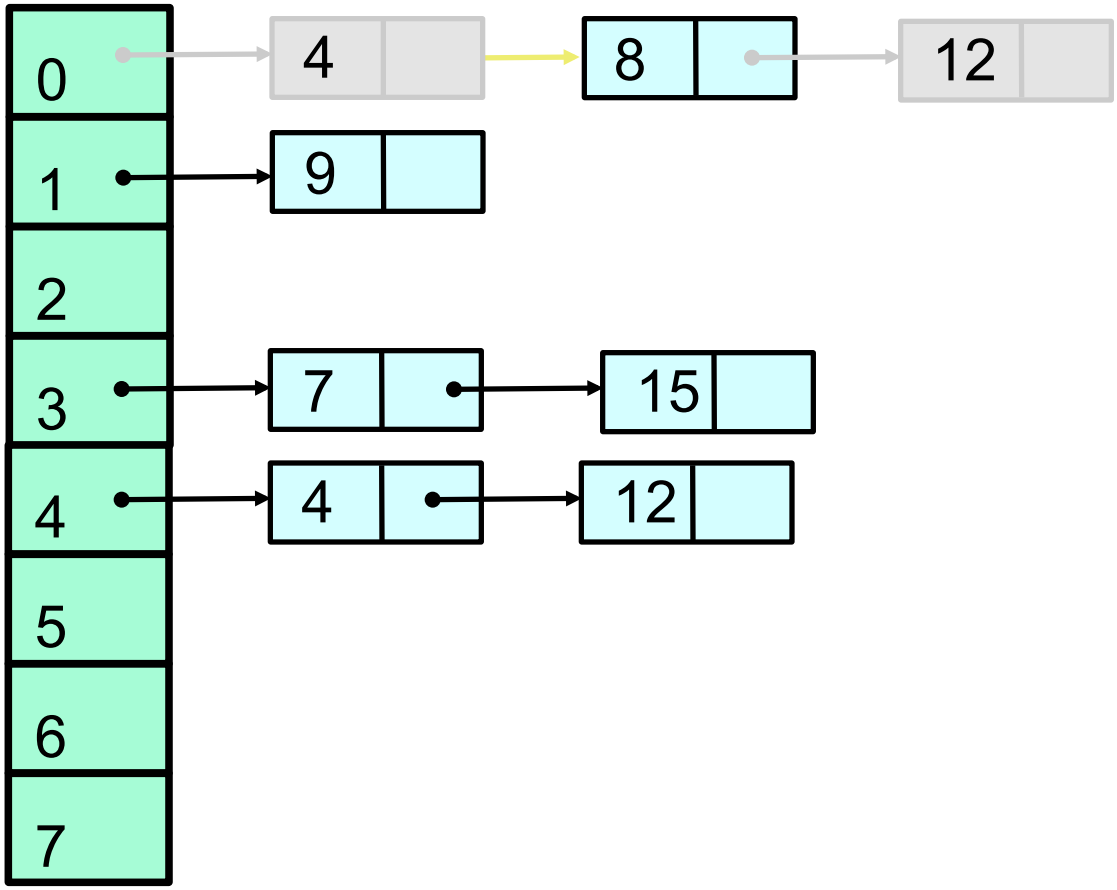
Lock-Free Resizing Problem



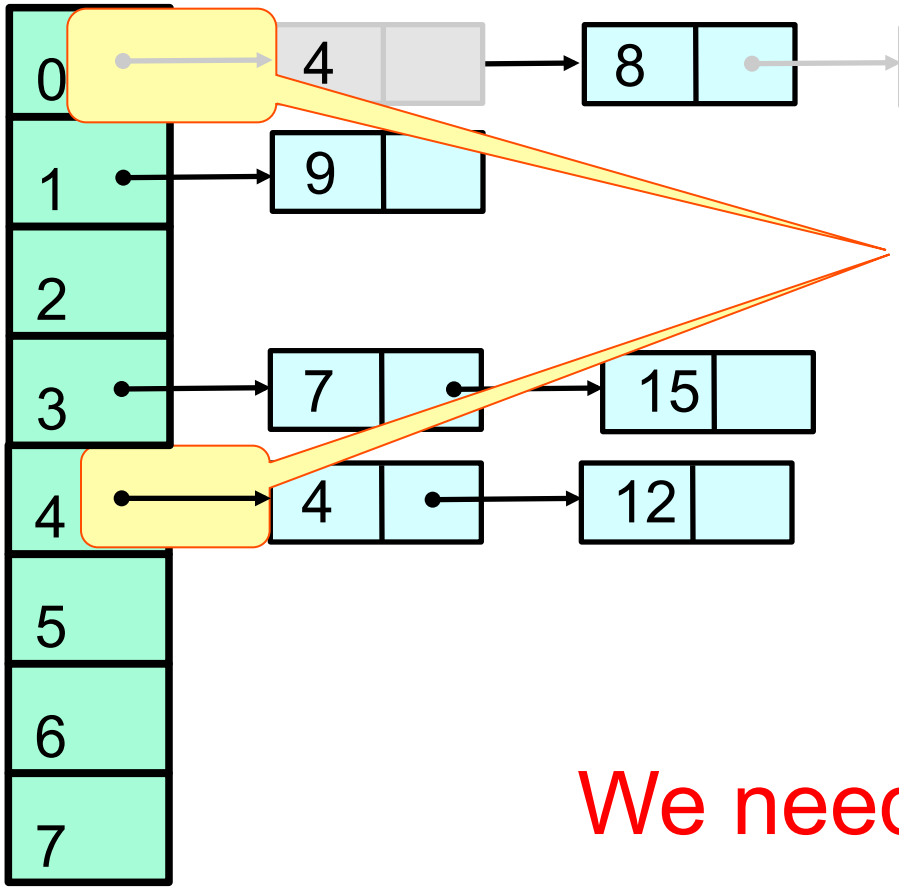
Lock-Free Resizing Problem



Lock-Free Resizing Problem



Lock-Free Resizing Problem



to remove and then add even a single item single location CAS not enough

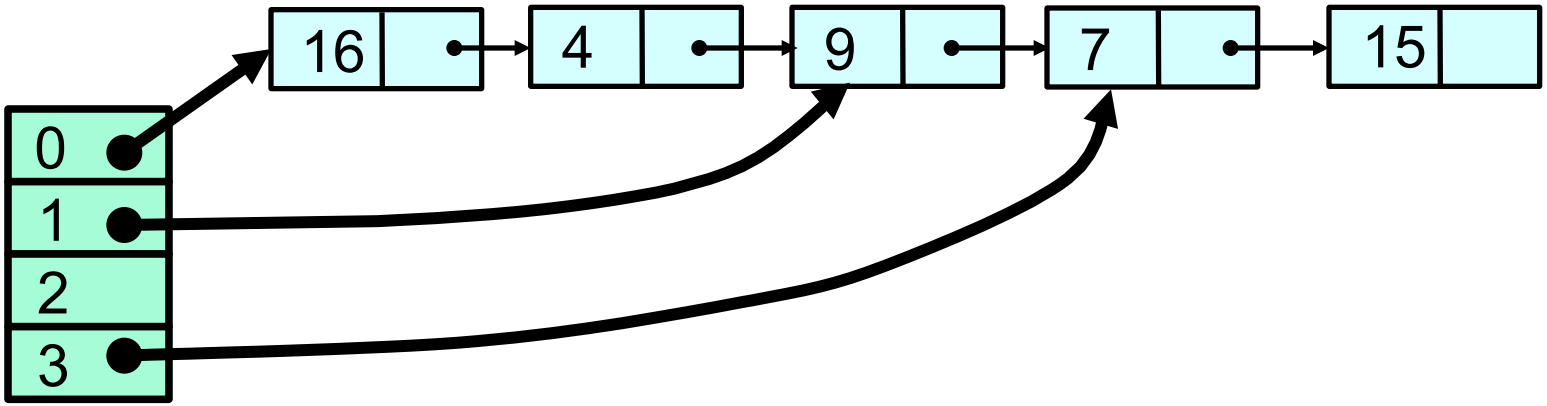
We need a new idea…

Recap

- Coarse-grained Locks
  - Not an option: no concurrency
- Fine-grained Locks
  - Good as long as table not resized
- Striped Locks
  - Keep one lock array
  - Resize the table
  - Bad if resized a lot: can not move items efficiently

Don't move the items

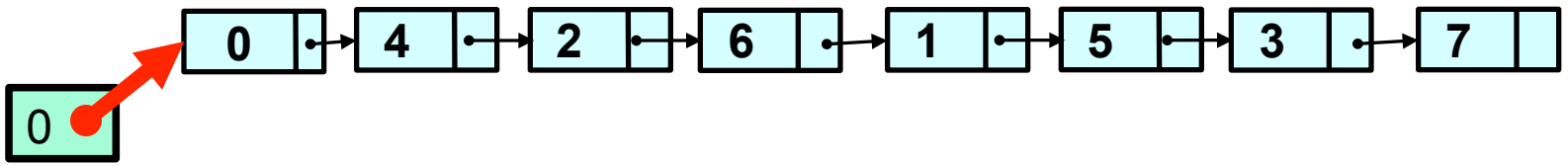
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



