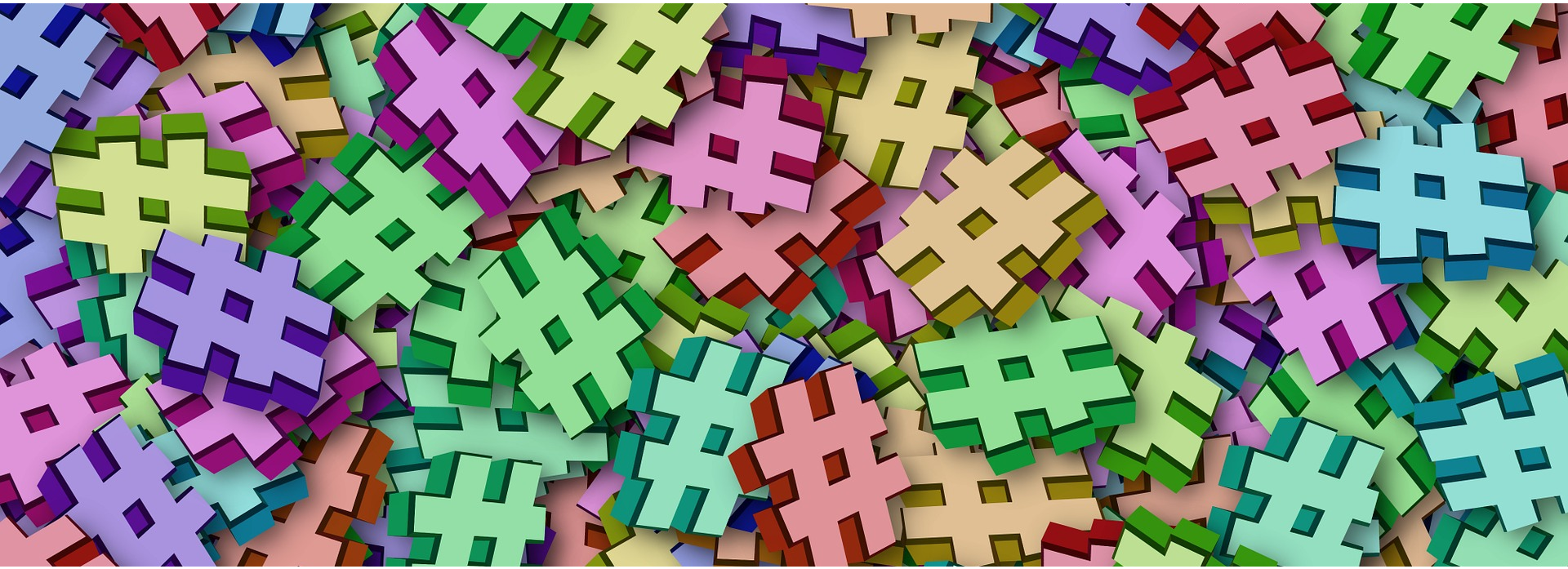
Concurrent Hashing



Christof Fetzer, TU Dresden

Based on slides by Maurice Herlihy and Nir Shavit

Linked Lists

- 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 missing?

