Concurrent Hashing

BROWN

Maurice Herlihy

CS176

Fall 2005

Linked Lists

- We looked at a number of ways to make highly-concurrent lists
  - Fine-grained locks
  - Optimistic synchronization
  - Lock-free synchronization
- What's missing?

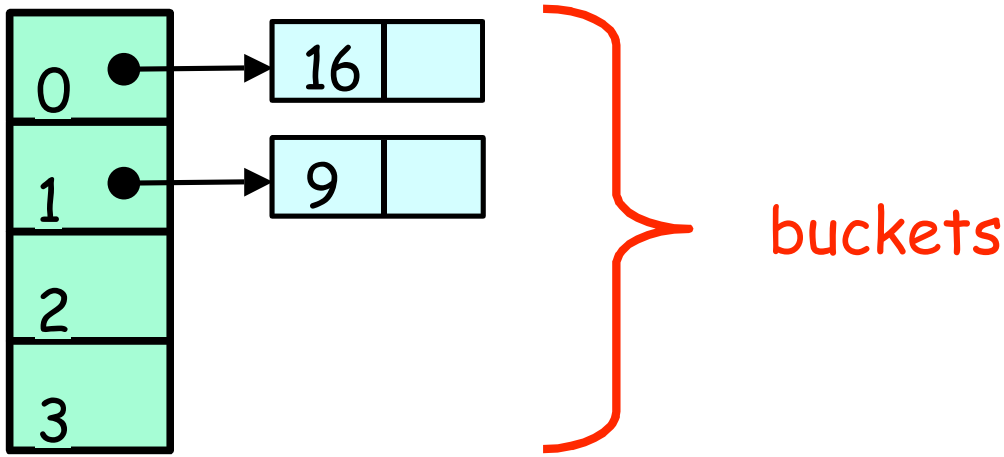
Linear-Time Methods

- Problem is
  - add(), remove(), contains() methods
  - All take time linear in set size
- What we want
  - Constant-time methods
  - (on average)

Hashing

- Hash function
  - h: objects _ integers
- Uniformly distributed
  - Different objects most likely have different hash values
- Java hashCode() method