What we want

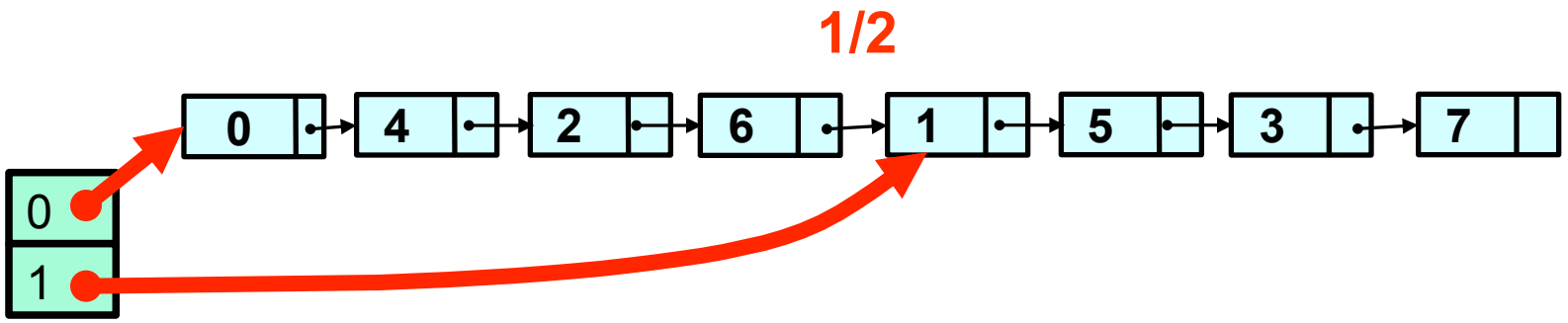
- Quick access to sparse regions
  - Keys clustered in value should be spread out
- Reasonable load balance among shortcuts
  - Shortcuts should cover similar number of keys
- Split based on the LSBs (to randomize)
- Recursive Split-Order:
  - Table with 2^i index ➔ I bits in the binary value of a key
  - Reverse the bit order
  - E.g., 6 in binary is 110 ➔ in reverse 011
  - Split-order ➔ keys sorted in reverse LSB order

Recursive Split Ordering



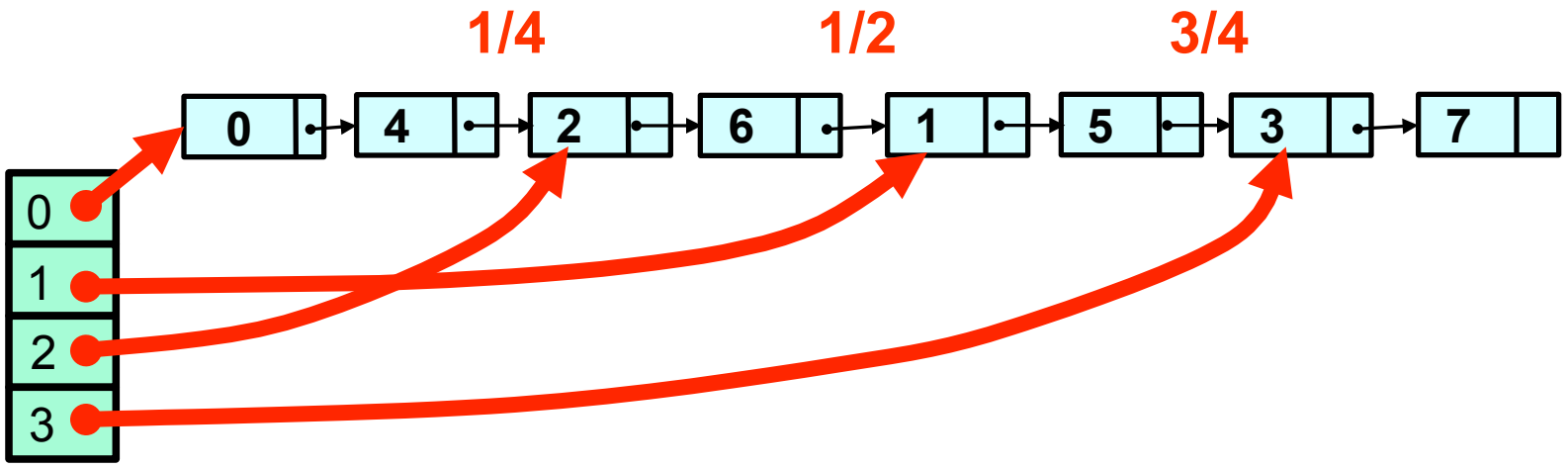
- We start with a list sorted in reverse LSB of keys
  - 000, 001, 010, 011, 100, 101, 110, 111

Recursive Split Ordering



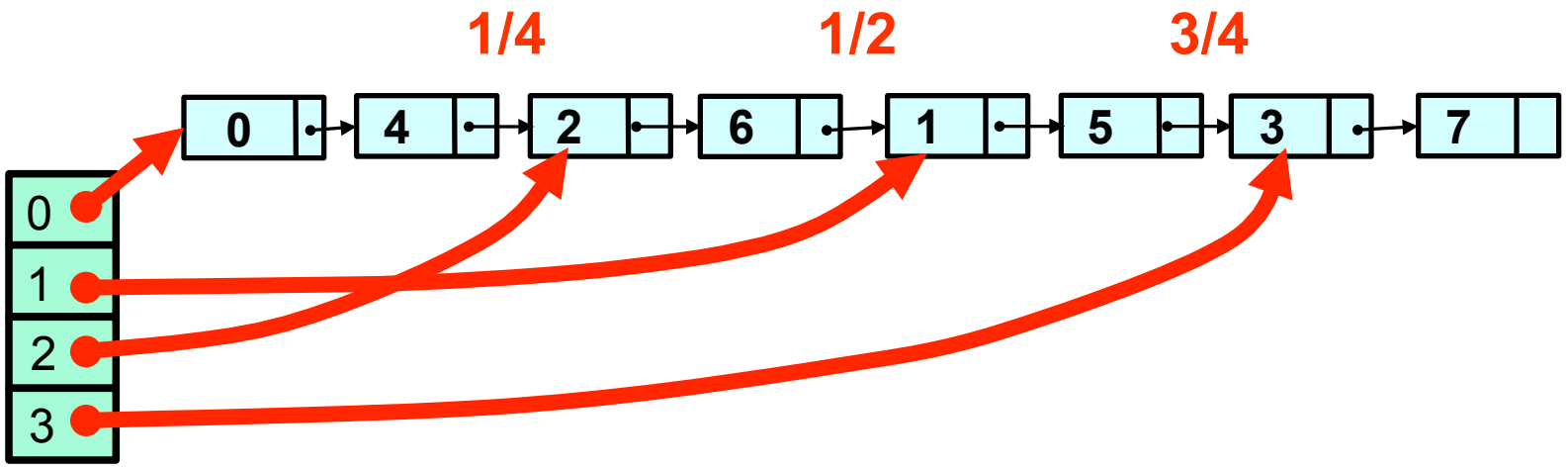
- Bucket index is LSB (single digit)
  - 6 is 110 and belongs to bucket 0
  - 3 is 011 and belongs to bucket 1

Recursive Split Ordering



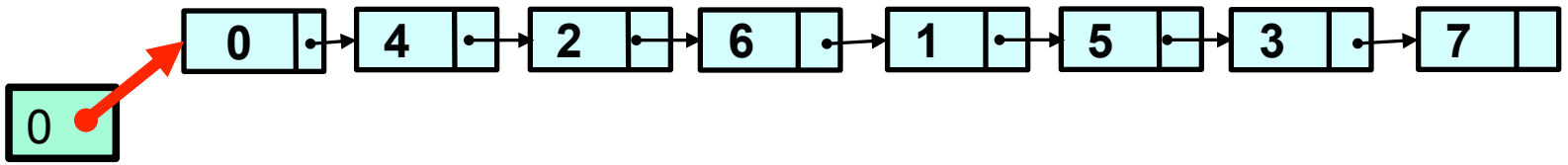
- Bucket index is two LSB digits
  - 6 is 110 and belongs to bucket 2 (10)
  - 3 is 011 and belongs to bucket 3 (11)
  - 1 is 001 and belongs to bucket 1 (01)