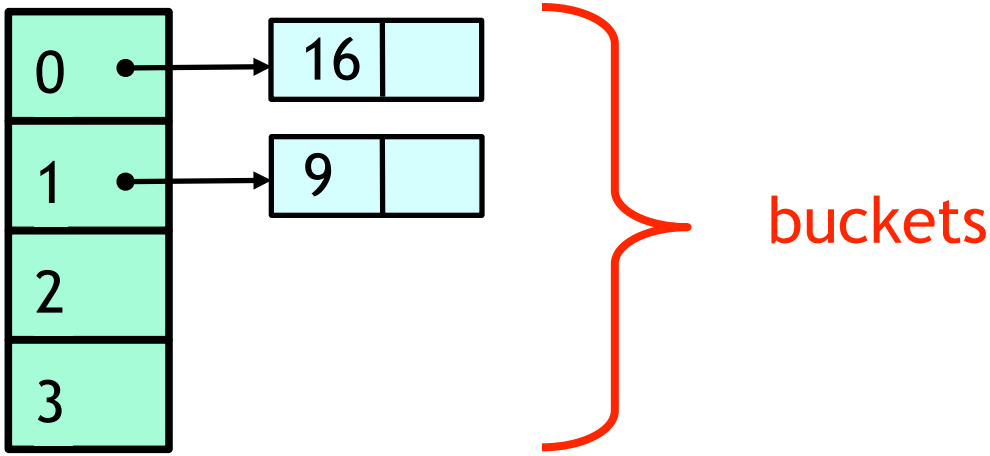
Linear-Time Set Methods

- 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

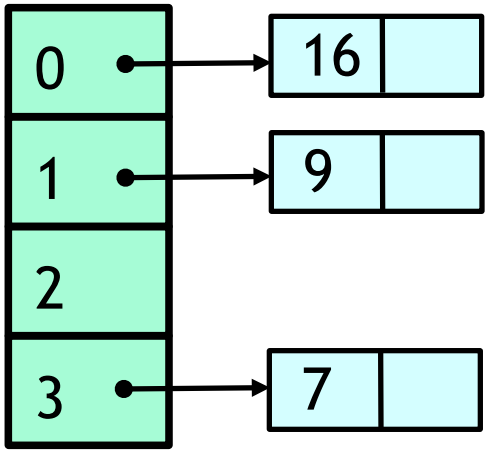
Sequential Hash Table



Item count: 2

$h(k) = k \bmod 4$