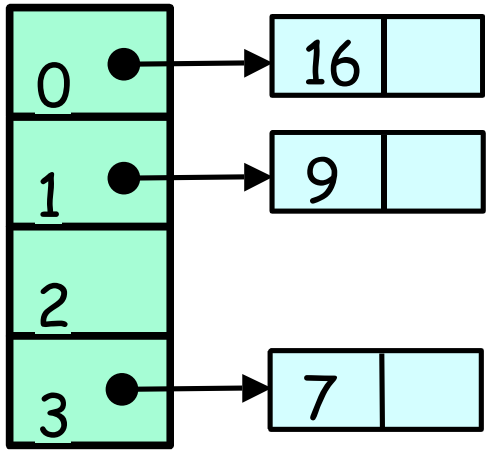
Sequential Hash Table



Item count: 2

$h(k) = k \bmod 4$

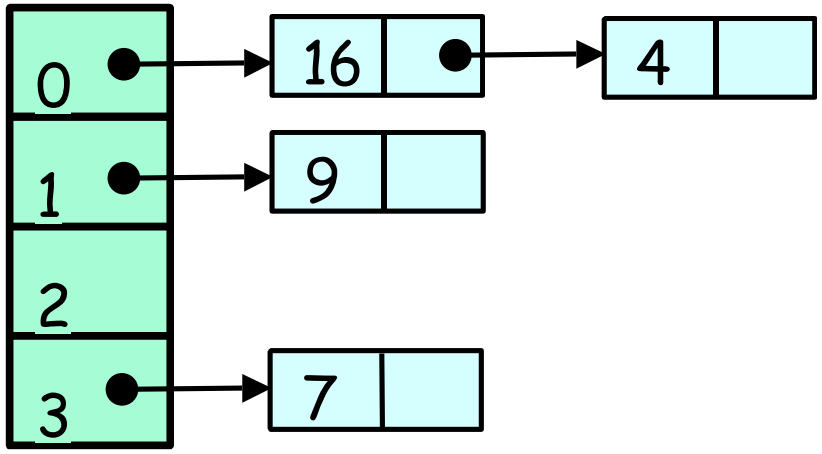
Sequential Hash Table



Item count: 3

$h(k) = k \bmod 4$

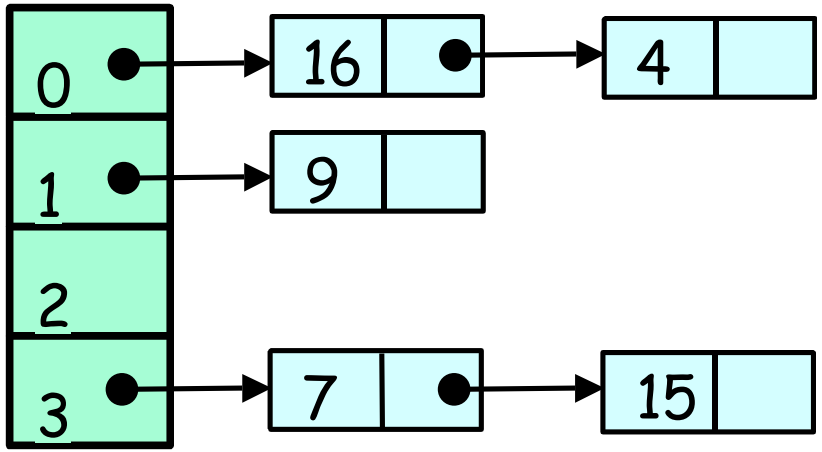
Sequential Hash Table



Item count: 4

$h(k) = k \bmod 4$

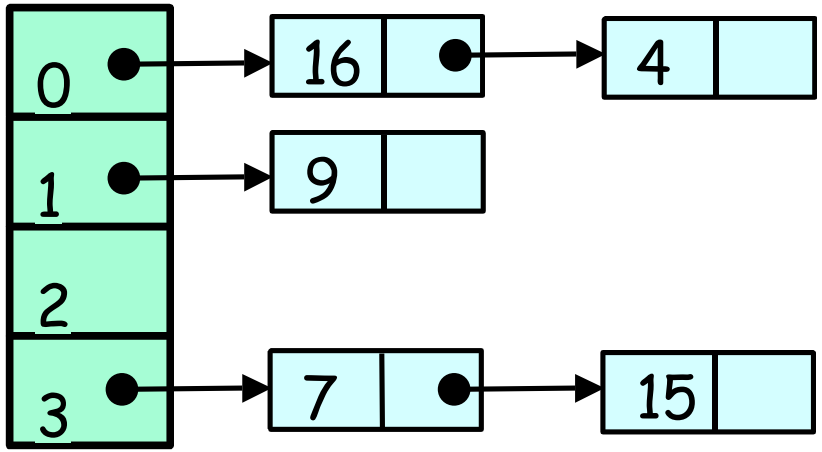
Sequential Hash Table



Item count: 5

$h(k) = k \bmod 4$

Sequential Hash Table

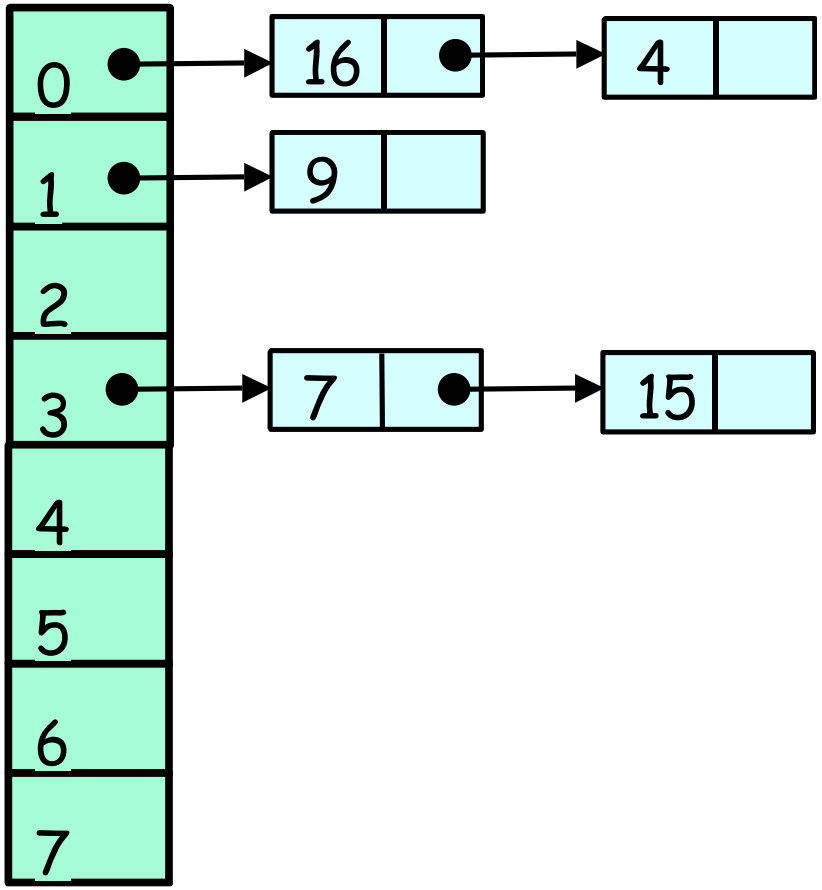


Problem:
buckets getting too long

Item count: 5

$h(k) = k \bmod 4$

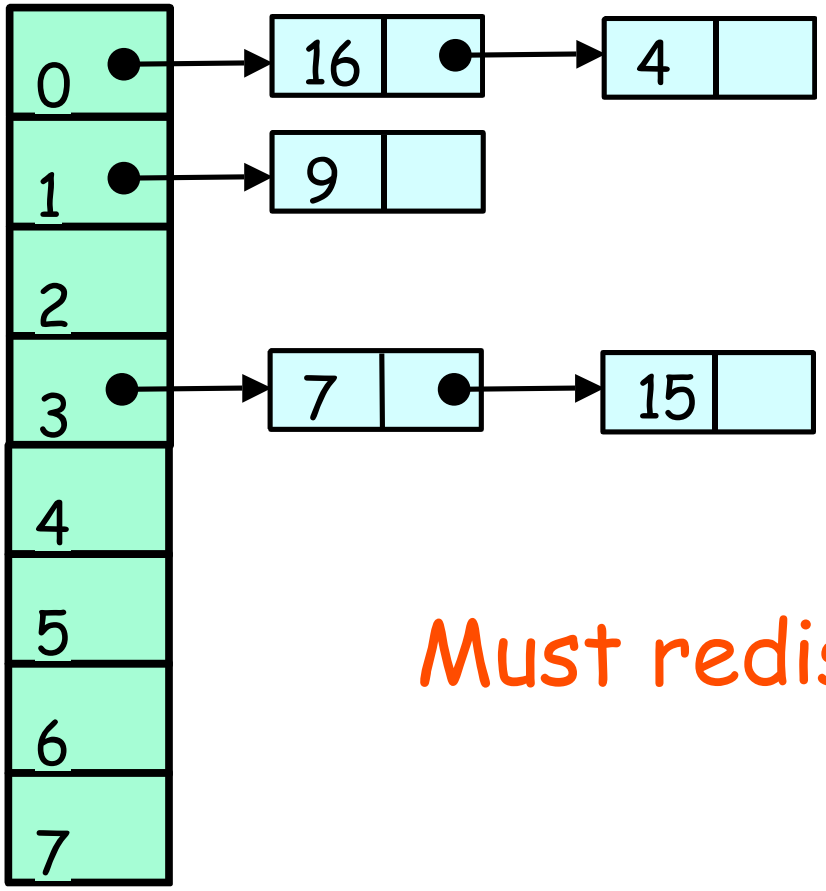
Resizing the table



Item count: 5

$h(k) = k \bmod 4$

Resizing the table

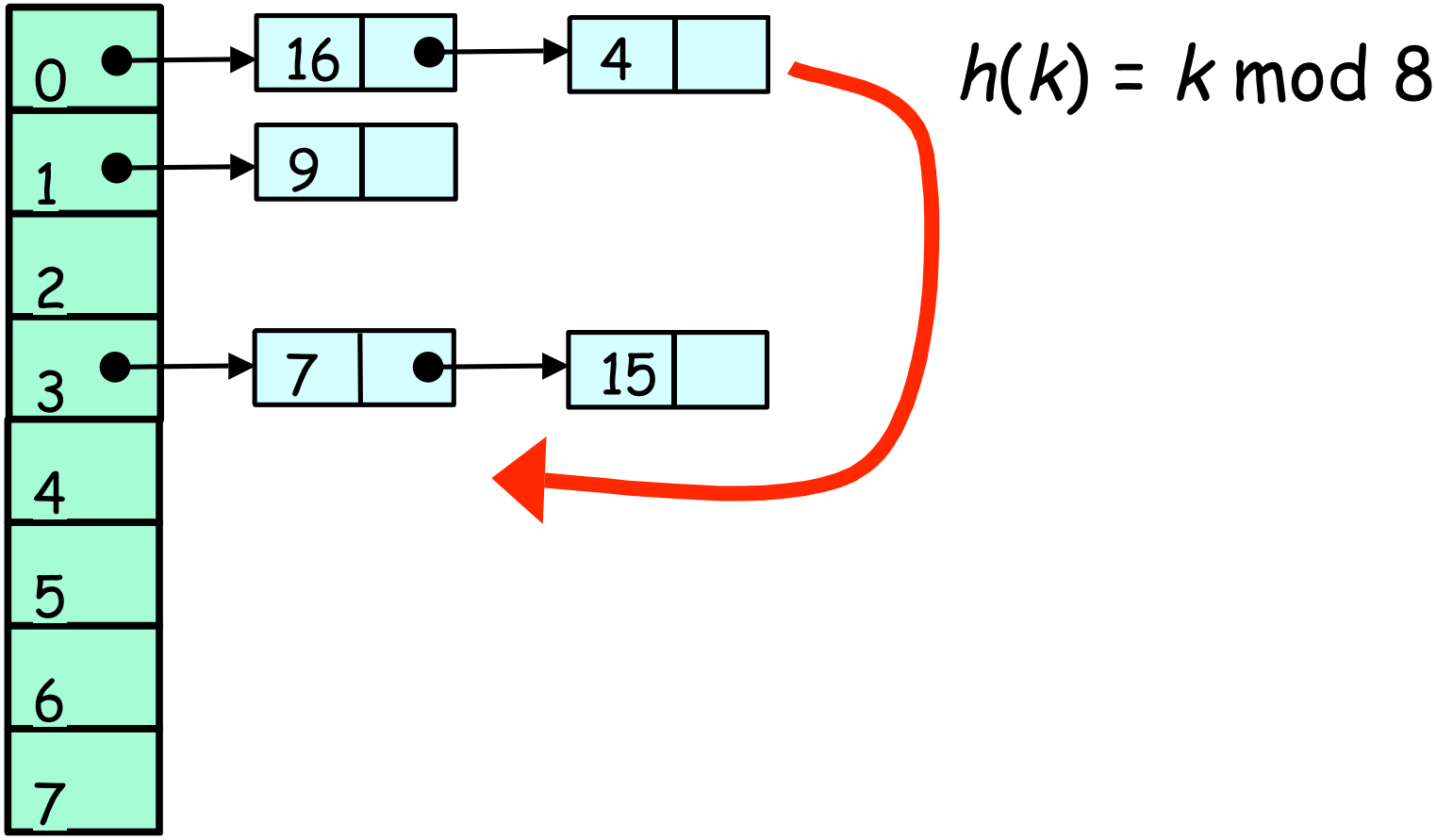


Item count: 5

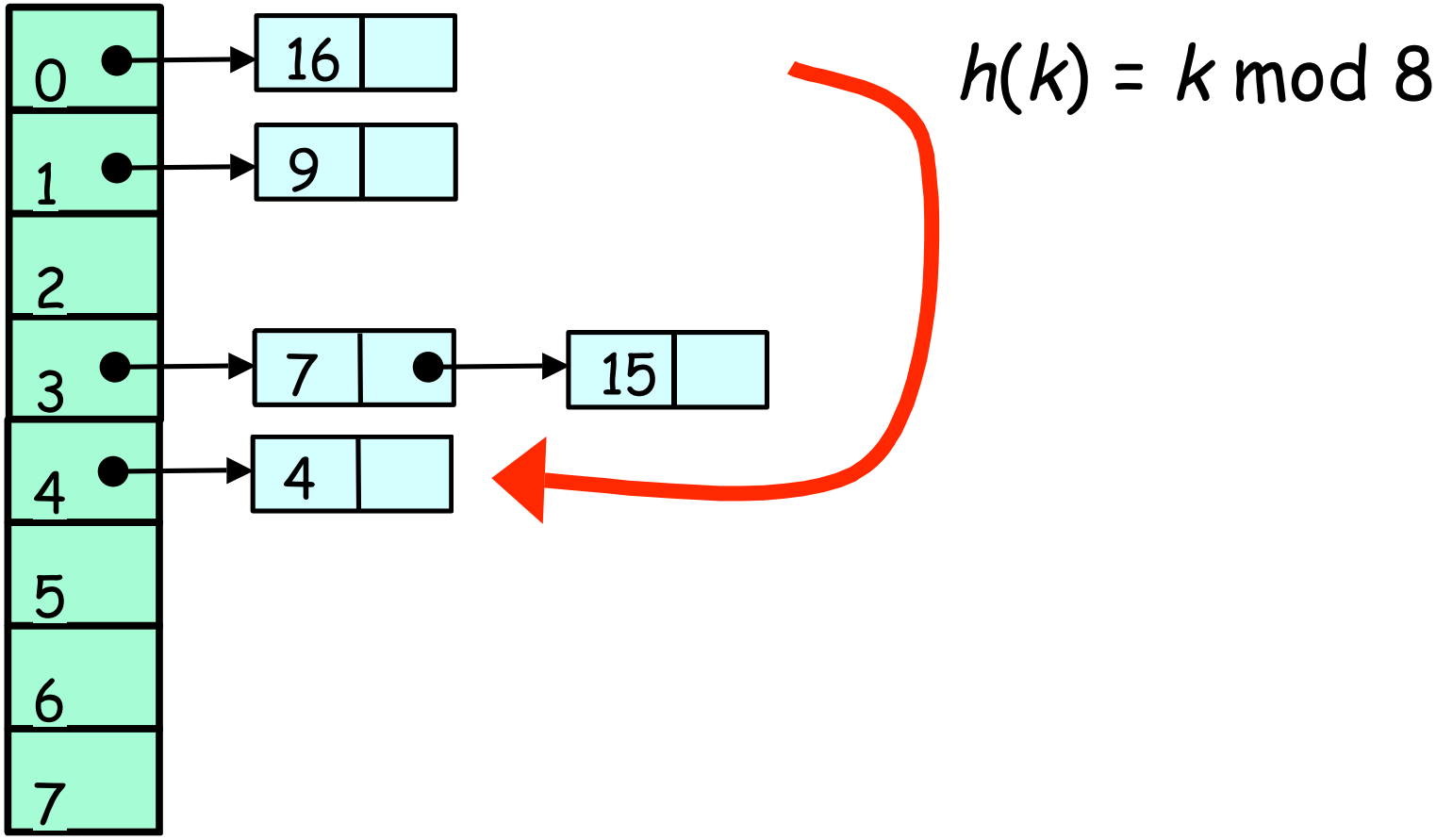
$h(k) = k \bmod 8$

Must redistribute items

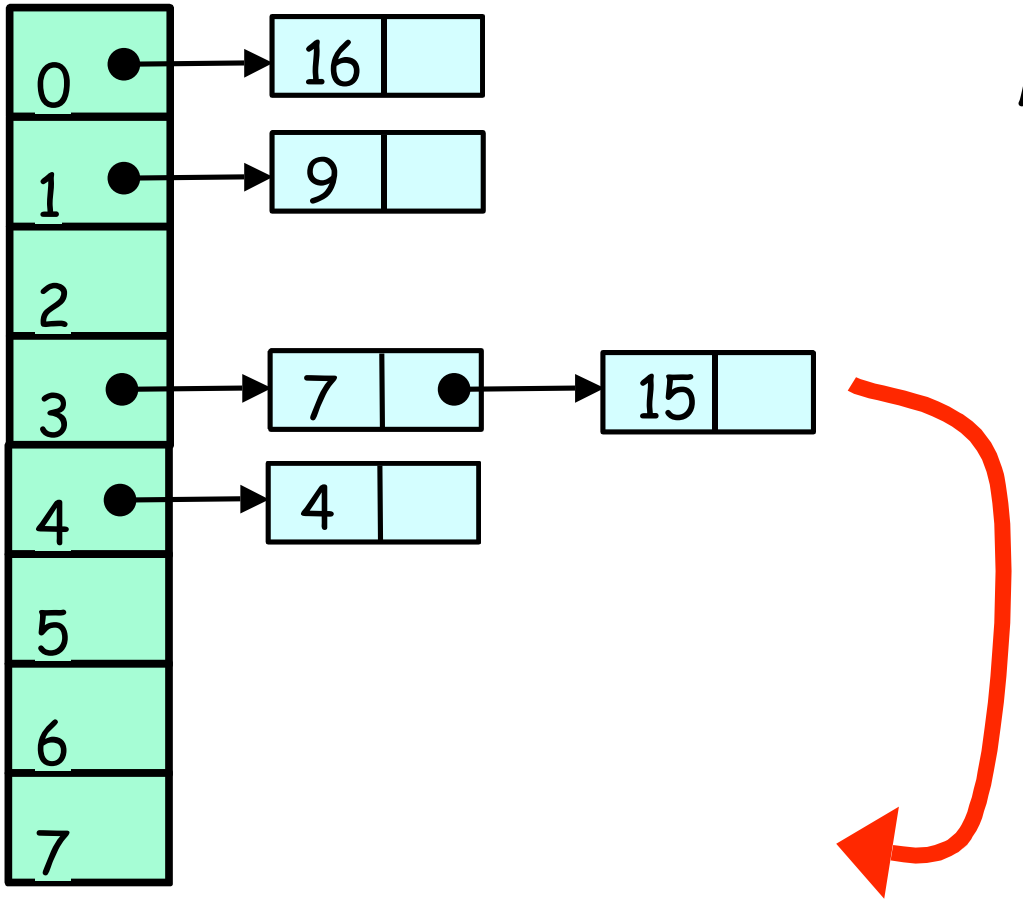
Moving the items



Moving the items

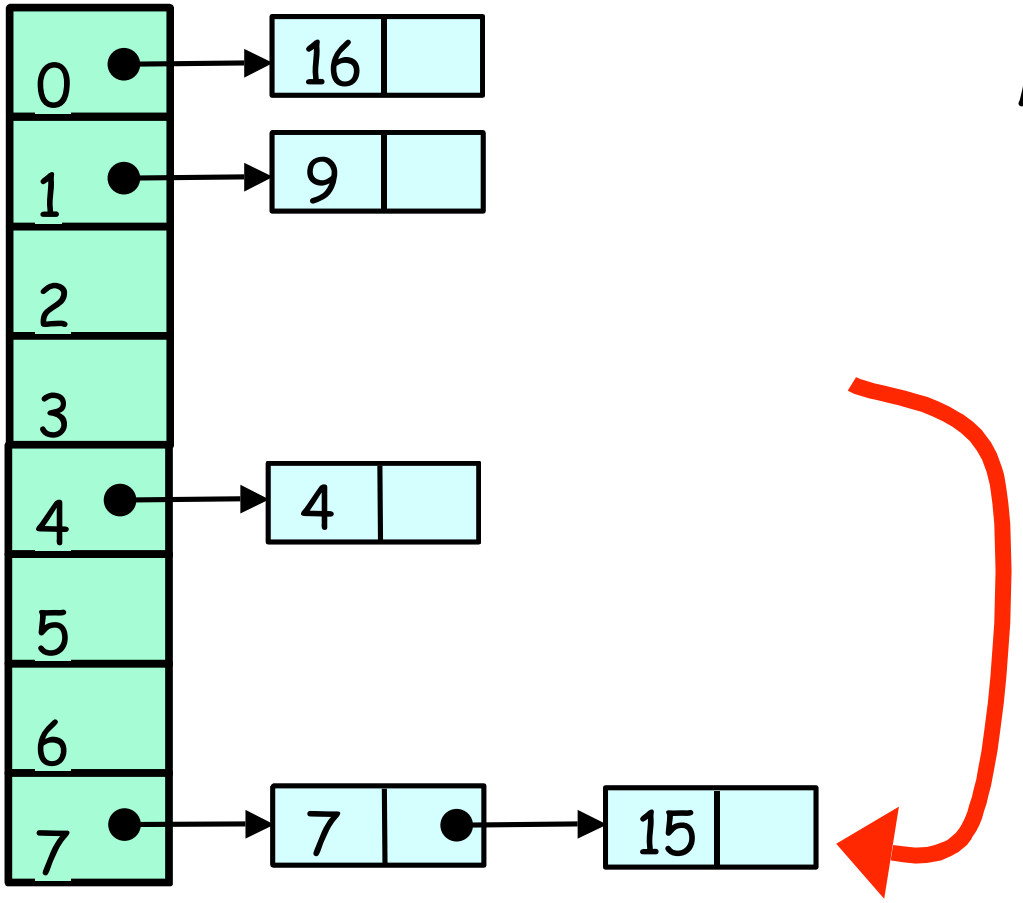


Moving the items