Recursive Split Ordering

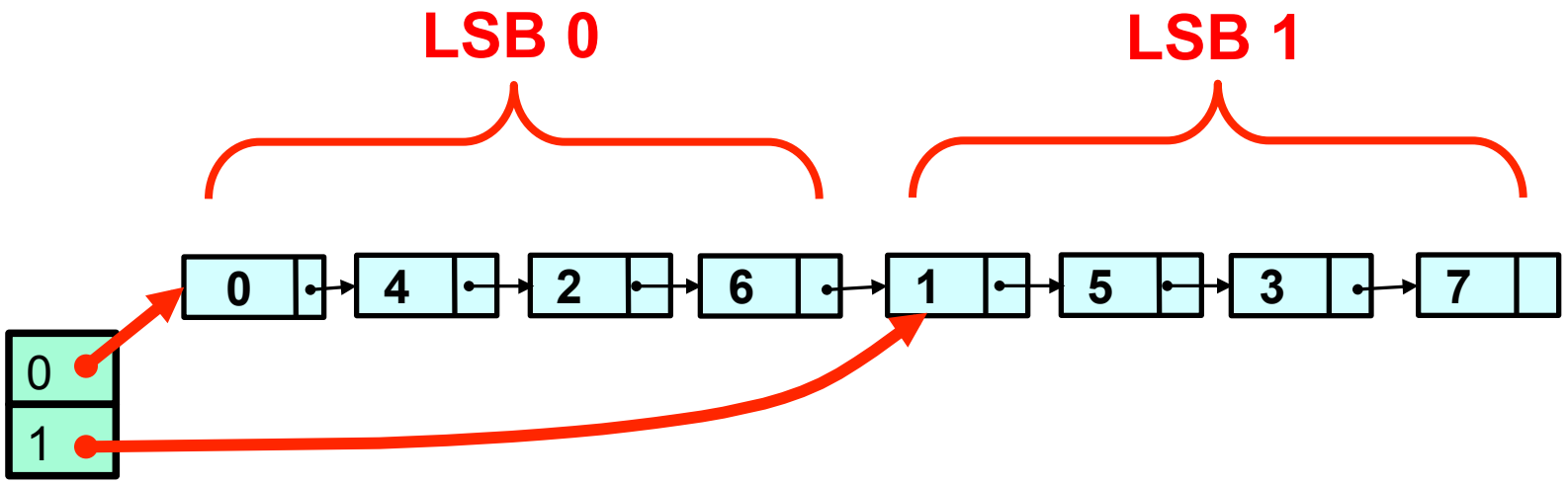


List entries sorted in order that allows recursive splitting

Recursive Split Ordering

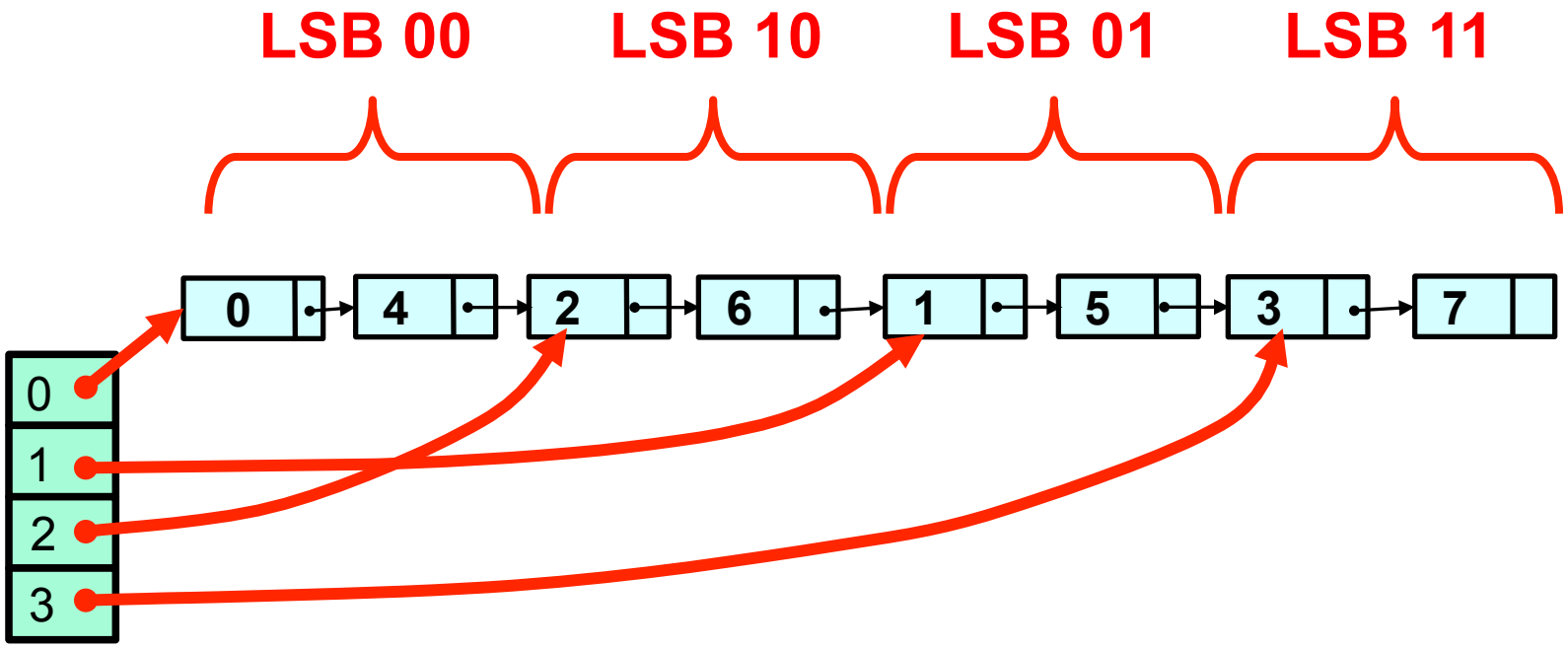


Recursive Split Ordering



LSB = Least significant Bit

Recursive Split Ordering



Split-Order

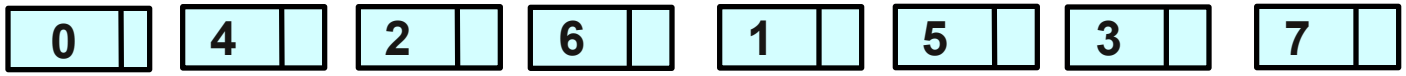
- If the table size is 2^i,
  - Bucket b contains keys k
    - $b = k \pmod{2^i}$
  - bucket index consists of key's i LSBs

When Table Splits

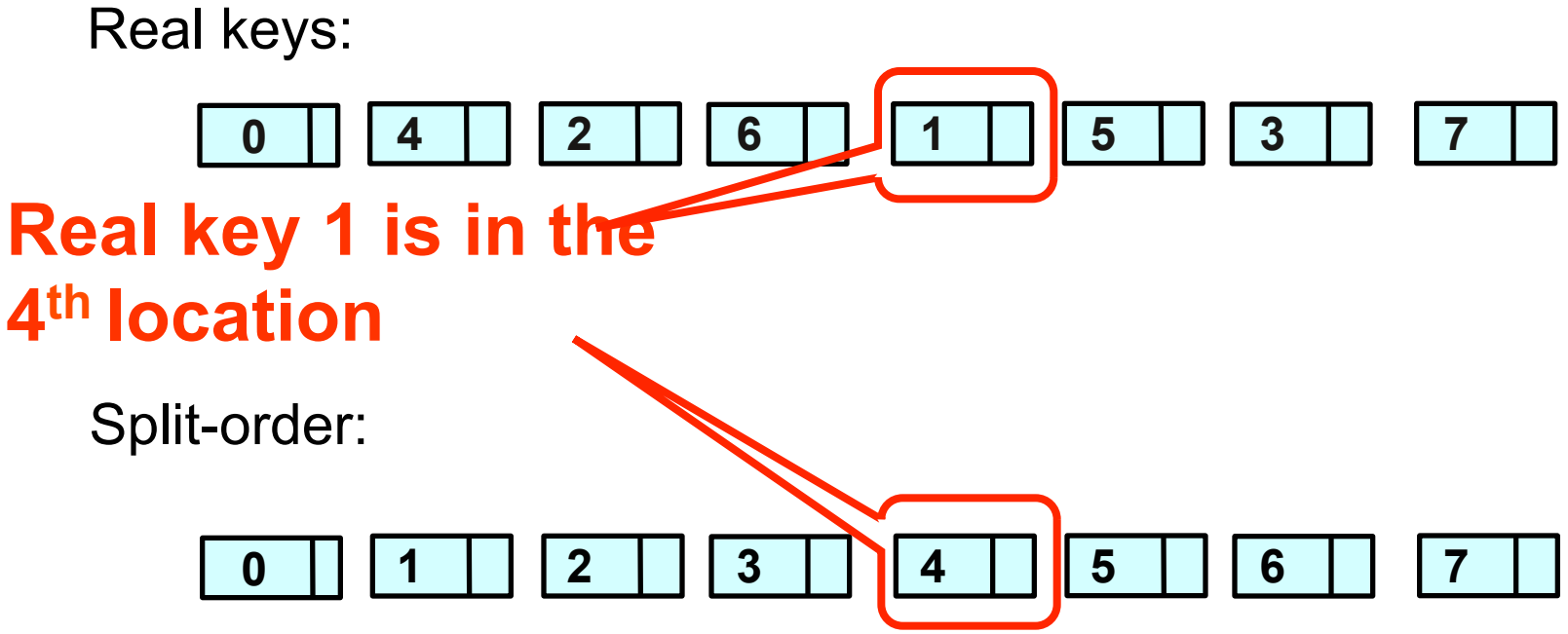
- Some keys stay
  - $b = k \bmod (2^{i+1})$
- Some move
  - $b+2^i = k \bmod (2^{i+1})$
- Determined by $(i+1)^{st}$ bit
  - Counting backwards
- Key must be accessible from both
  - Keys that will move must come later