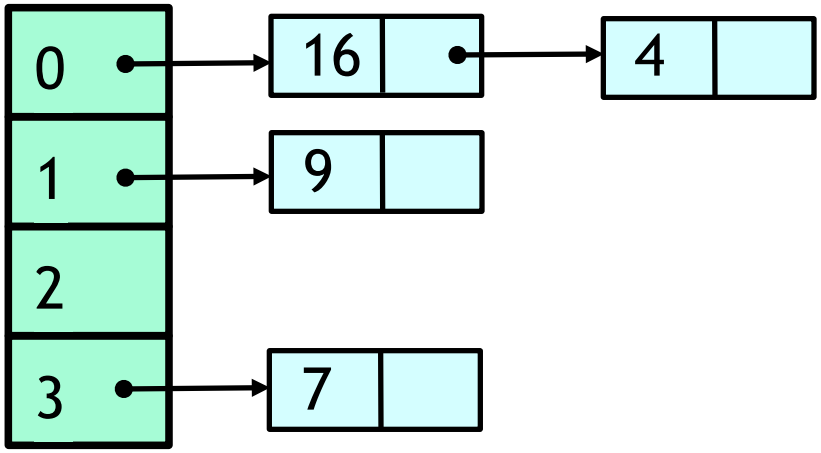
Sequential Hash Table



Item count: 3

$h(k) = k \bmod 4$

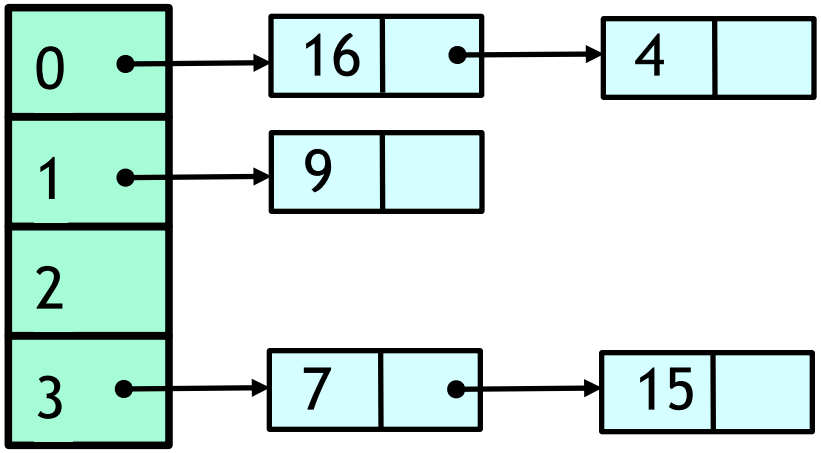
Sequential Hash Table



Item count: 4

$h(k) = k \bmod 4$

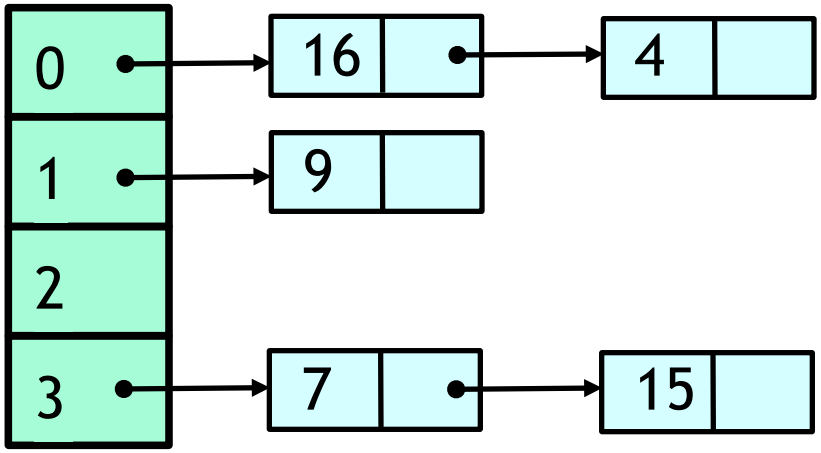