$h(k) = k \bmod 8$

Moving the items



$h(k) = k \bmod 8$

Hash Sets

- Set object implemented with hashing
- See also hash maps
- Here's one …

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();
  }
...
```

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 set implementation
- What's not to like?

No Brainer?

- We just saw a
  - Simple
  - Lock-free
  - Concurrent hash 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

Method Mix

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

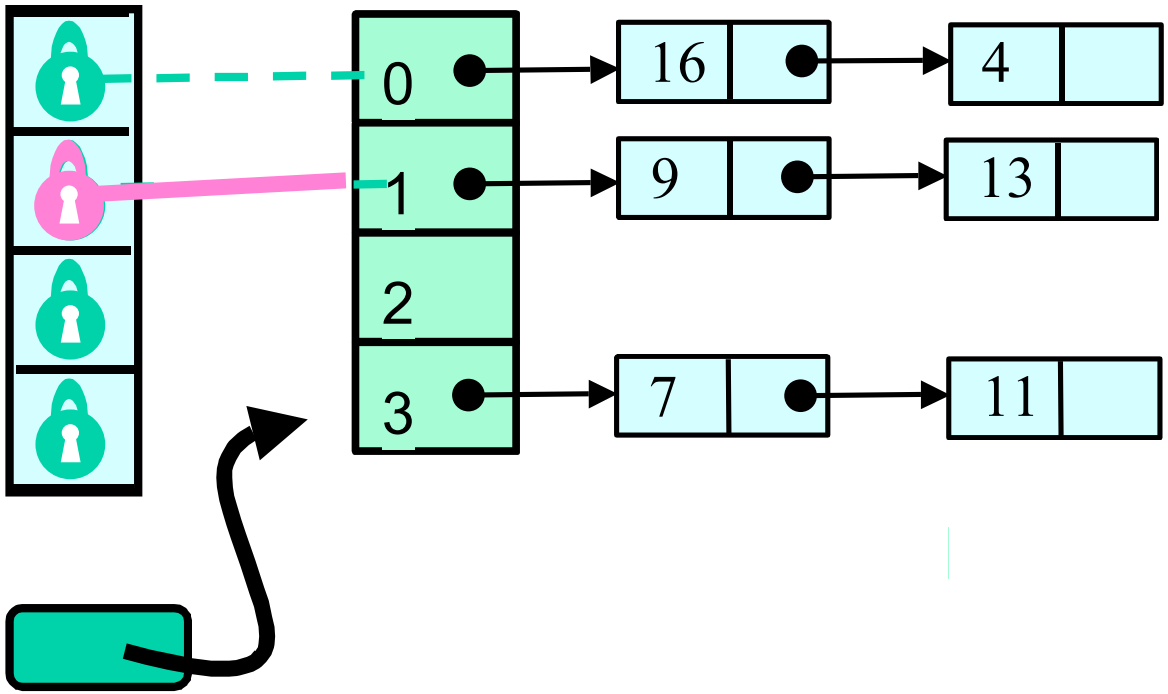
When to Resize?

- Many reasonable policies. Here's one.
- Bucket threshold
  - When ≥ _ buckets exceed this value
- Global threshold
  - When any bucket exceeds this value

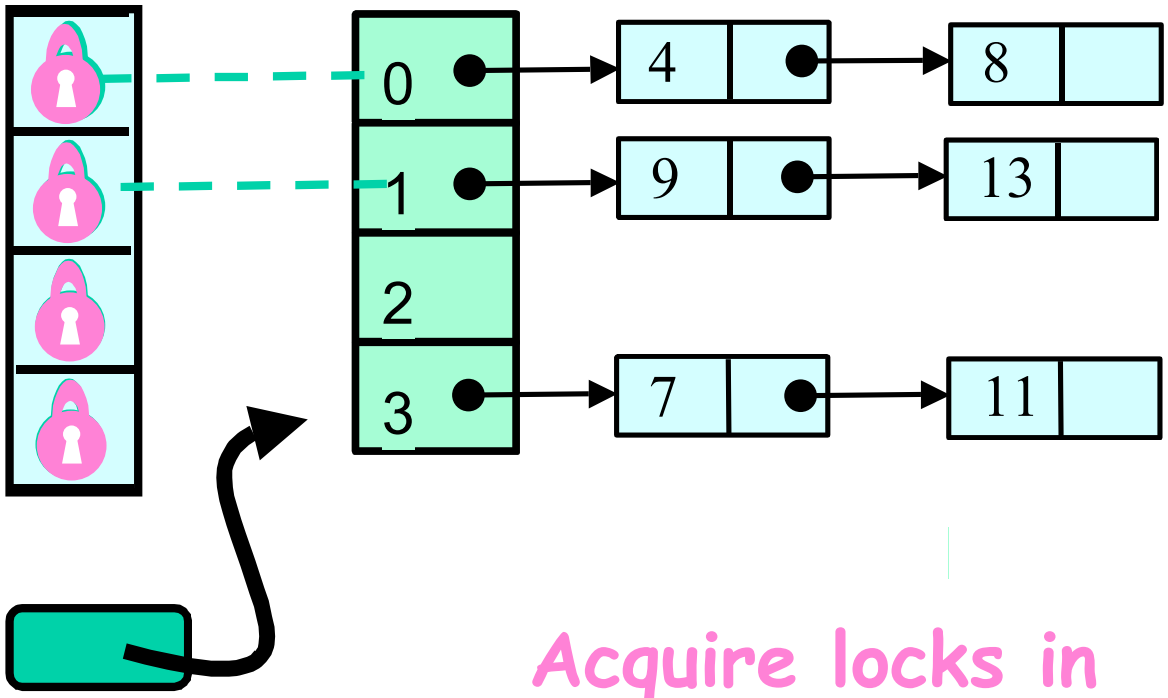
Coarse-Grained Locking

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

Fine-grained Locking

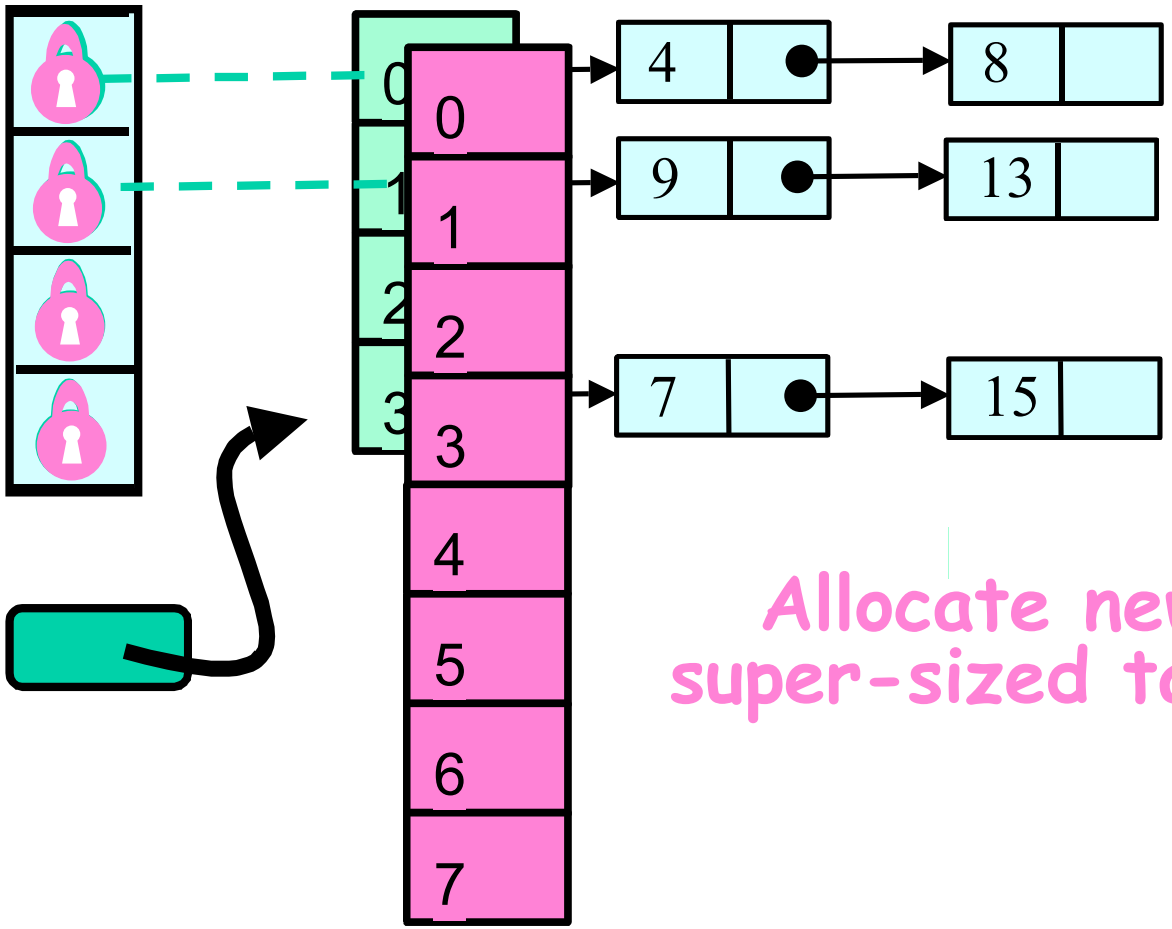


Resizing



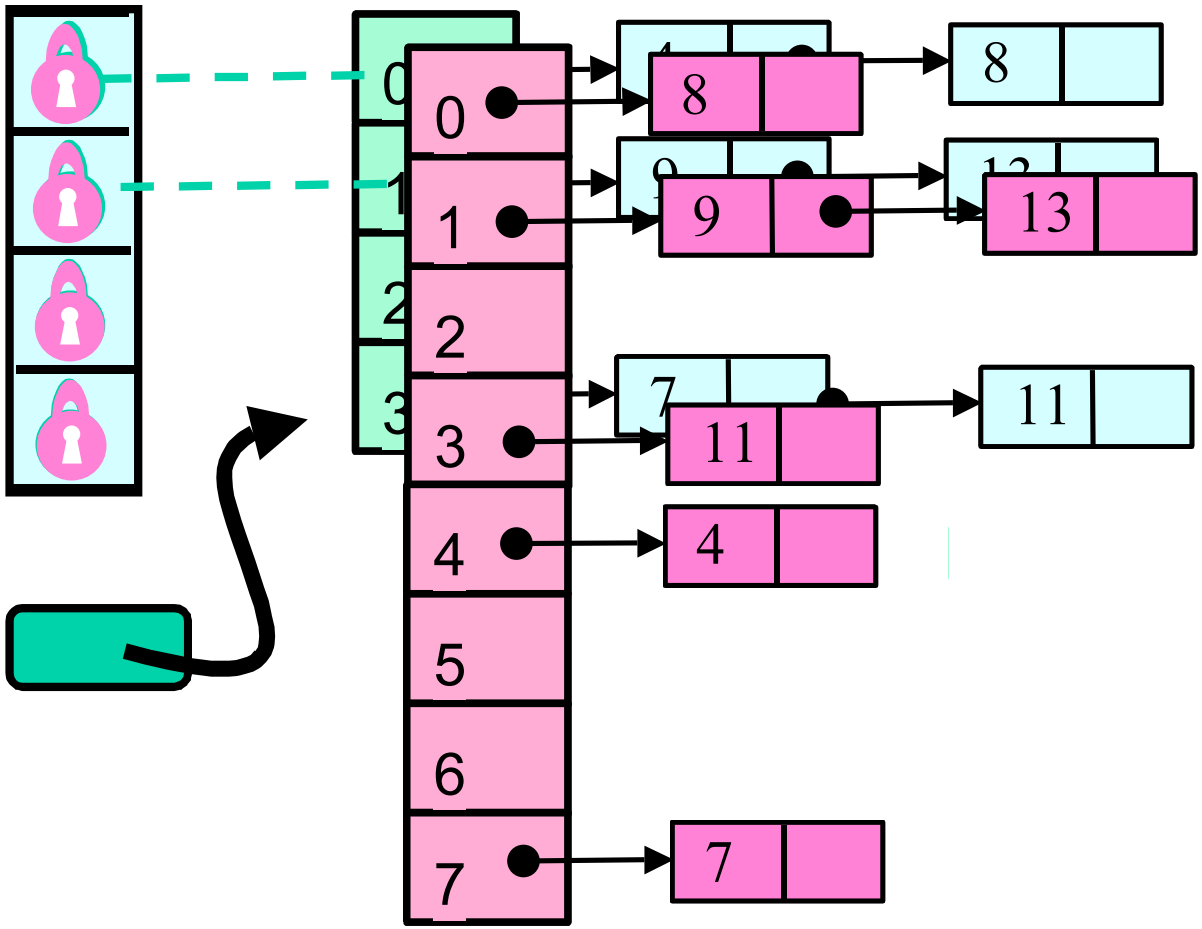