A Bit of Magic

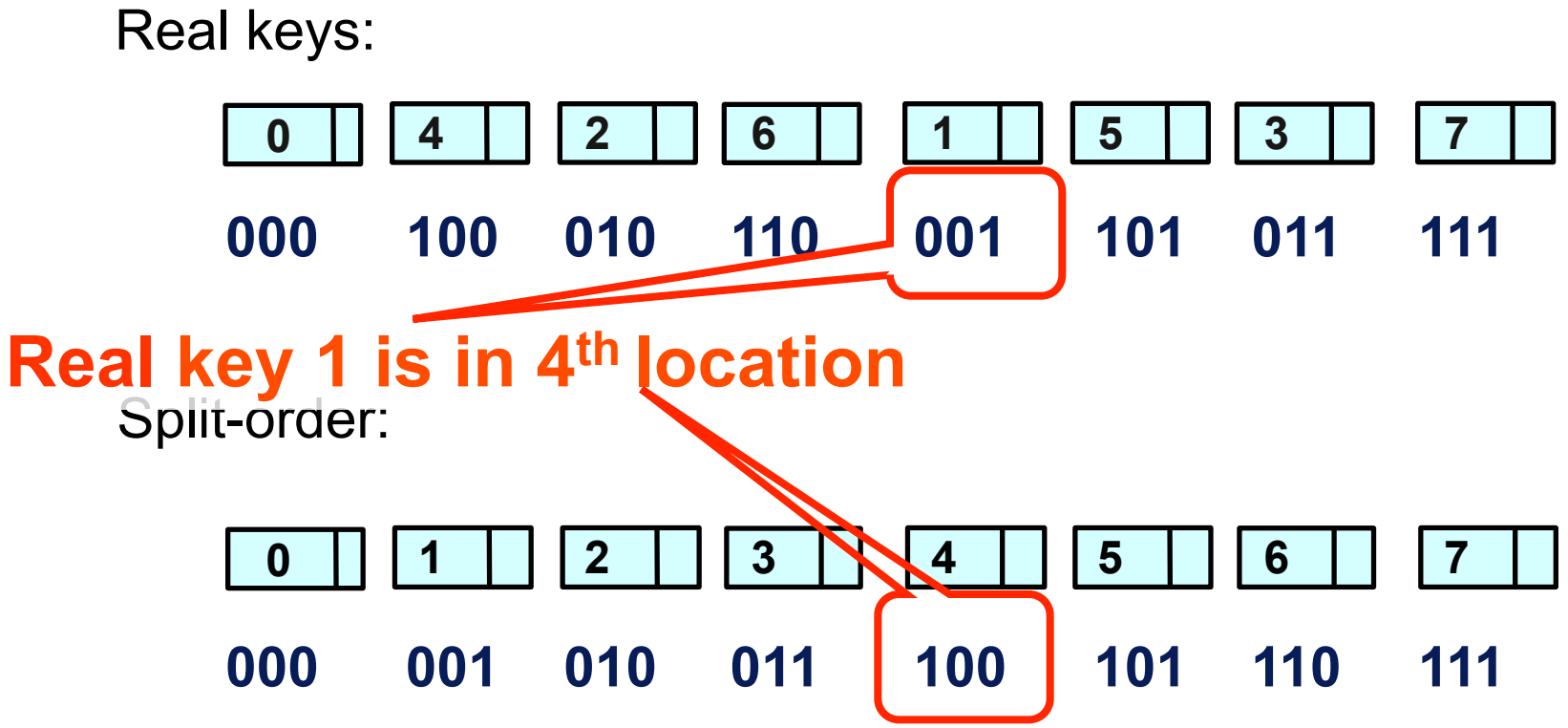
Real keys:



A Bit of Magic



A Bit of Magic



A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

Split-order:

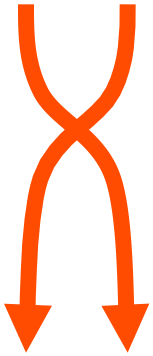
000 001 010 011 100 101 110 111

A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

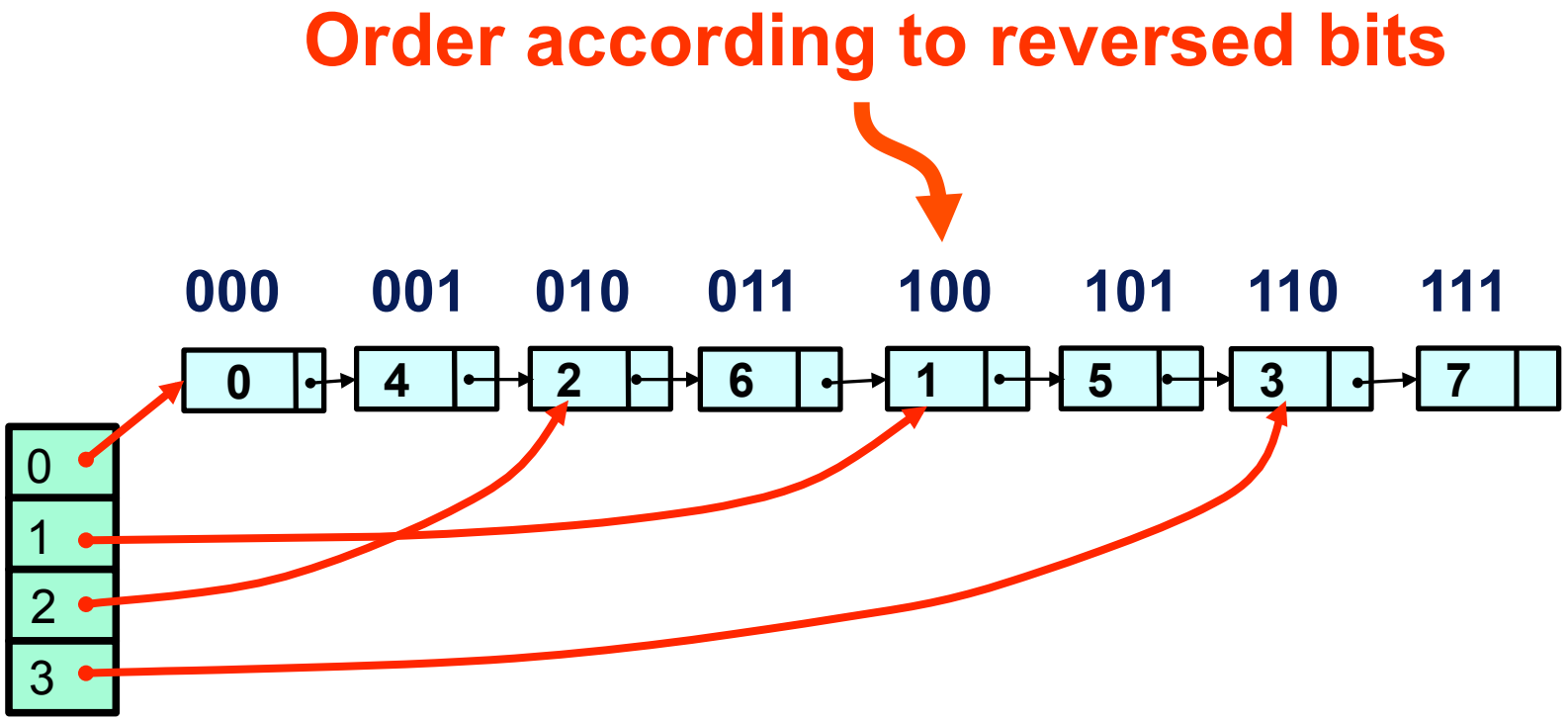
Split-order:



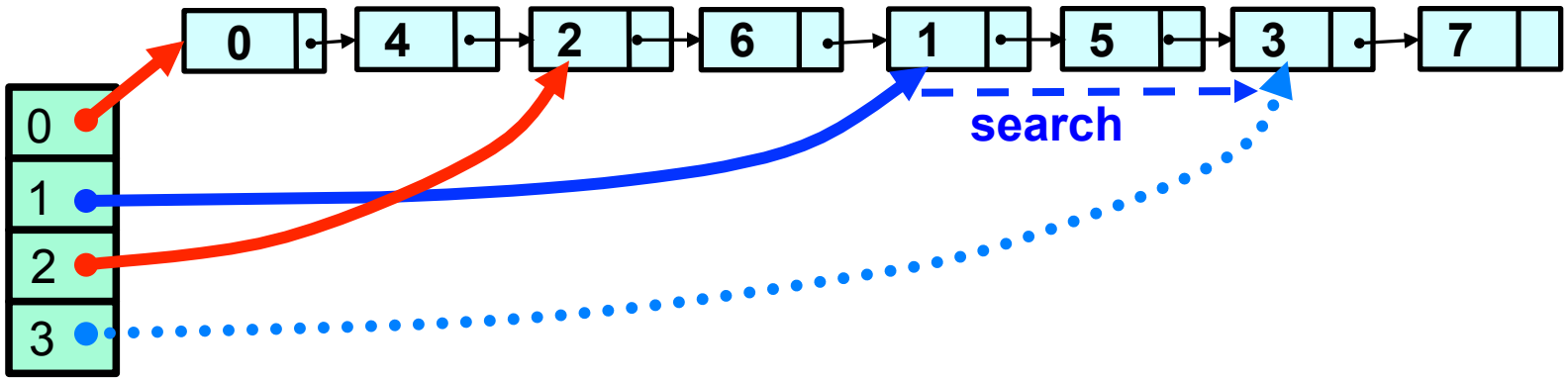
000 001 010 011 100 101 110 111

Just reverse the order of the key bits

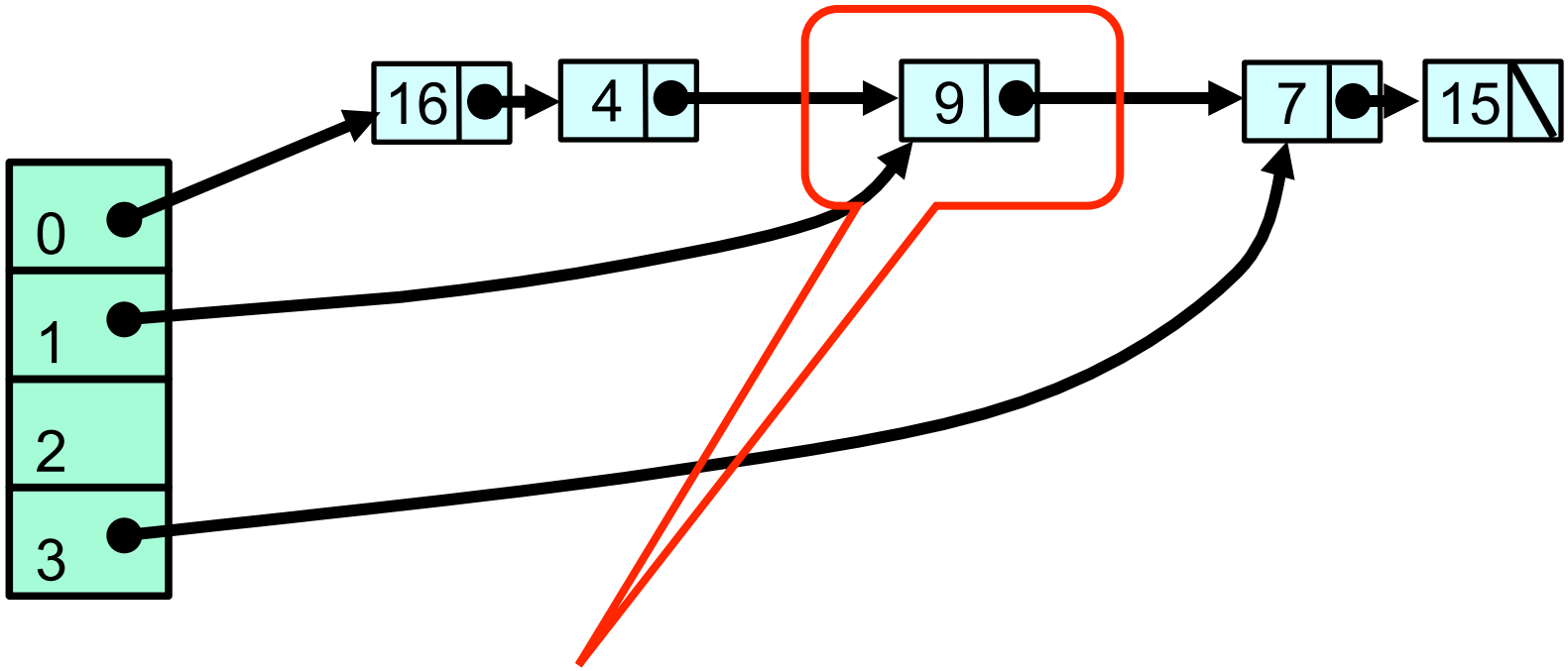
Split Ordered Hashing



Parent Always Provides a Short Cut

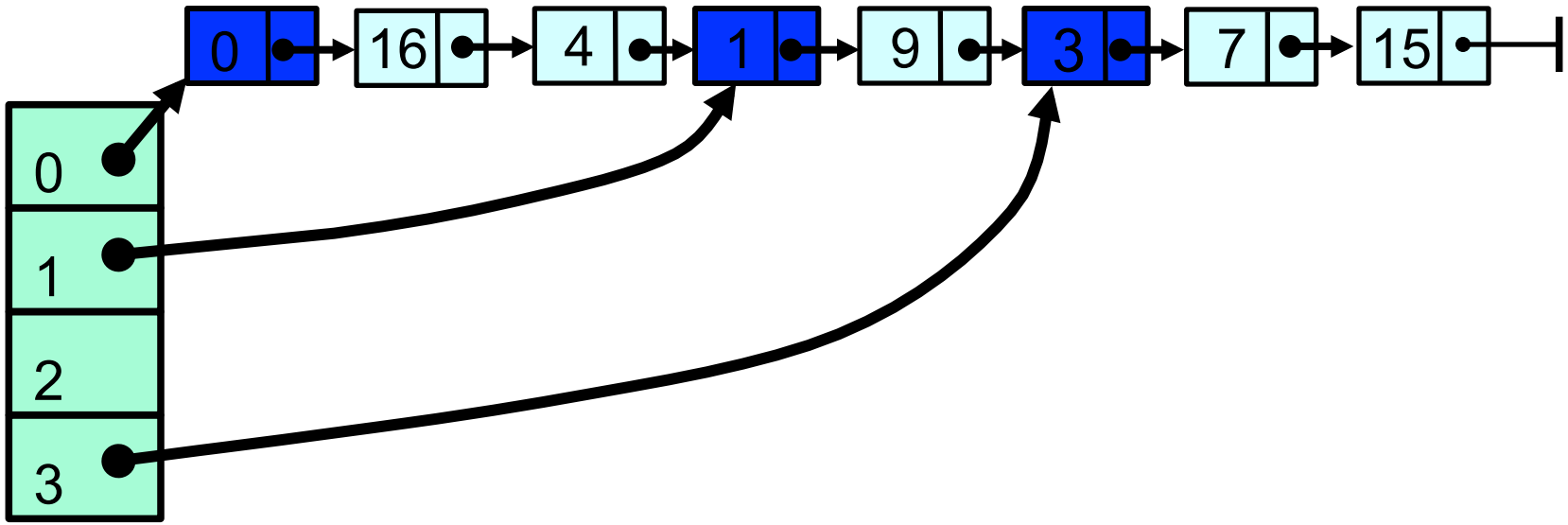


Sentinel Nodes



Problem: how to remove a node pointed by 2 sources using CAS

Sentinel Nodes



Solution: use a Sentinel node for each bucket

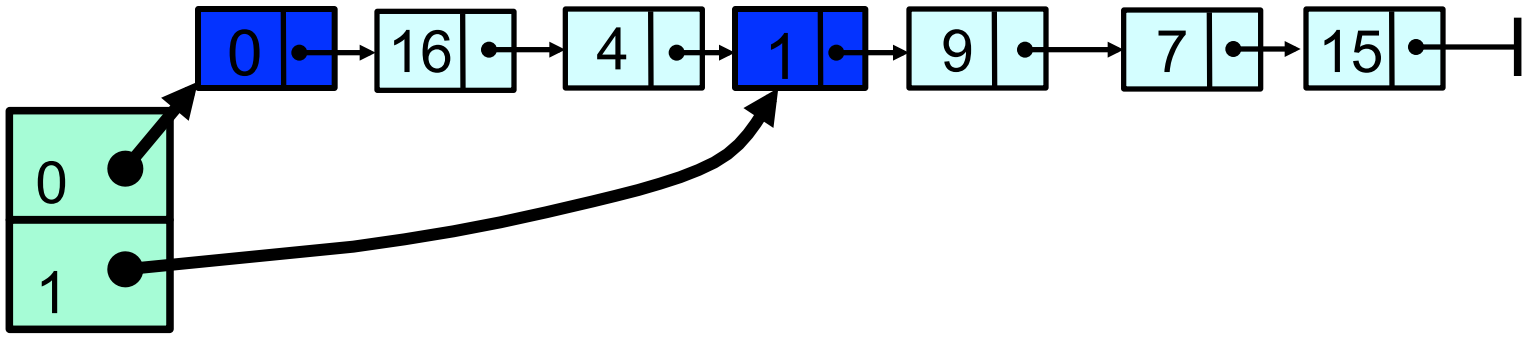
Sentinel vs Regular Keys

- Want sentinel key for i ordered
  - before all keys that hash to bucket i
  - after all keys that hash to bucket (i-1)