Sequential Hash Table



Item count: 5

$h(k) = k \bmod 4$

Sequential Hash Table

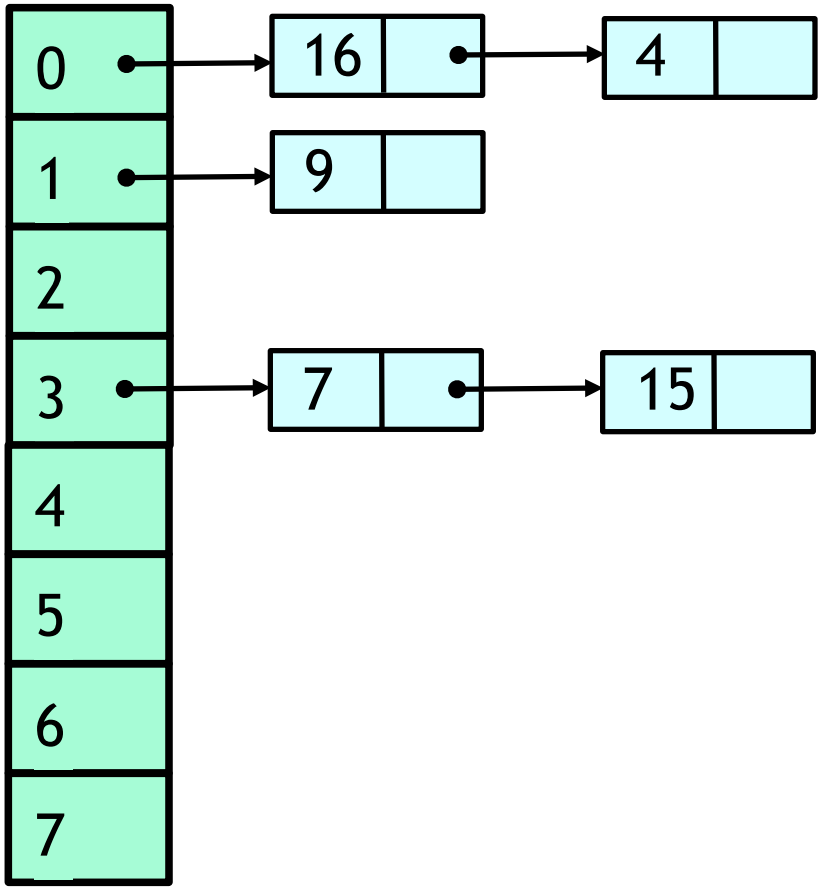


Problem:
buckets getting too long

Item count: 5

$h(k) = k \bmod 4$

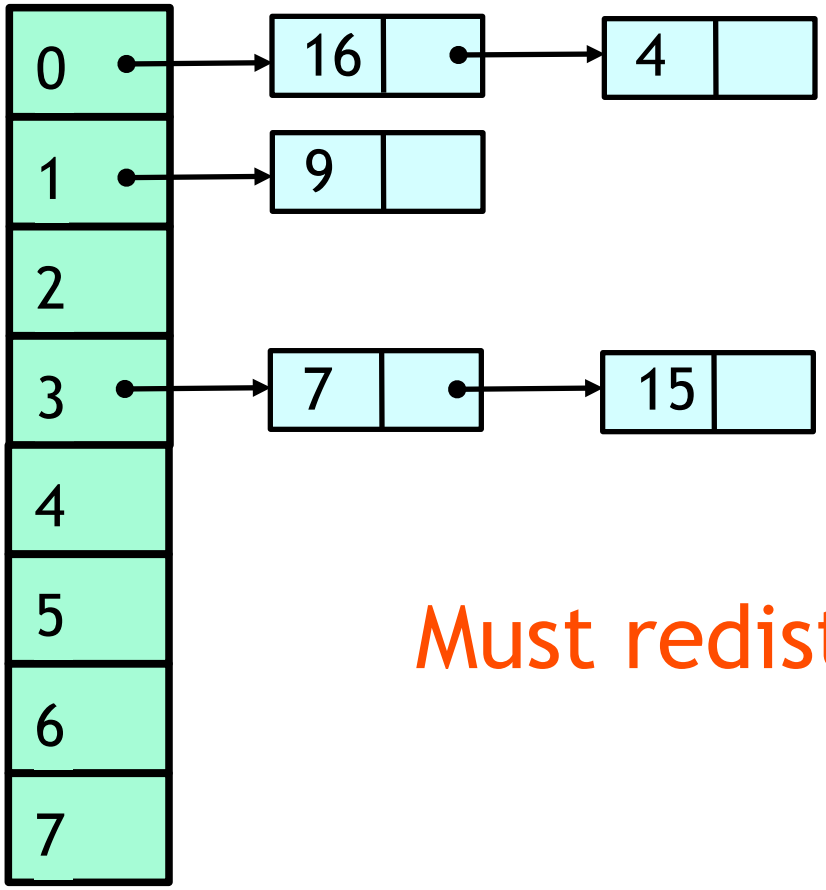