Acquire locks in ascending order

Fine-grained Locking



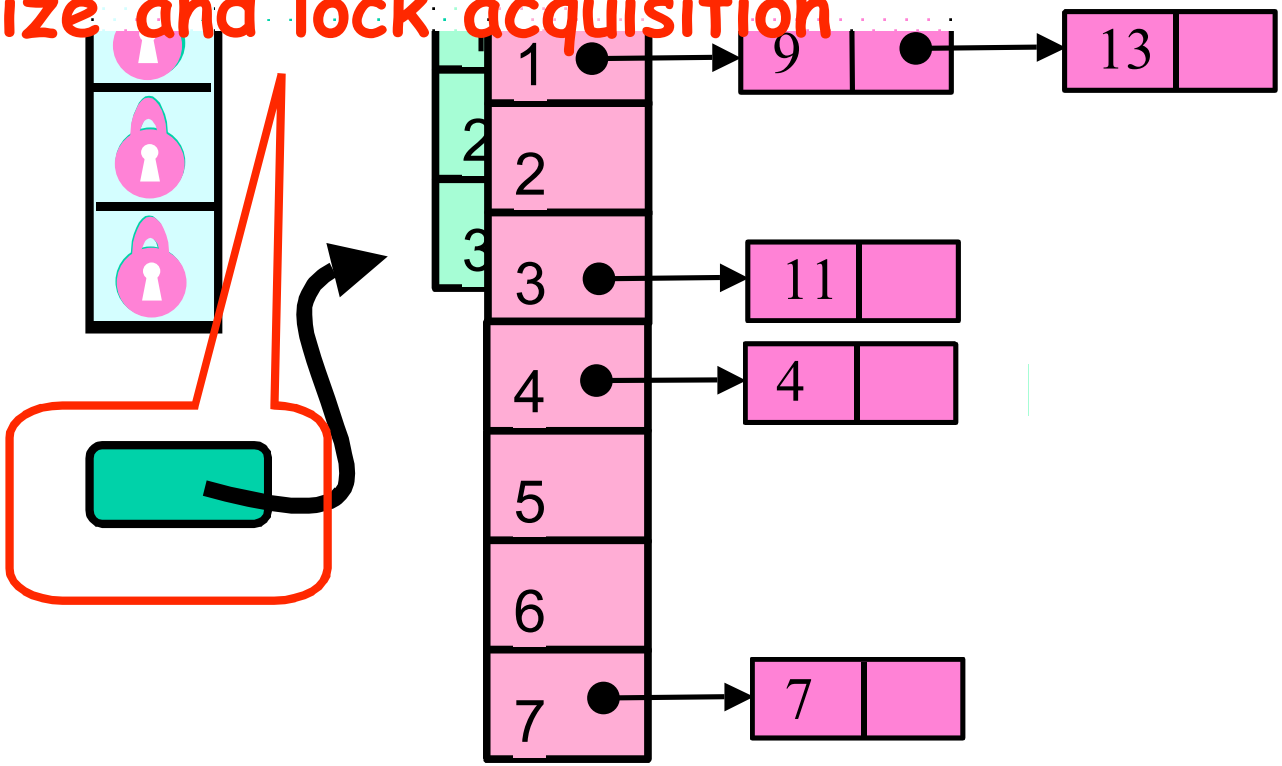
Allocate new super-sized table

Resizing

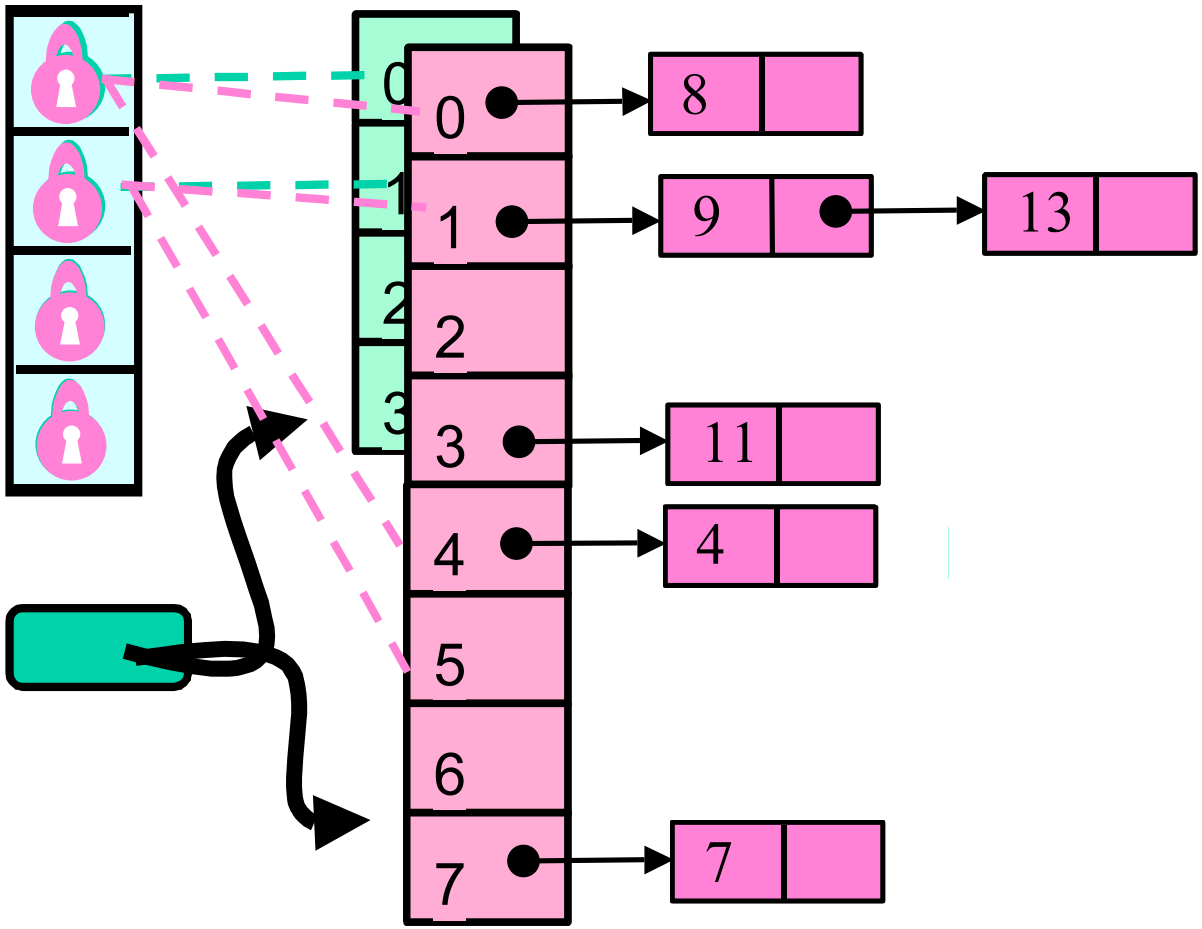


Resizing

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



Resizing



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?

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 FG
 protected Ran
 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

Fine-Grained Locking

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

Fine-Grained Locking

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

Which bucket?

Fine-Grained Locking

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

Which lock?

Fine-Grained Locking

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

Acquire lock

Fine-Grained Locking

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

Call 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();
 }}}
```