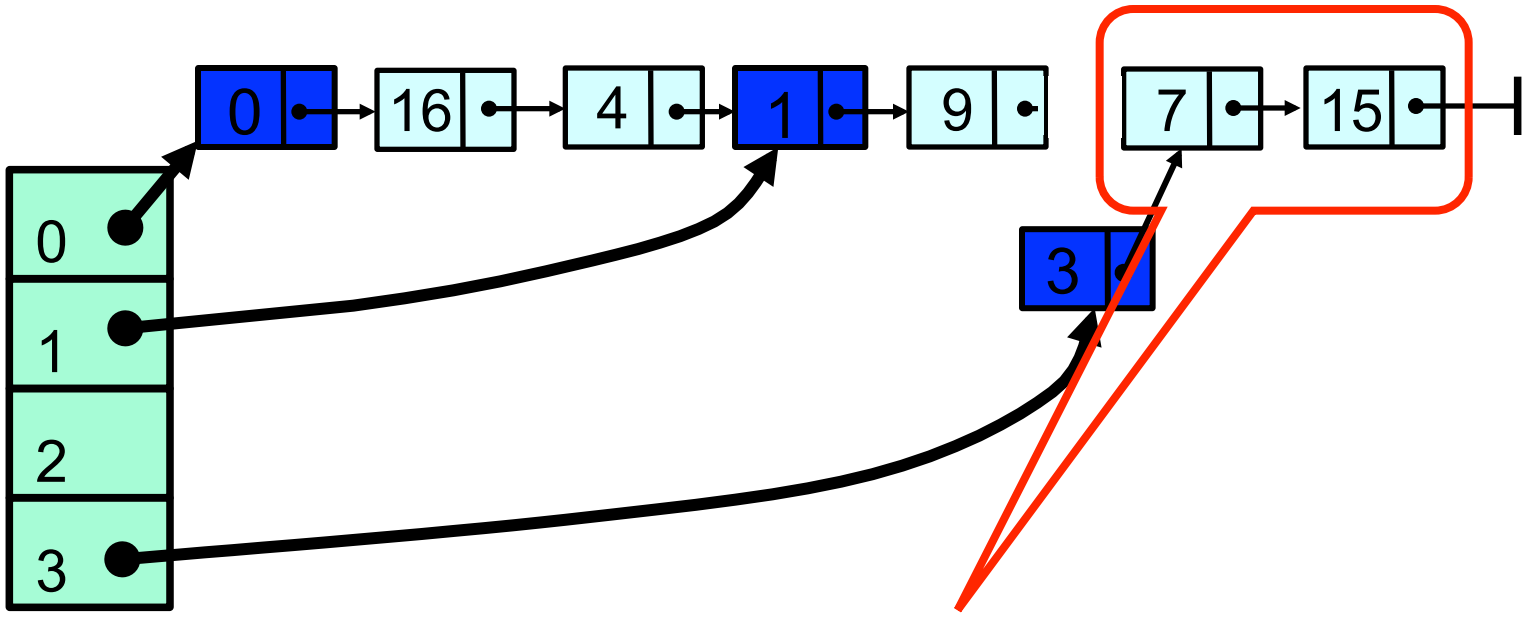
Splitting a Bucket

- We can now split a bucket
- In a lock-free manner
- Using two CAS() calls ...
  - One to add the sentinel to the list
  - The other to point from the bucket to the sentinel