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
    sequen
 }}}
```

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

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();
 }}}
```

Acquire next lock

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
```

Check that no one else has resized

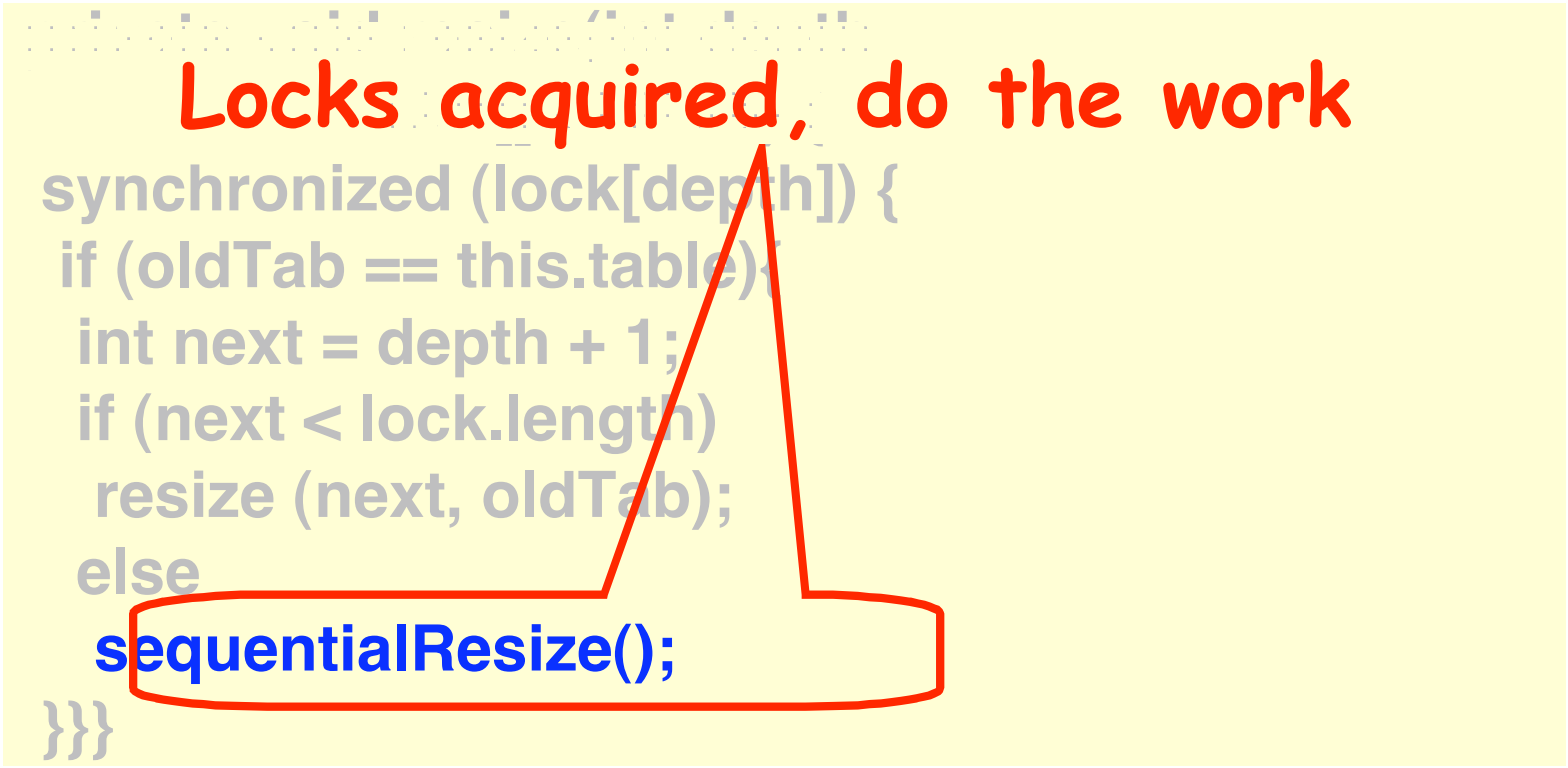
```
}}}
```

Fine-Grained Locking

Recursively acquire next lock

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

Fine-Grained Locking



Fine-Grained Locks

- We can resize the table
- But not the locks
- Debatable whether method calls are linear-time in presence of contention

...

Insight

- The contains() method
  - Does not modify any fields
  - Why should multiple contains() methods conflict?

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

- 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

Read/Write Lock

- Satisfies safety properties
  - If readers > 0 then writer == false
  - If writer = true then readers == 0
- Liveness?
  - Lots of readers …
  - Writers locked out?

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 the hash table is the hard part
- Fine-grained locks
  - Range locks (not resized)
- Read/Write locks
  - FIFO property tricky

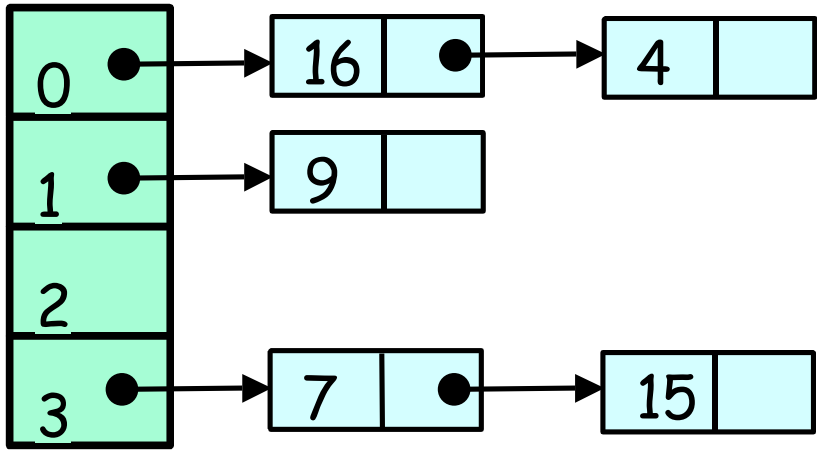
Optimistic Synchronization

- If the contains() method
  - Scans without locking