Initialization of Buckets

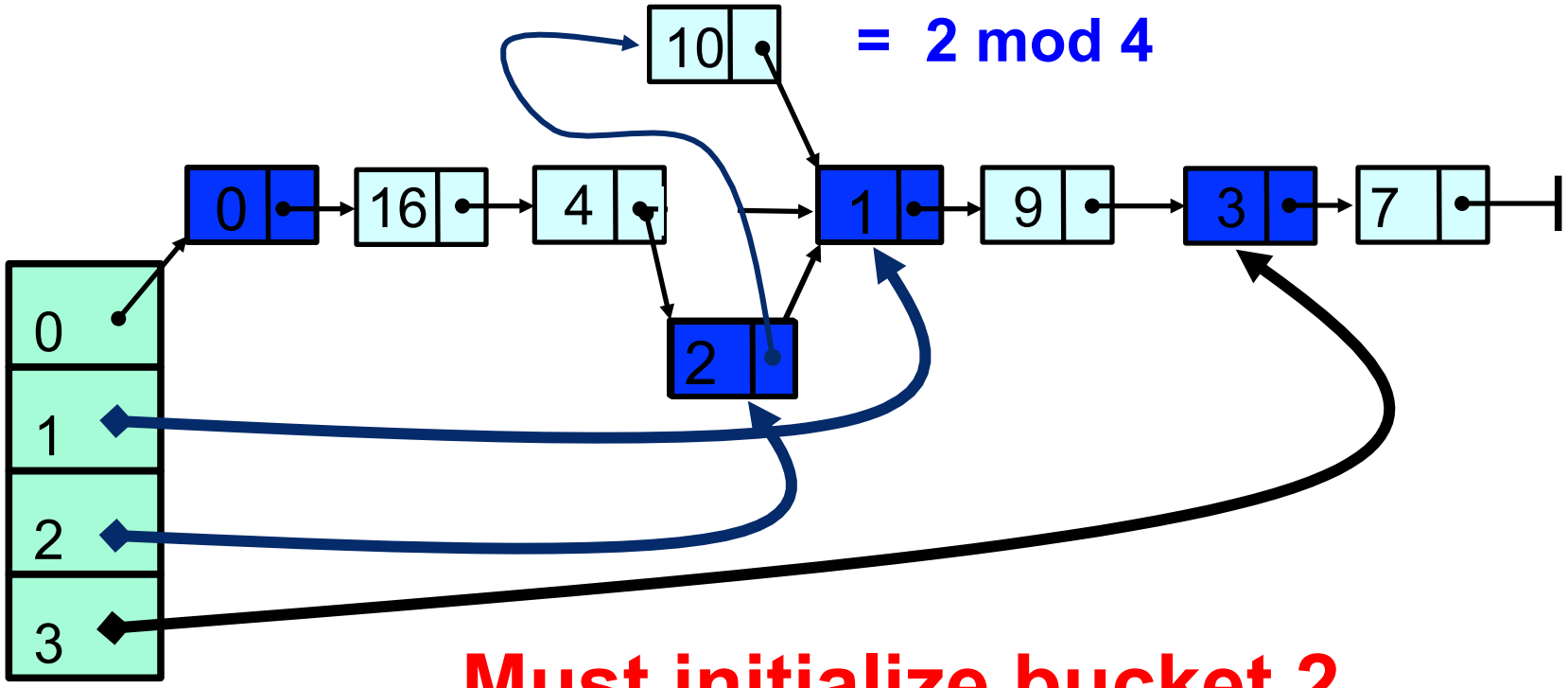


Initialization of Buckets



Need to initialize bucket 3 to split bucket 1

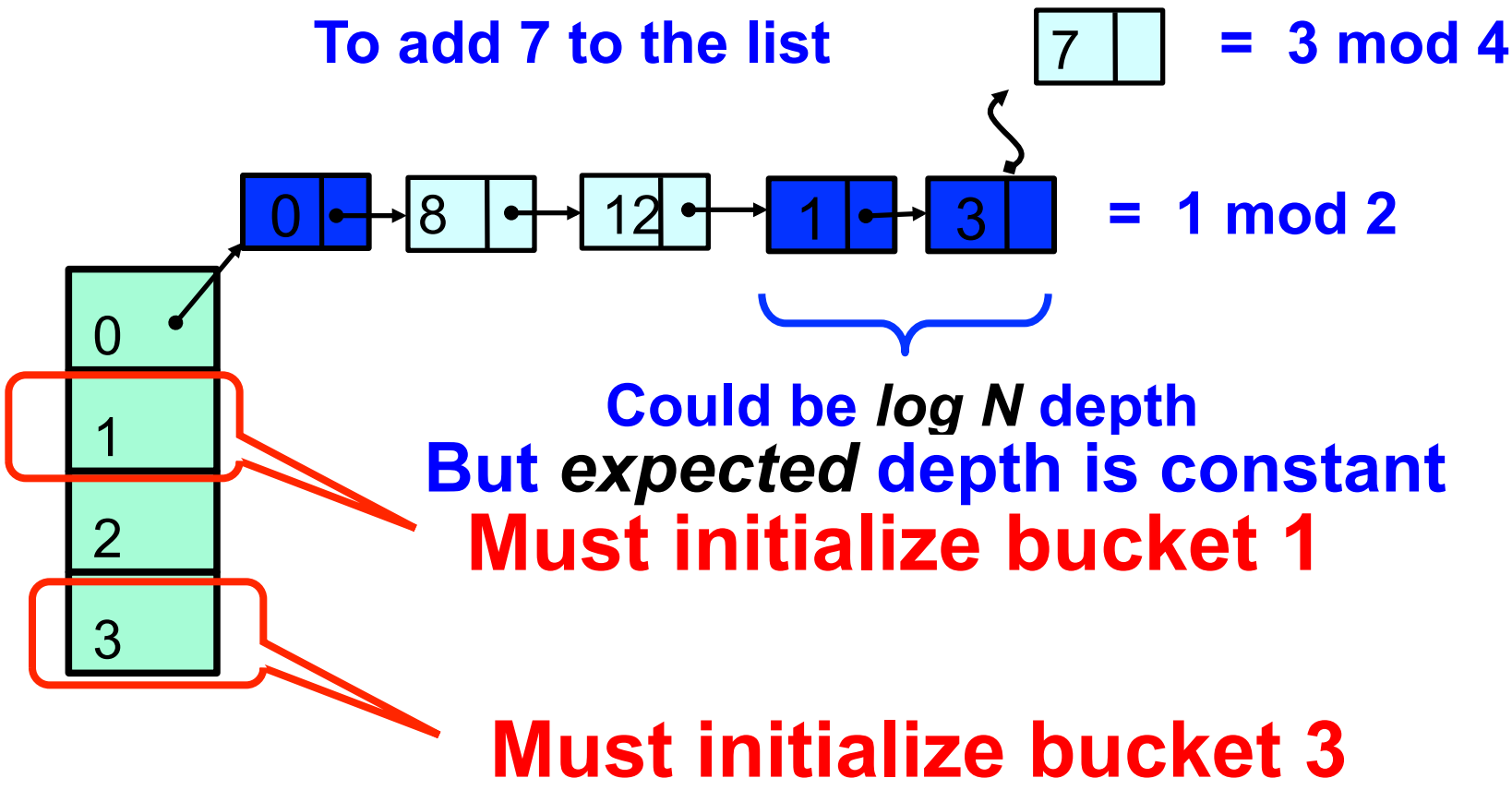
Adding 10



Must initialize bucket 2

Before adding 10

Recursive Initialization



To distinguish Sentinels from Keys

- In the pictures, use a different color
  - Dark blue sentinel, light blue key
- In actual list
  - Set the MSB bit to one for all regular keys
  - Then reverse
- So, sentinels have the LSB equal to 0

Lock-Free List

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

Lock-Free List

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

Regular key: set high-order bit to 1 and reverse

Lock-Free List

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

Sentinel key: simply reverse (high-order bit is 0)

Main List

- Lock-Free List from earlier class
- With some minor variations

Lock-Free List

```
public class LockFreeList {
 public boolean add(Object object,
                    int key) {...}
 public boolean remove(int k) {...}
 public boolean contains(int k) {...}
 public
  LockFreeList(LockFreeList parent,
               int key) {...};
}
```

Lock-Free List

```
public class LockFreeList {
 public boolean add(Object object,
                    int key) {...}
 public boolean remove(int k) {...}
 public boolean contains(int k) {...}
 public
  LockFreeList(LockFreeUtil.Parameter,
               int key) {...};
}
```

Change: add takes key argument

Lock-Free List

Inserts sentinel with key if not already present …

```
public class LockFreeList {
  public boolean add(Object object,
                     int key) {...}
  public boolean remove(int k) {...}
  public boolean contains(int k) {...}
  public
   LockFreeList(LockFreeList parent,
                int key) {...};
}
```

Lock-Free List

… returns new list starting with sentinel (shares with parent)

```
public class LockFreeList {
                                    t,
                        int key) {...}
 public boolean remove(int k) {...}
 public boolean contains(int k) {...}
 public
  LockFreeList(LockFreeList parent,
               int key) {...};
}
```

Split-Ordered Set: Fields

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

  public SOSet(int capacity) {
    table = new LockFreeList[capacity];
    table[0] = new LockFreeList();
    tableSize = new AtomicInteger(2);
    setSize = new AtomicInteger(0);
  }
```

Fields

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

  public SOSet(int capacity) {
    table = new LockFreeList[capacity];
    table[0] = new LockFreeList();
    ta
    se
  }
```

For simplicity treat table as big array …

Fields

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

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

In practice, want something that grows dynamically

Fields

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

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

How much of table array are we actually using?

Fields

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

  public SOSet(int capacity) {
    table = new LockFreeList[capacity];
    table[0] = new LockFreeList();
    tableSize = new AtomicInteger(2);
    setSize = new AtomicInteger(0);
  }
```

Track set size so we know when to resize

Fields

Initially use single bucket, and size is zero

```
public class SOSet {
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

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

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;
}
```

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

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;
}
```

Non-Sentinel split-ordered key

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;
}
```

Get reference to bucket's sentinel, initializing if necessary

add()

Call bucket's add() method with reversed key

```
int key = makeRegularKey(hash);
LockFreeList list
  = getBucketList(bucket);
if (!list.add(object, key))
    return false;
 resizeCheck();
 return true;
}
```

add()

No change? We're done.

```
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;
}
```

add()

Time to resize?

```
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;
}
```

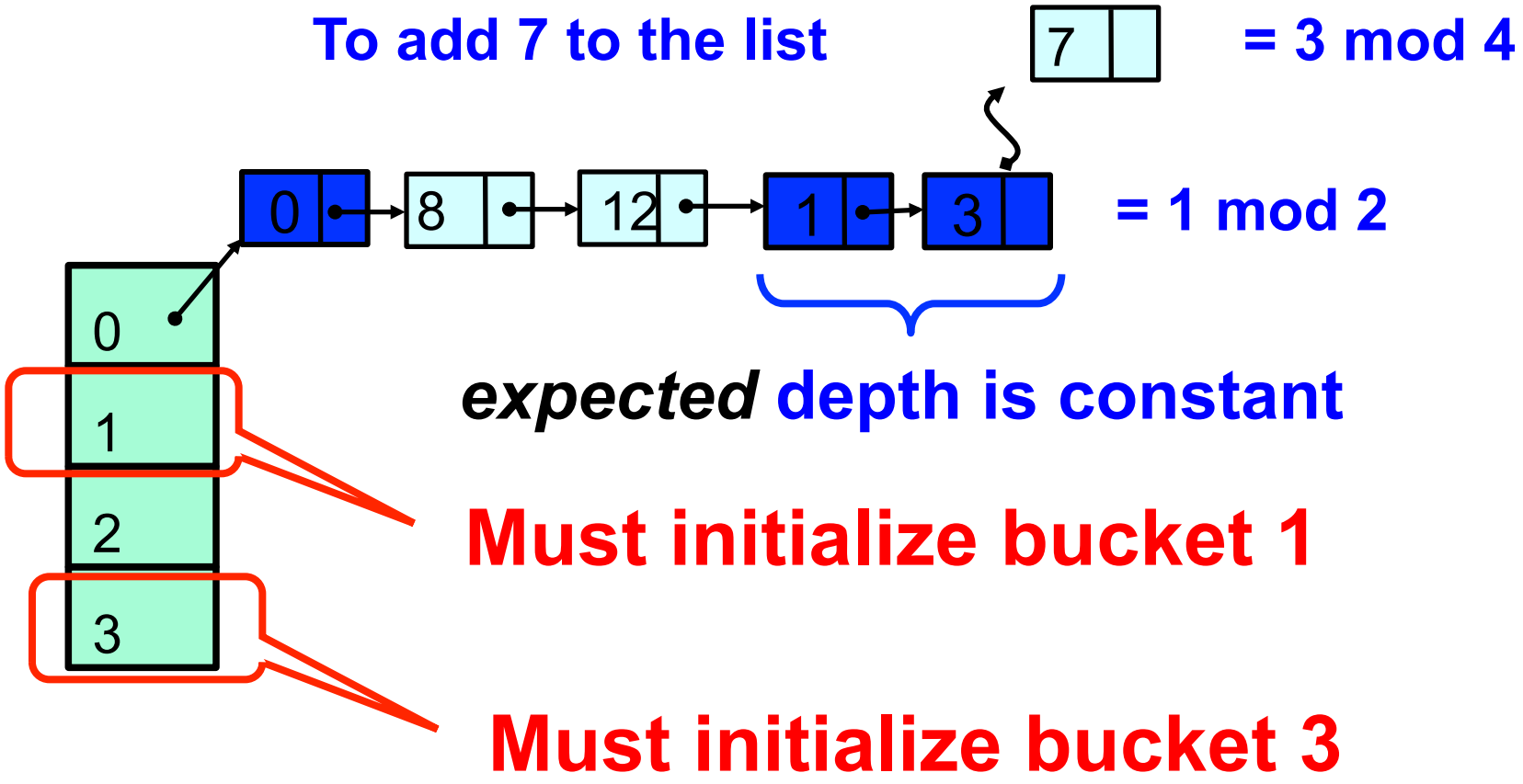
add()

- Divide set size by total number of buckets
- If quotient exceeds threshold
  - Double **tableSize** field
  - Up to fixed limit

Initialize Buckets

- Buckets originally null
- If you find one, initialize it
- Go to bucket's parent
  - Earlier nearby bucket
  - Recursively initialize if necessary
- Constant expected work

Recall: Recursive Initialization



Initialize Bucket

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

Initialize Bucket

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

Find parent, recursively

initialize if needed

Initialize Bucket

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

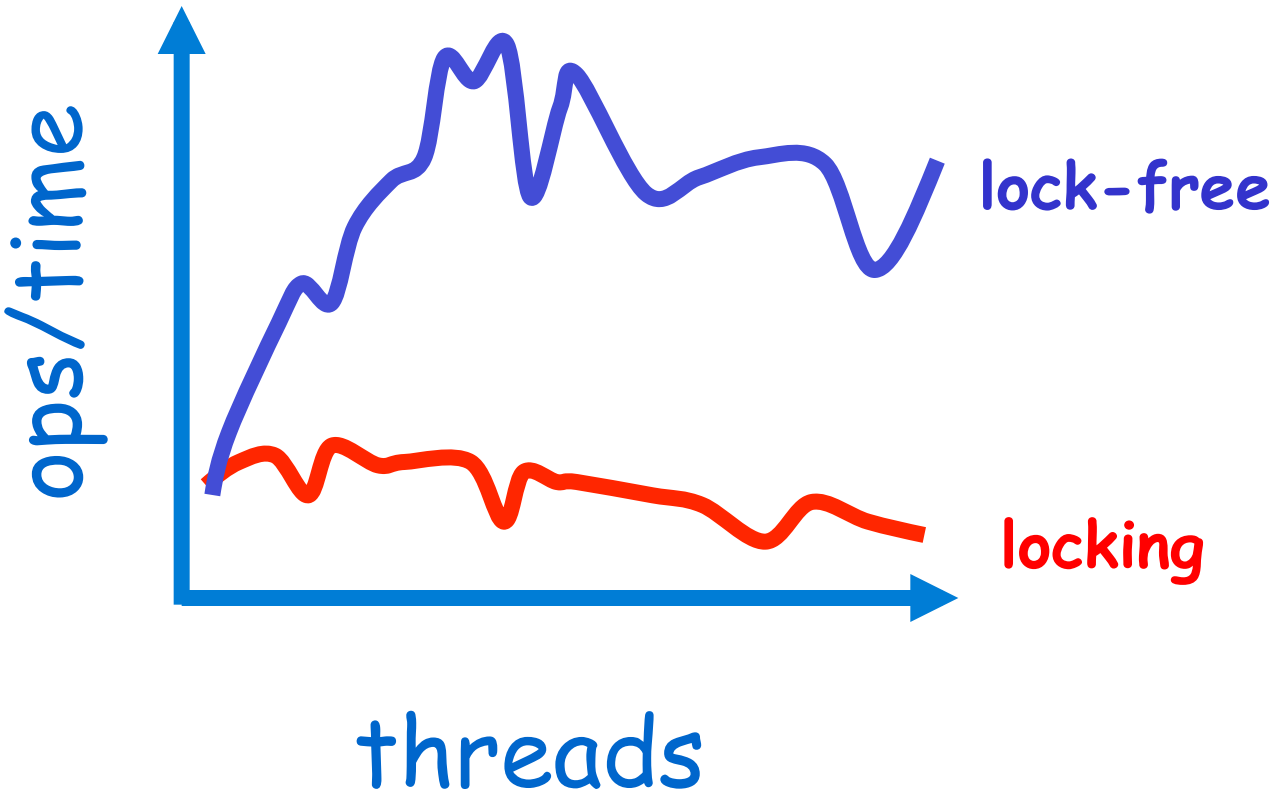
Prepare key for new sentinel

Initialize Bucket

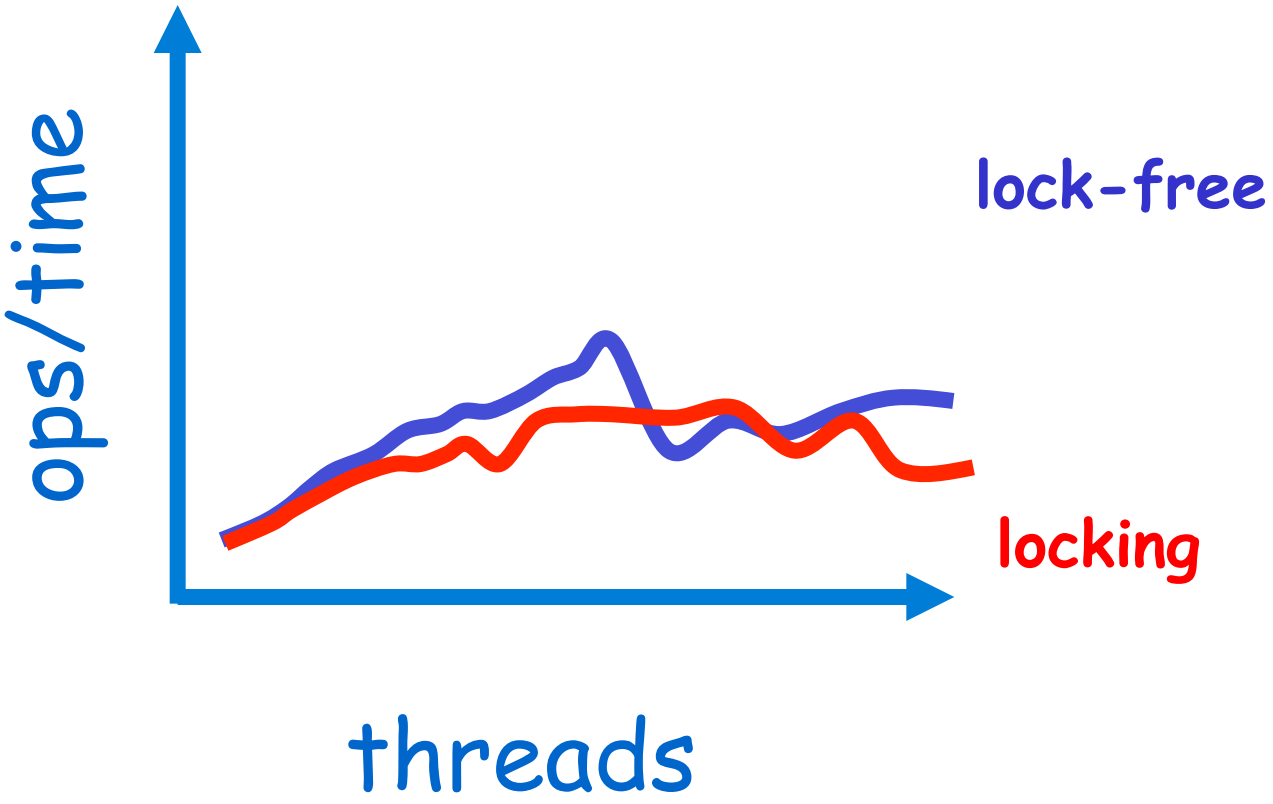
Insert sentinel if not present, and get back reference to rest of list

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

Performance? Assuming little Work, Work = 0



Performance? With more Work, Work = 500



Summary

- Concurrent resizing is tricky
- Lock-based
  - Fine-grained
  - Read/write locks
  - Optimistic
- Lock-free
  - Builds on lock-free list

Additional Performance

- The effects of the choice of locking granularity
- The effects of bucket size