- If it finds the key
  - OK to return true
  - Actually requires a proof ….
- What if it doesn't find the key?

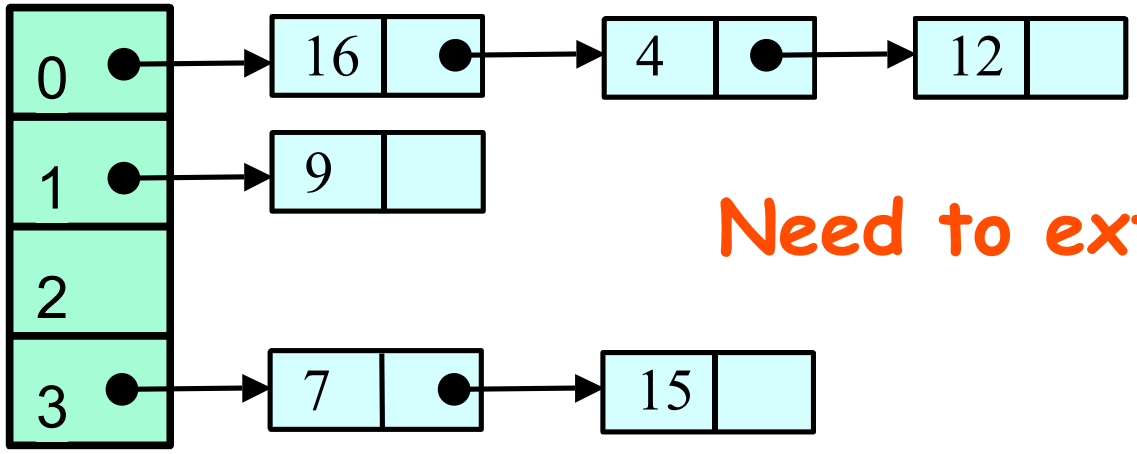
Optimistic Synchronization

- If it doesn't find the key
  - May be victim of resizing
- Must try again
  - Getting a read lock this time
- Makes sense if
  - Resizes are rare
  - Keys are present

Lock-Free Resizing Problem

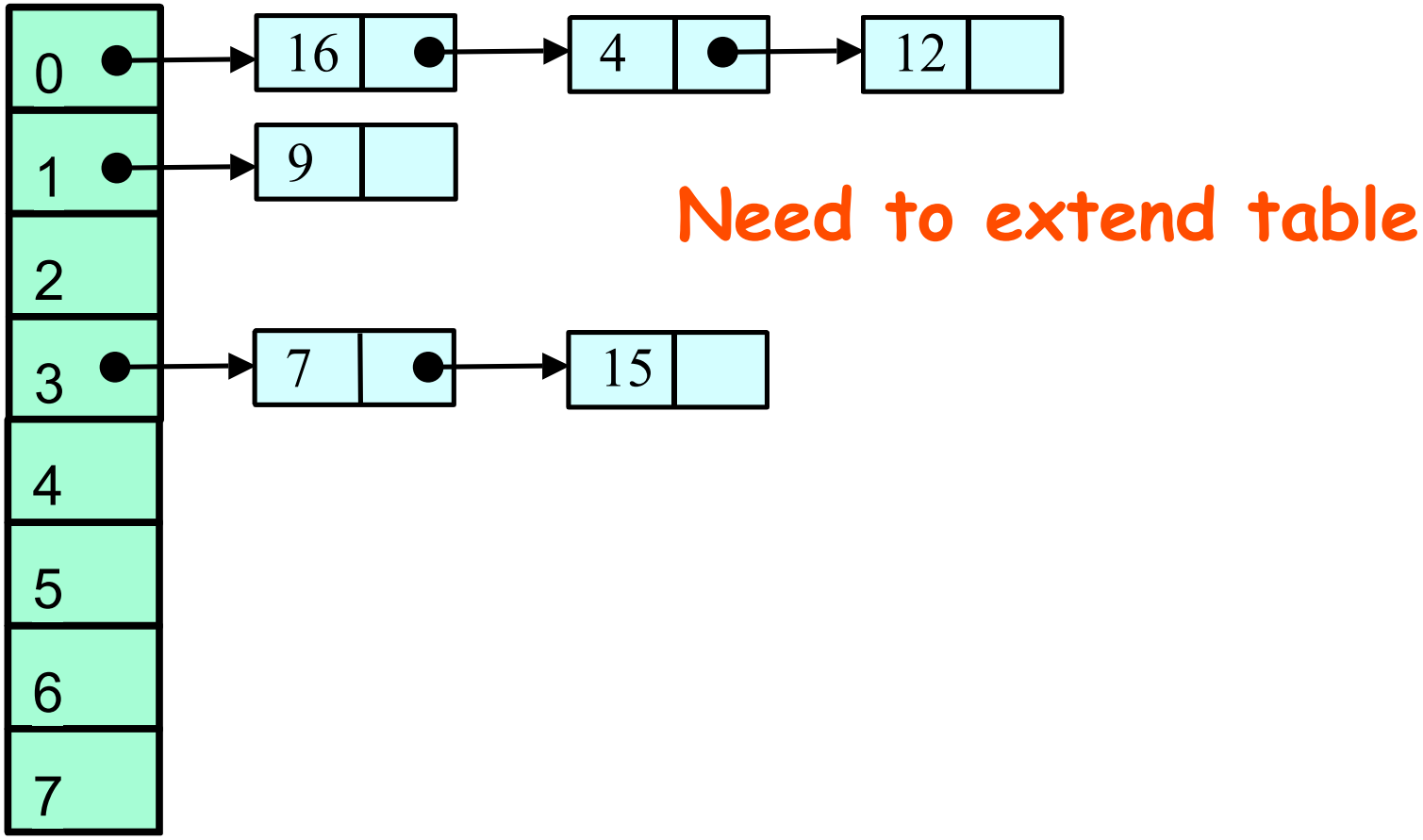


Lock-Free Resizing Problem



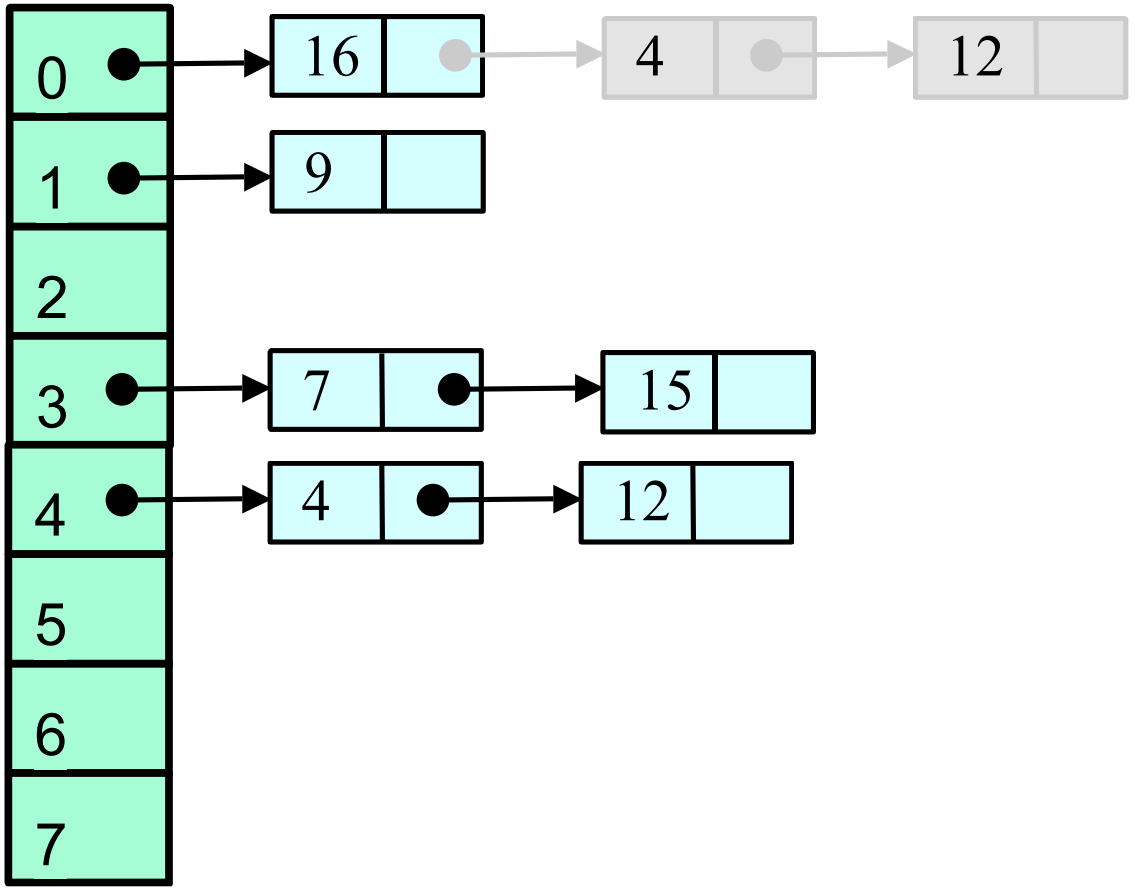
Need to extend table

Lock-Free Resizing Problem

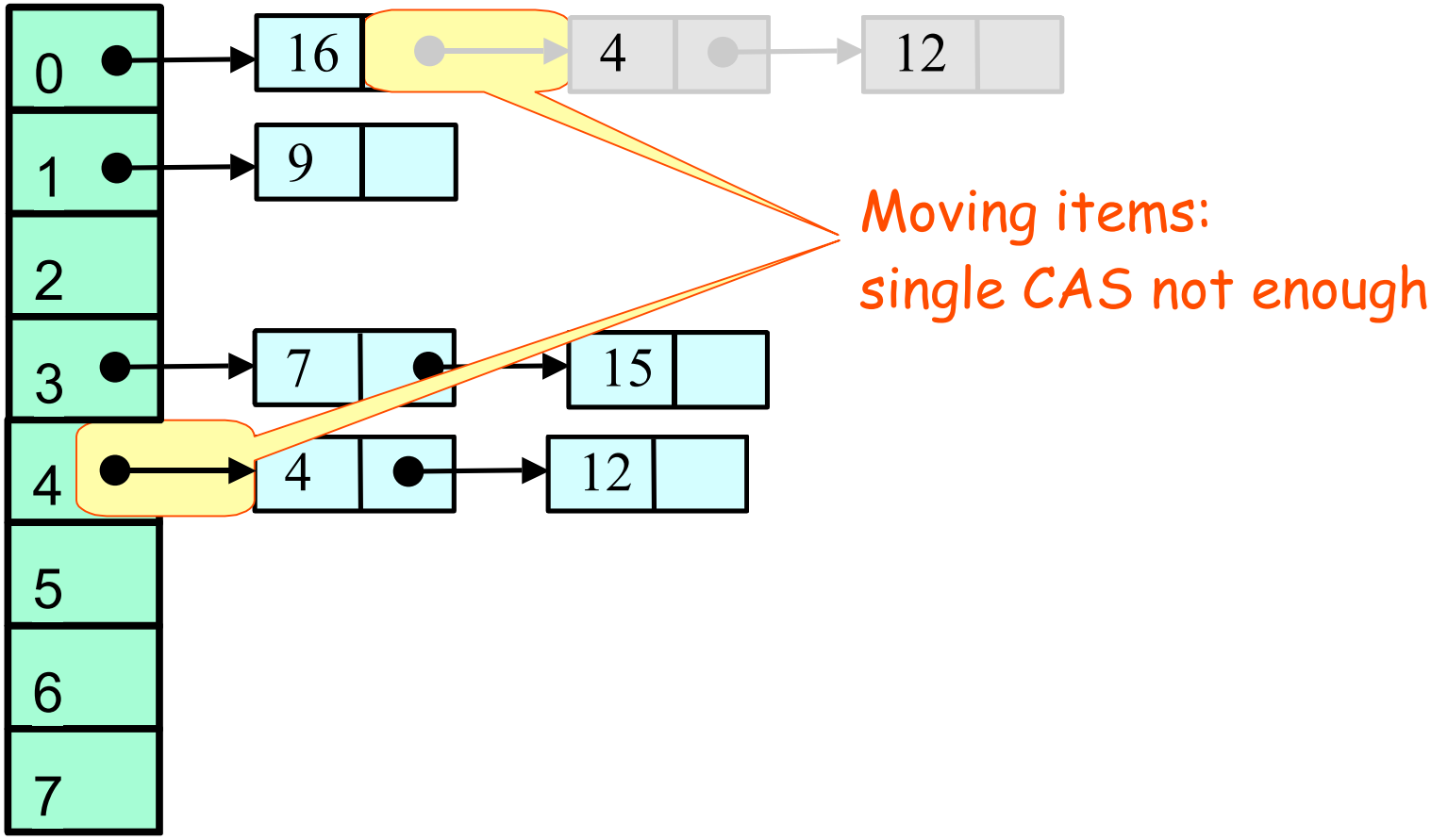


Need to extend table

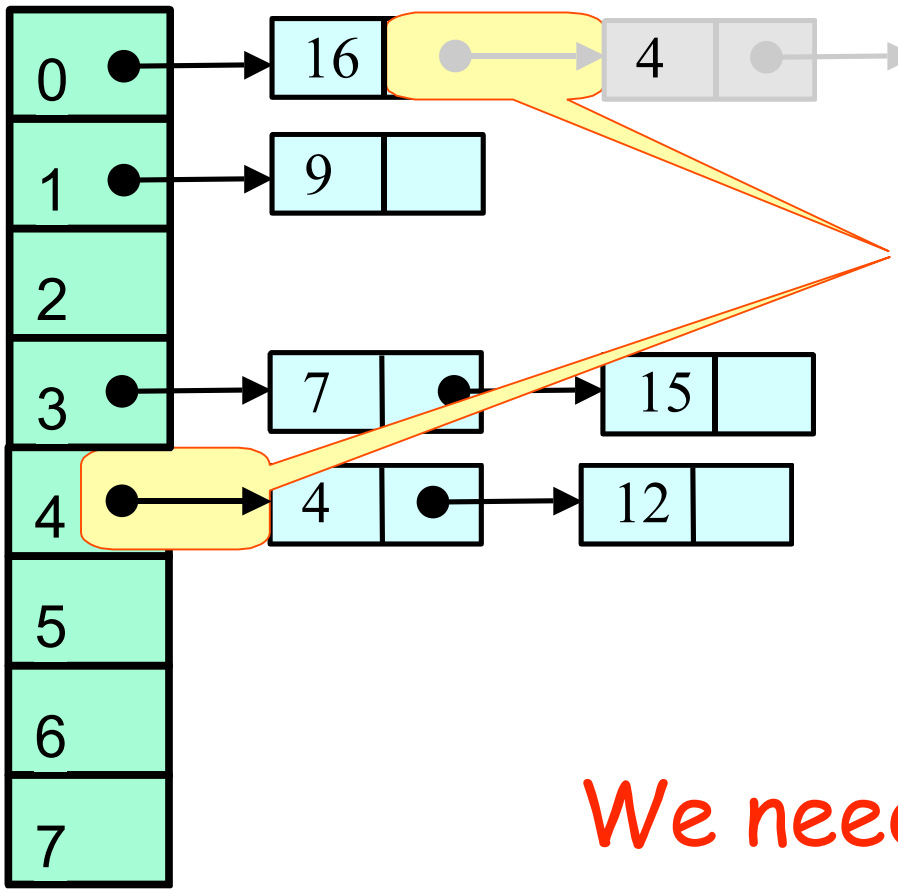
Lock-Free Resizing Problem



Lock-Free Resizing Problem



Lock-Free Resizing Problem

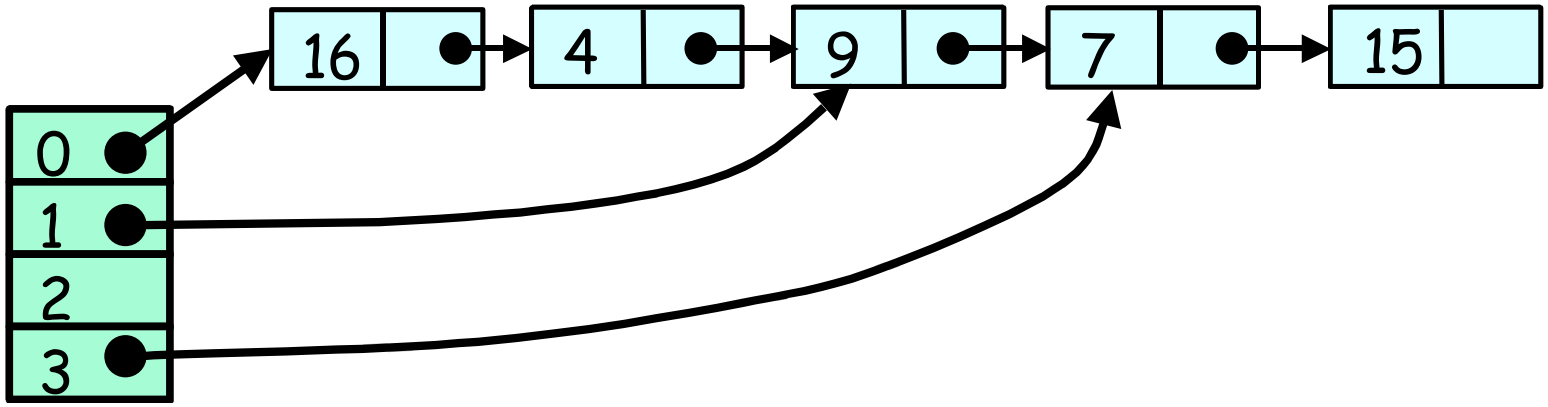


Moving items:
single CAS not enough

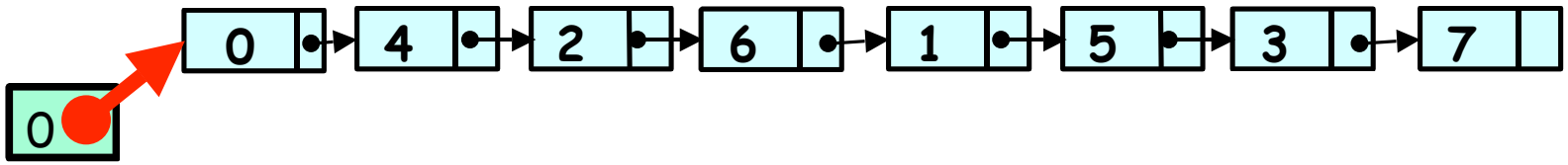
We need a new idea…

Don't move the items

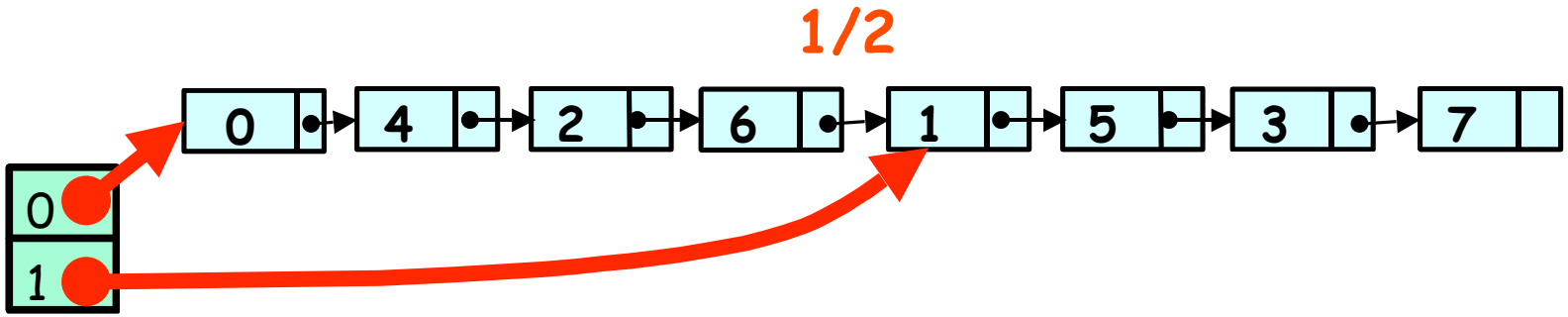
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



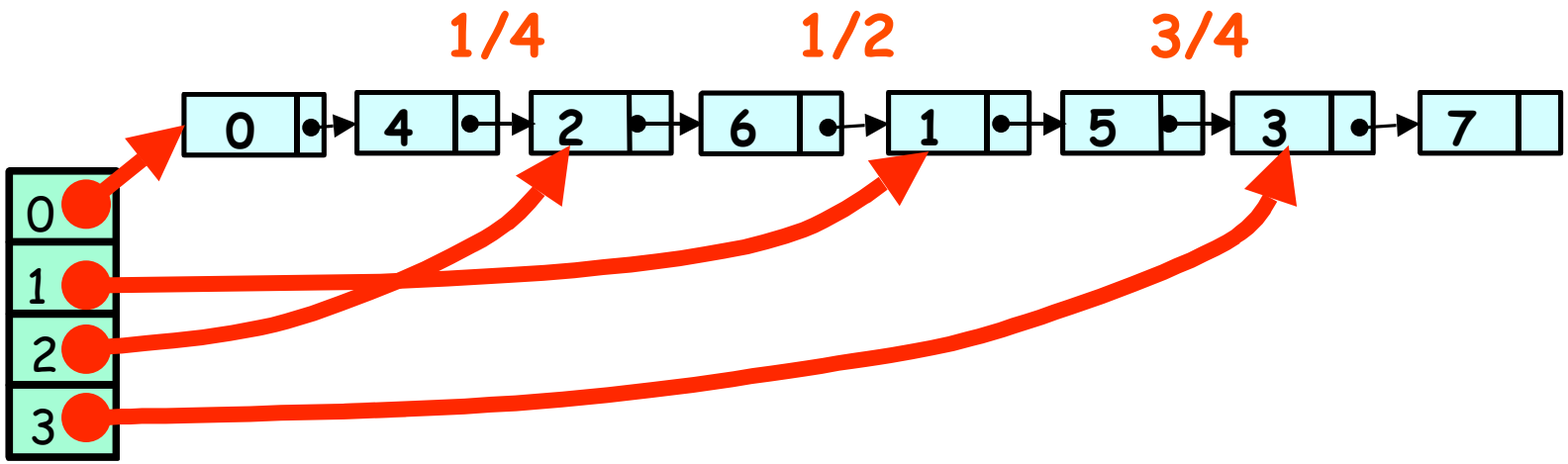
Recursive Split Ordering



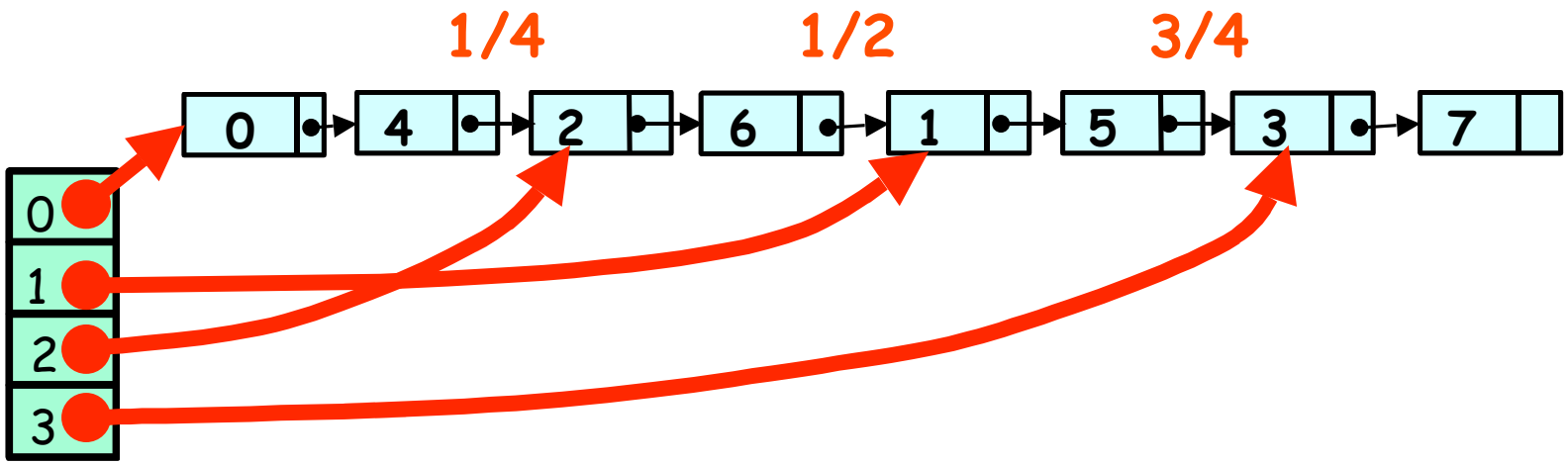
Recursive Split Ordering



Recursive Split Ordering

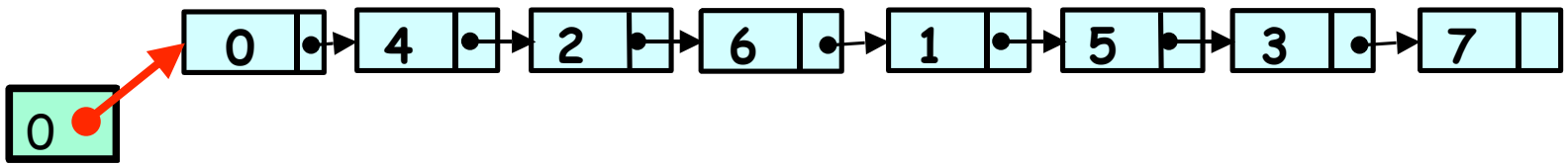


Recursive Split Ordering

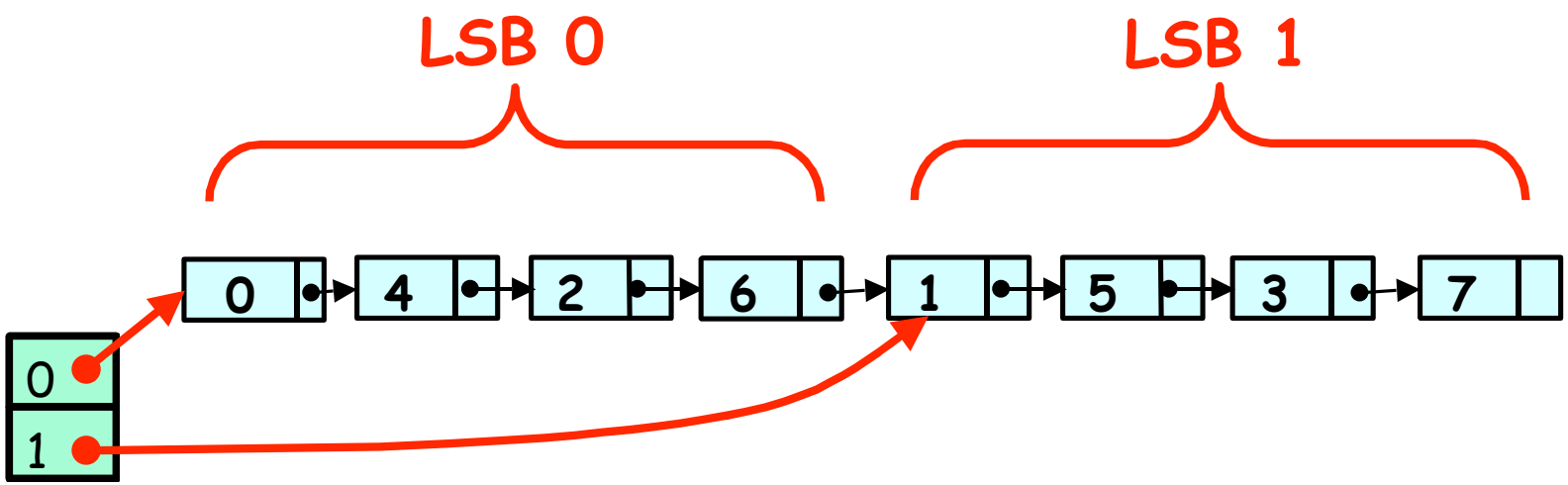


List entries sorted in order that allows recursive splitting. How?

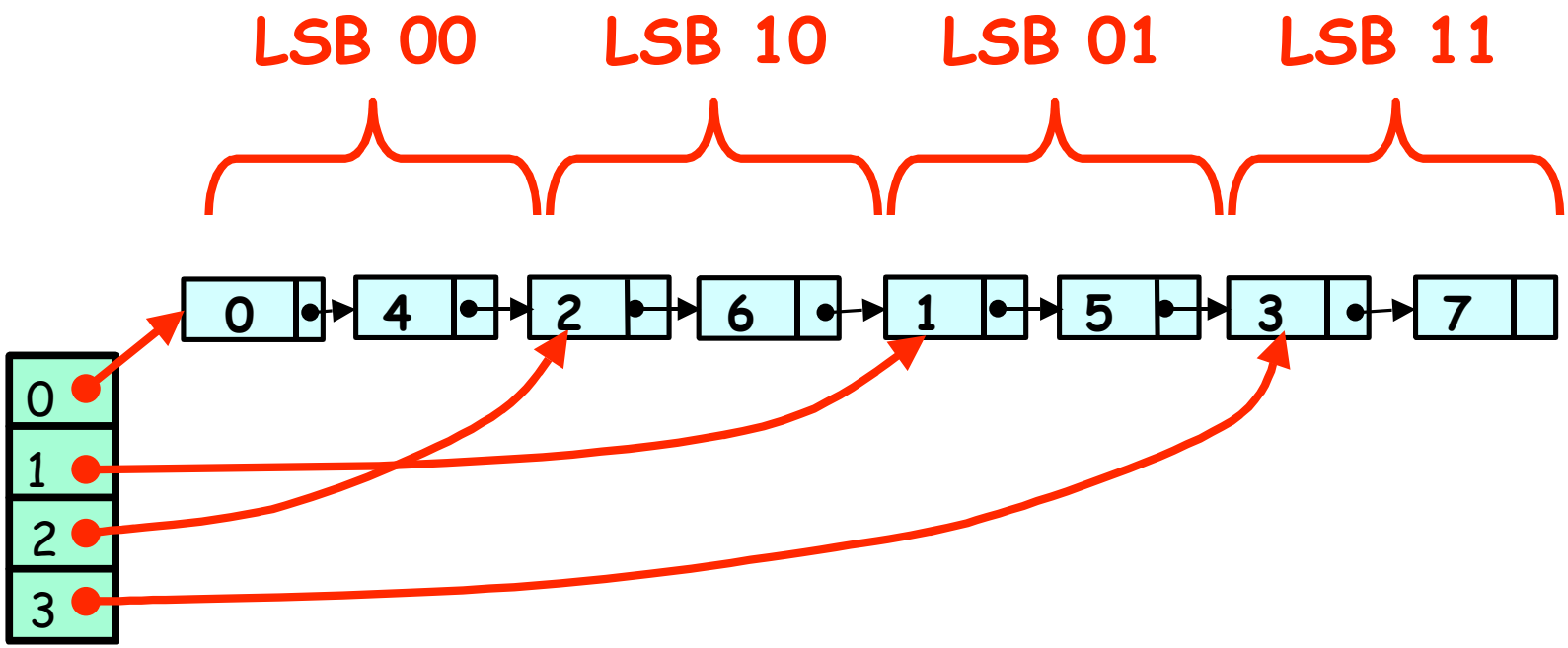
Recursive Split Ordering



Recursive Split Ordering



Recursive Split Ordering



Split-Order

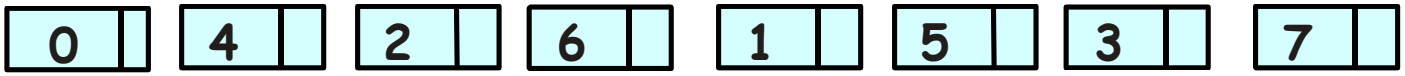
- If the table size is 2^i,
  - Bucket b contains keys k
    - $k = b \pmod{2^i}$
  - bucket index is key's i LSBs
  - (least significant bits)

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

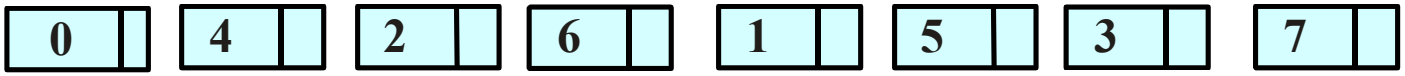
A Bit of Magic

Real keys:

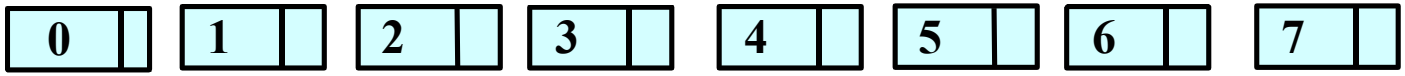


A Bit of Magic

Real keys:

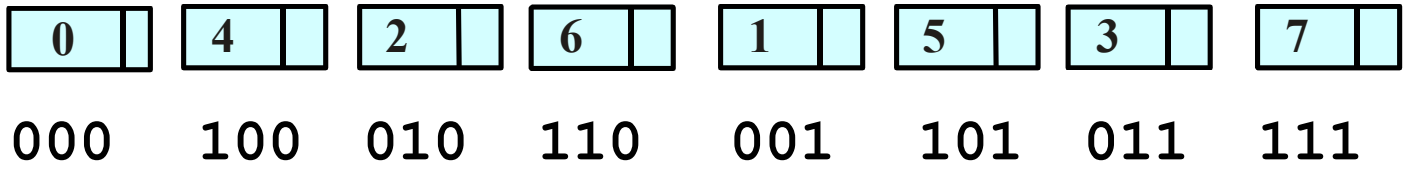


Split-order:

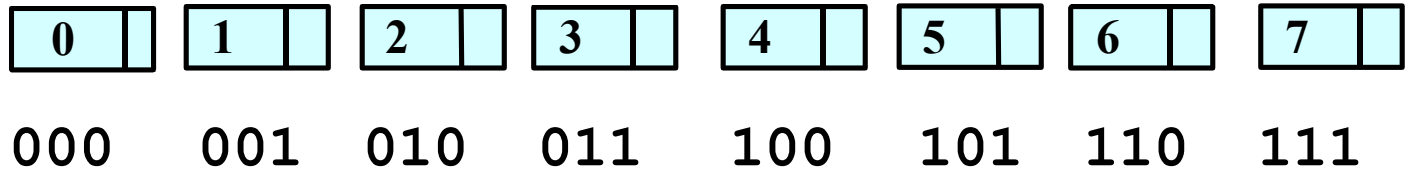


A Bit of Magic

Real keys:



Split-order:



A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

Split-order:

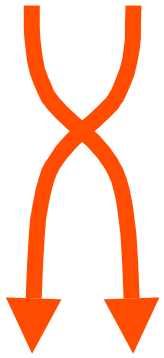
000 001 010 011 100 101 110 111

A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

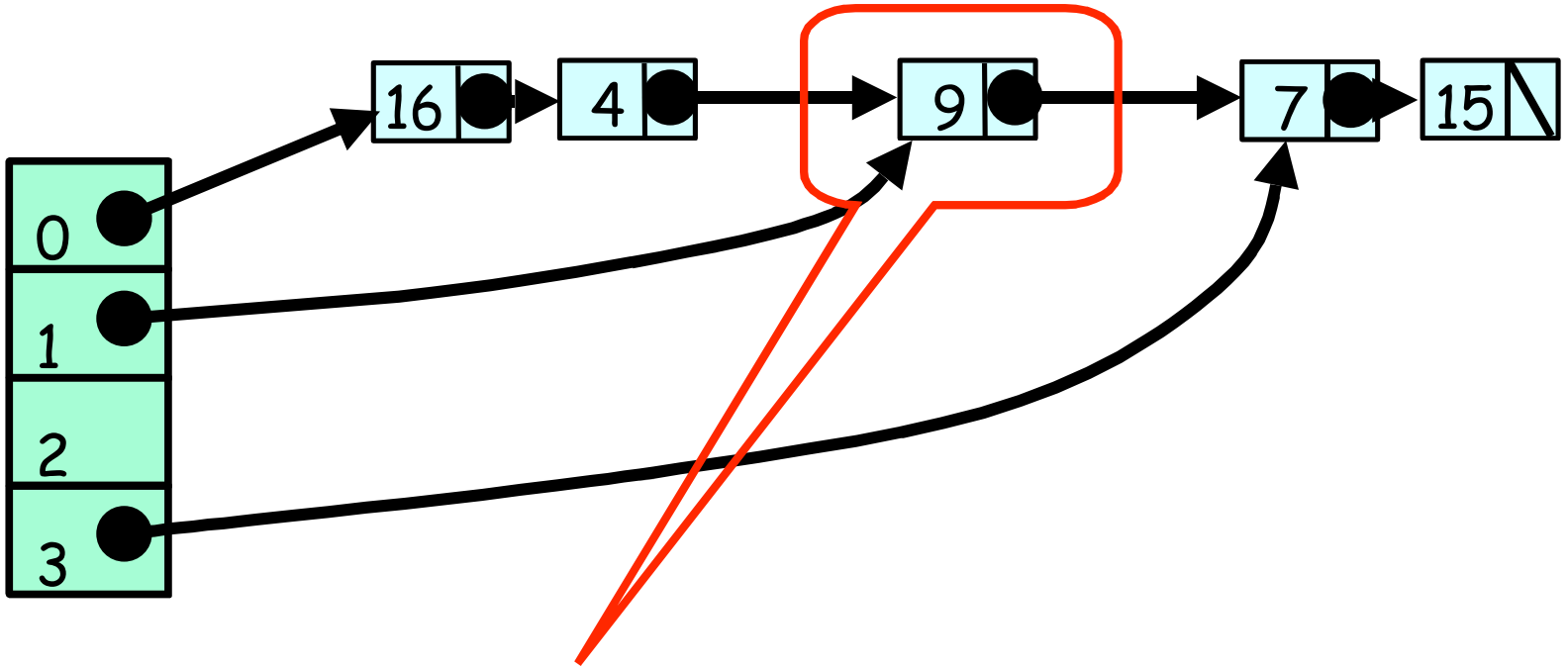
Split-order:



000 001 010 011 100 101 110 111

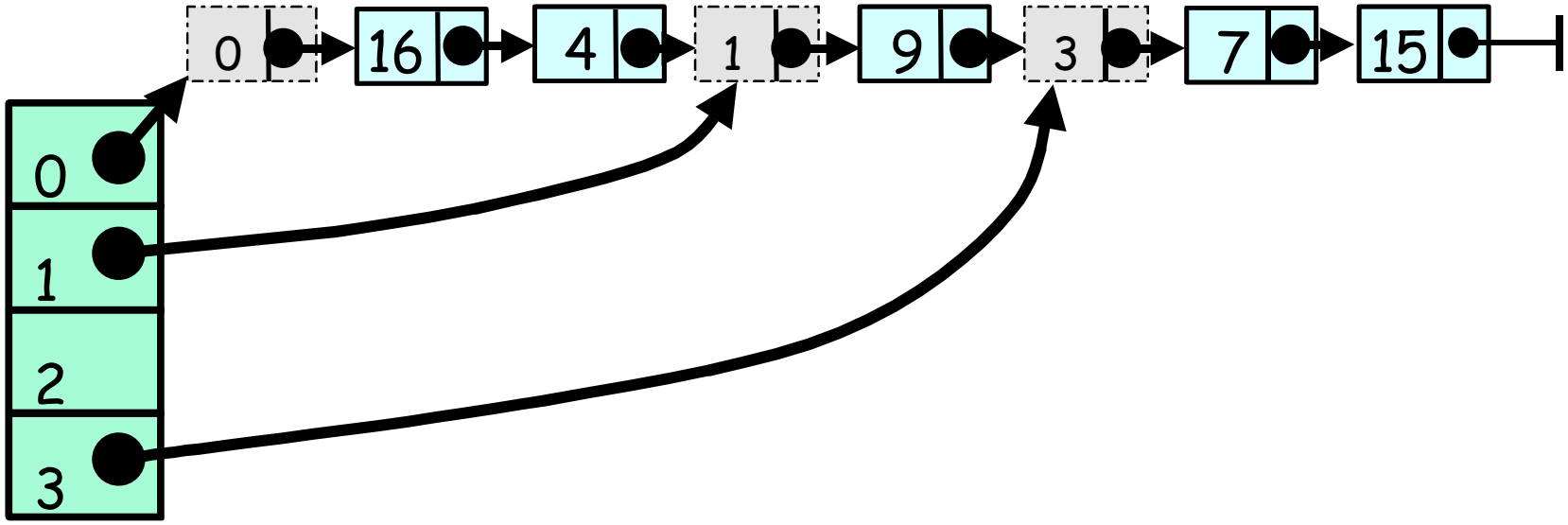
Just reverse the order of the key bits

Sentinel Nodes



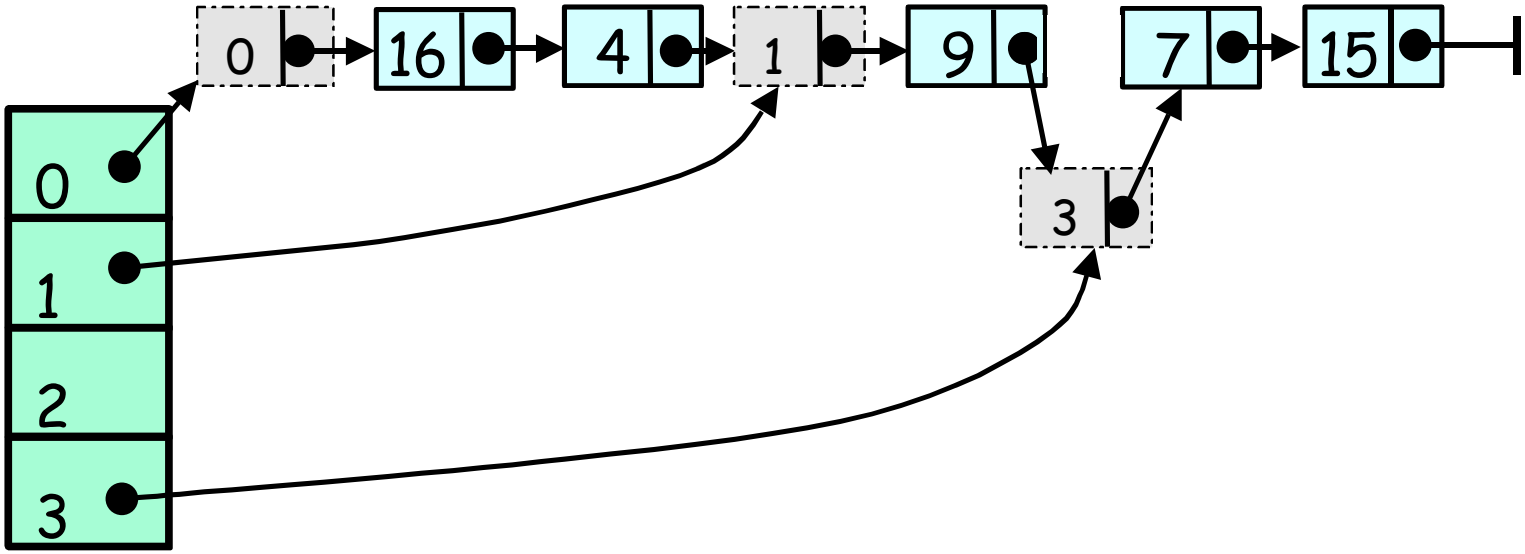
Problem: how to remove a node pointed by 2 sources using CAS

Sentinel Nodes



Solution: use a Sentinel node for each bucket

Lazy Initialization of Buckets



Sentinel vs Regular Keys

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

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
   Locl

}
```

Change: add takes key argument

Lock-Free List

Inserts sentinel with key if not already present …

```
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)

```
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();
    table[
    setSi
  }
```

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();
    table
    setSi
  }
```

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();
    tableS
    setSi
  }
```

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();
    tableS
    setSi
  }
```

Track set size so we know when to resize

Fields

Initially use 2 buckets and size is zero

```
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);
}
```

Add() Method

```
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() Method

```
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() Method

```
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() Method

```
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;
  re
  re
}
```

Get pointer to bucket's sentinel, initializing if necessary

Add() Method

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() Method

No change? We're done.

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

Add() Method

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

Time to resize?

Resize

- 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

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

```
initializeBucket(parent);
int key = makeSentinelKey(bucket);
LockFreeList list =
 new LockFreeList(table[parent],
                  key);
}
```

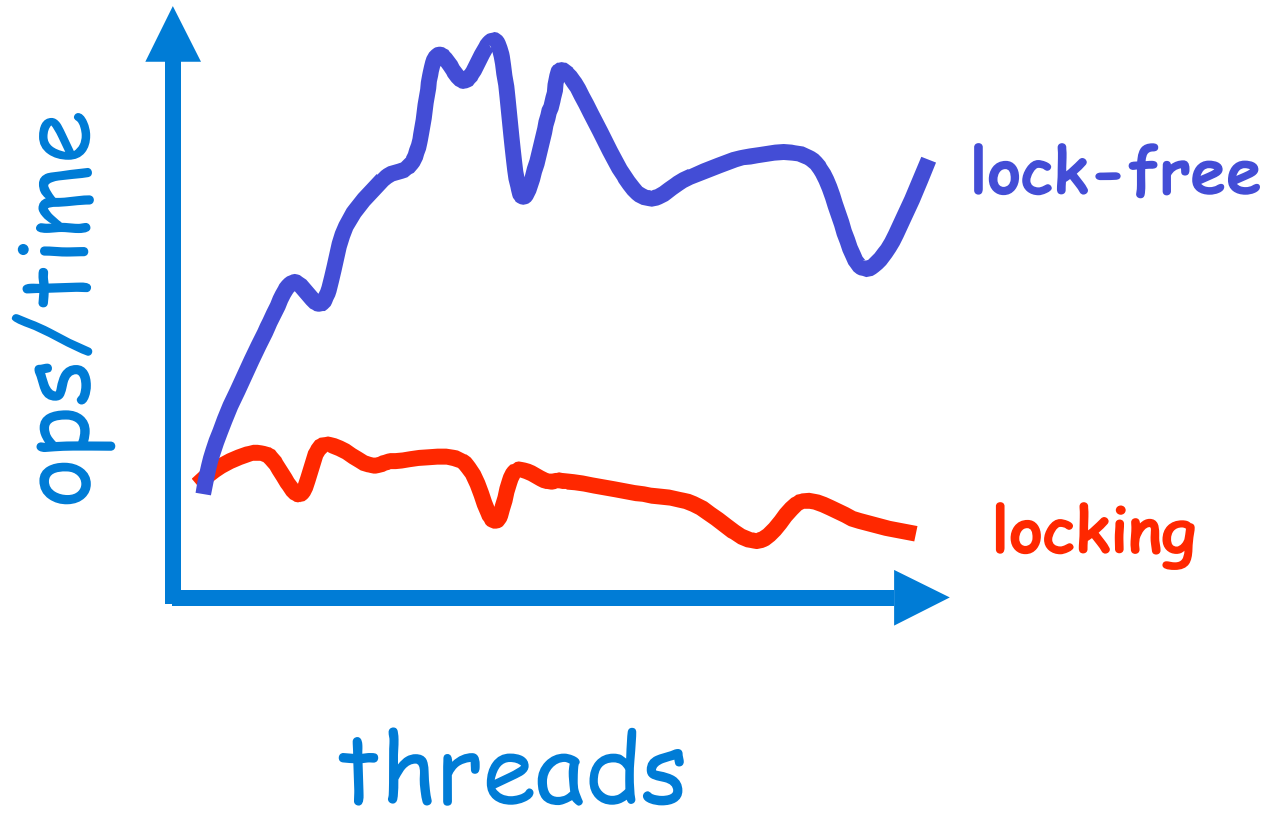
Correctness

- Linearizable concurrent set implementation
- Theorem: $O(1)$ expected time
  - No more than $O(1)$ items expected between two dummy nodes on average
  - Lazy initialization causes at most $O(1)$ expected recursion depth in initializeBucket()

Empirical Evaluation

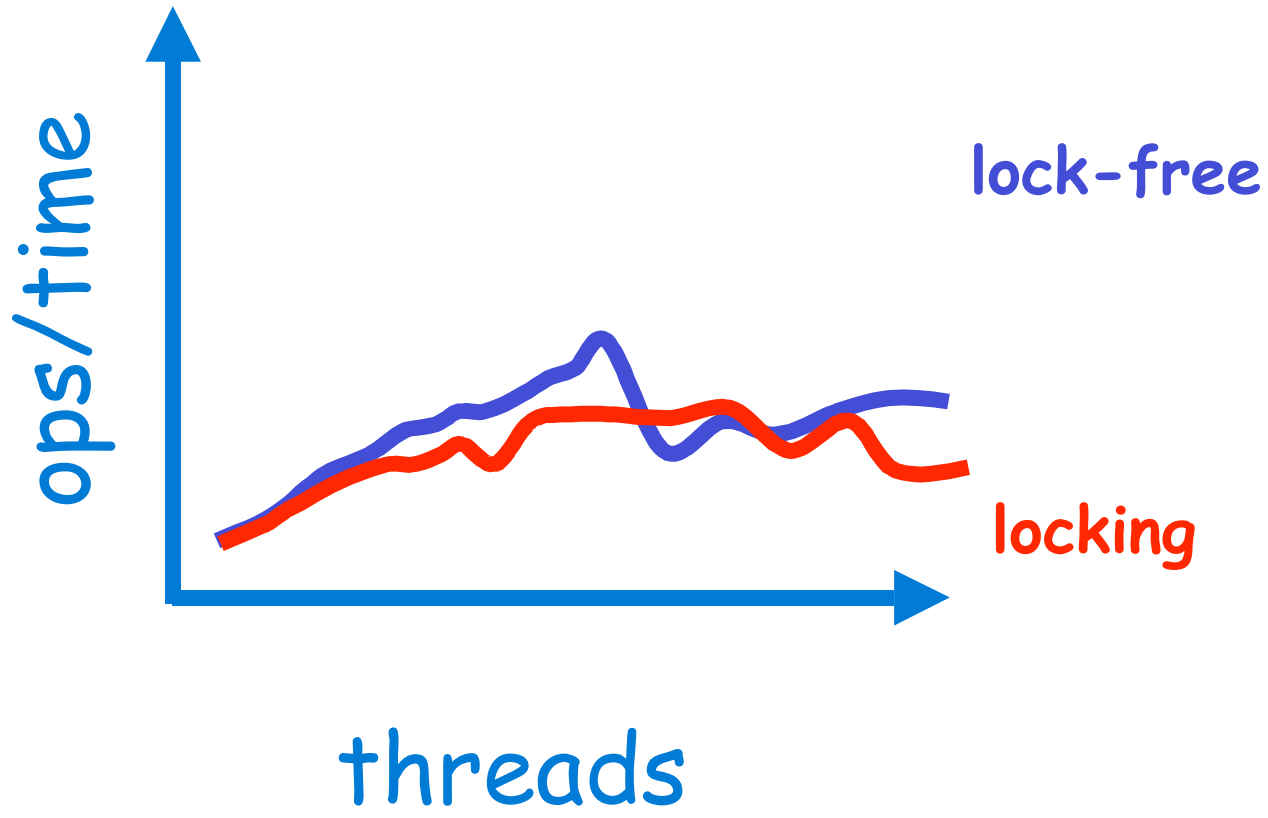
- On a 72-processor Sun Fire™ 15K
- Lock-Free vs. fine-grained optimistic
- In a non-multiprogrammed environment
- 10^6 operations: **88%** *contains()*, **10%** *add()*, **2%** *remove()*

Work = 0



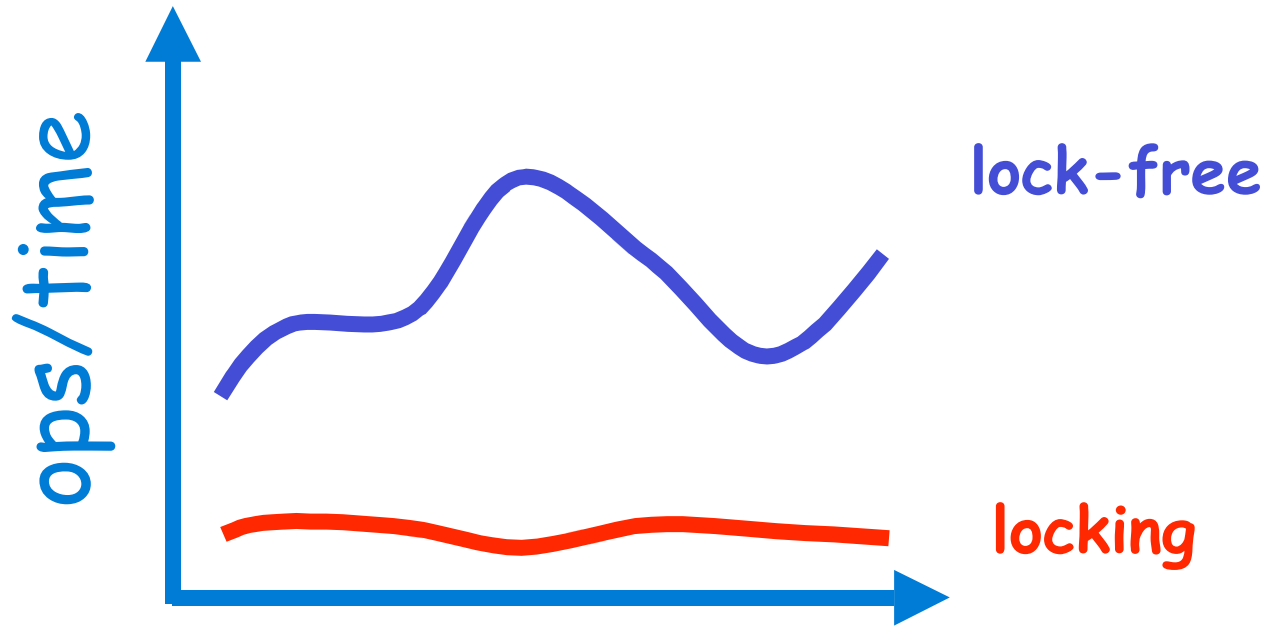
Adapted from Shalev & Shavit 2003

Work = 500



Adapted from Shalev & Shavit 2003

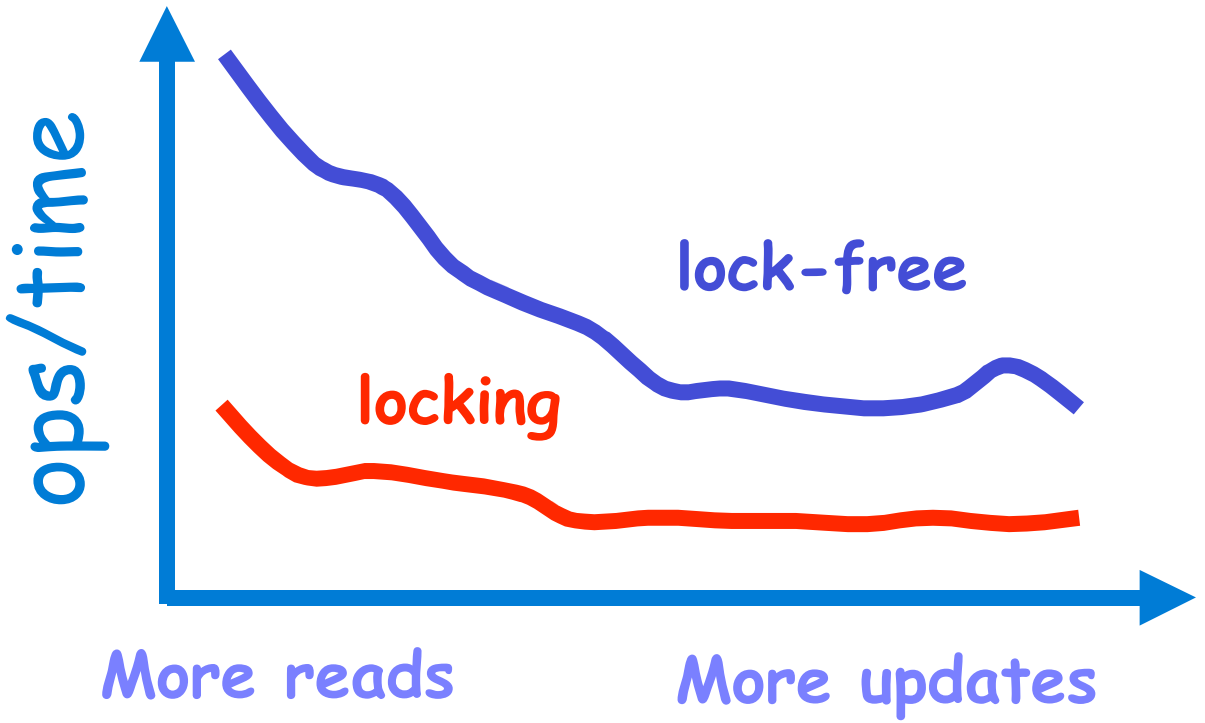
Expected Bucket Length



64 threads

Adapted from Shalev & Shavit 2003

Varying The Mix



64 threads

Adapted from Shalev & Shavit 2003

Summary

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