Resizing the table



Item count: 5

$h(k) = k \bmod 4$

Resizing the table

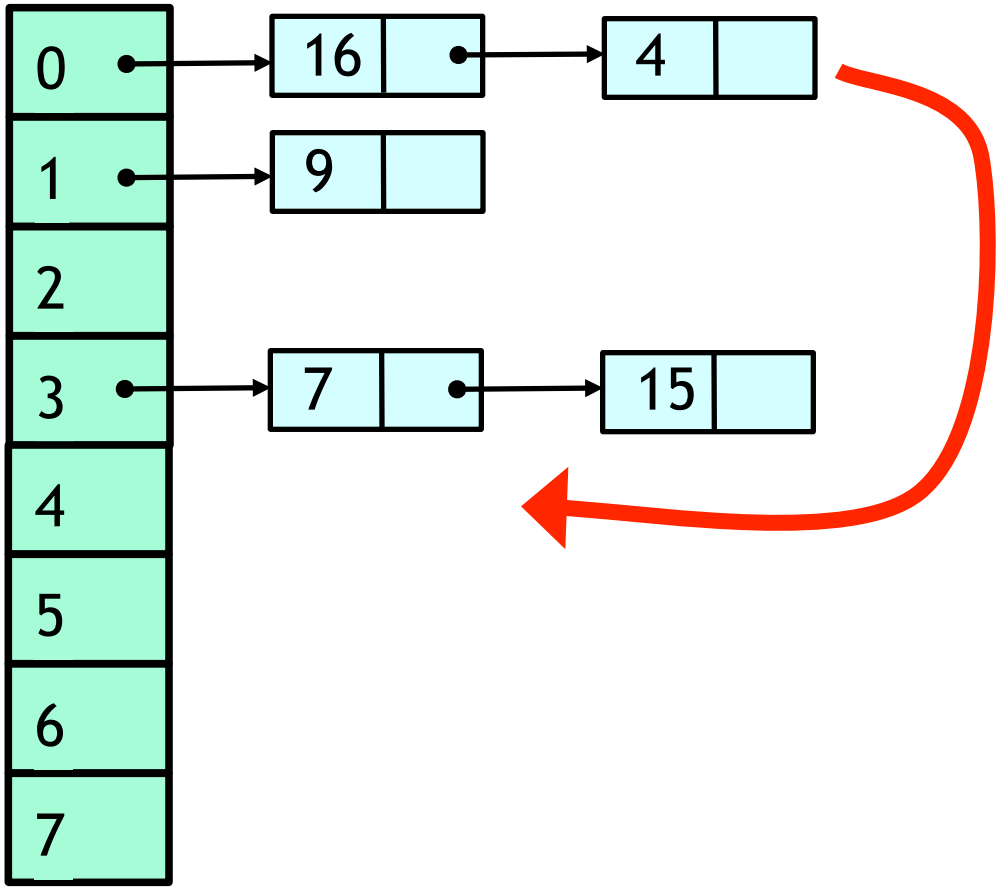


Item count: 5

$h(k) = k \bmod 8$

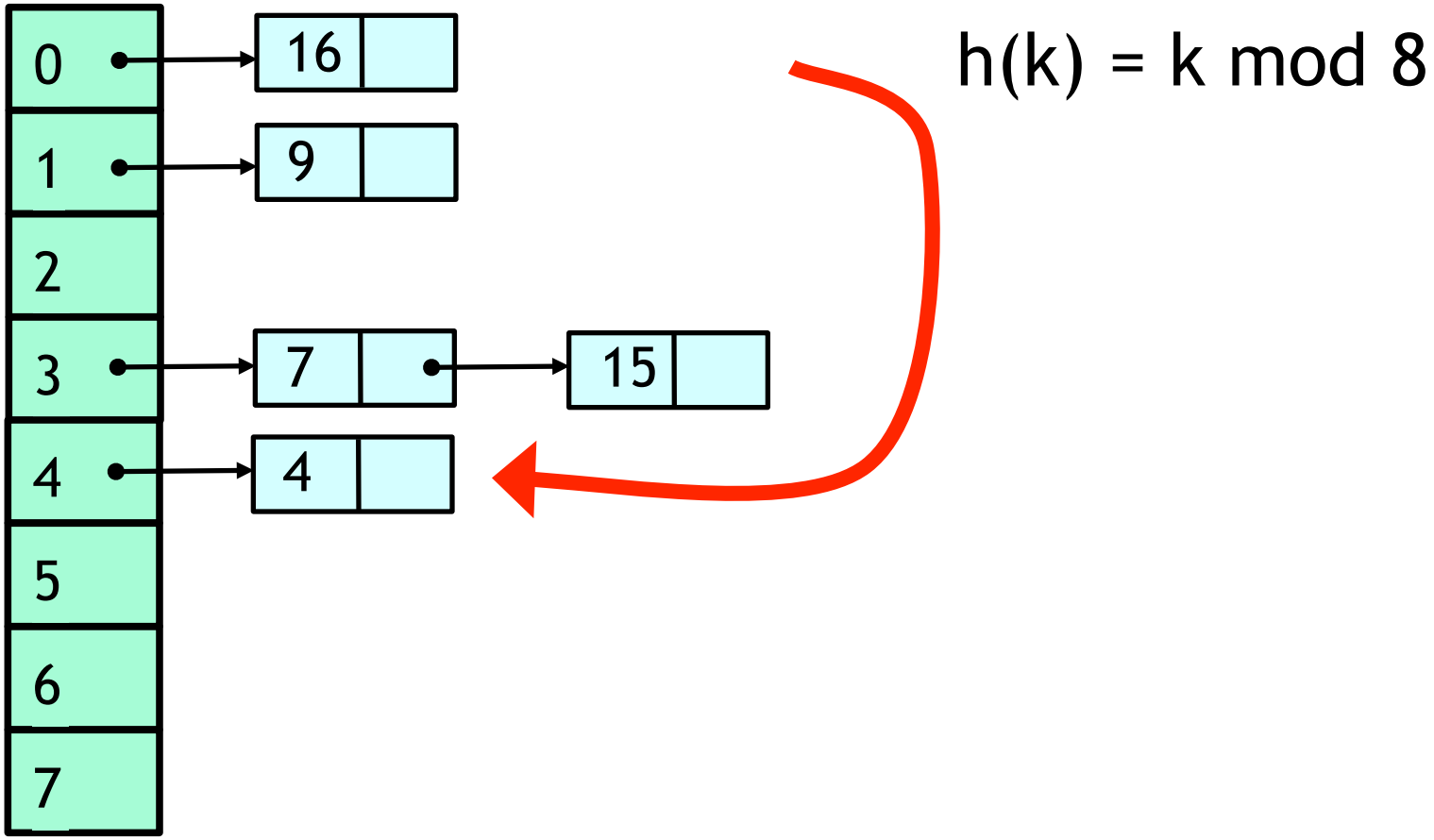
Must redistribute items

Moving the items

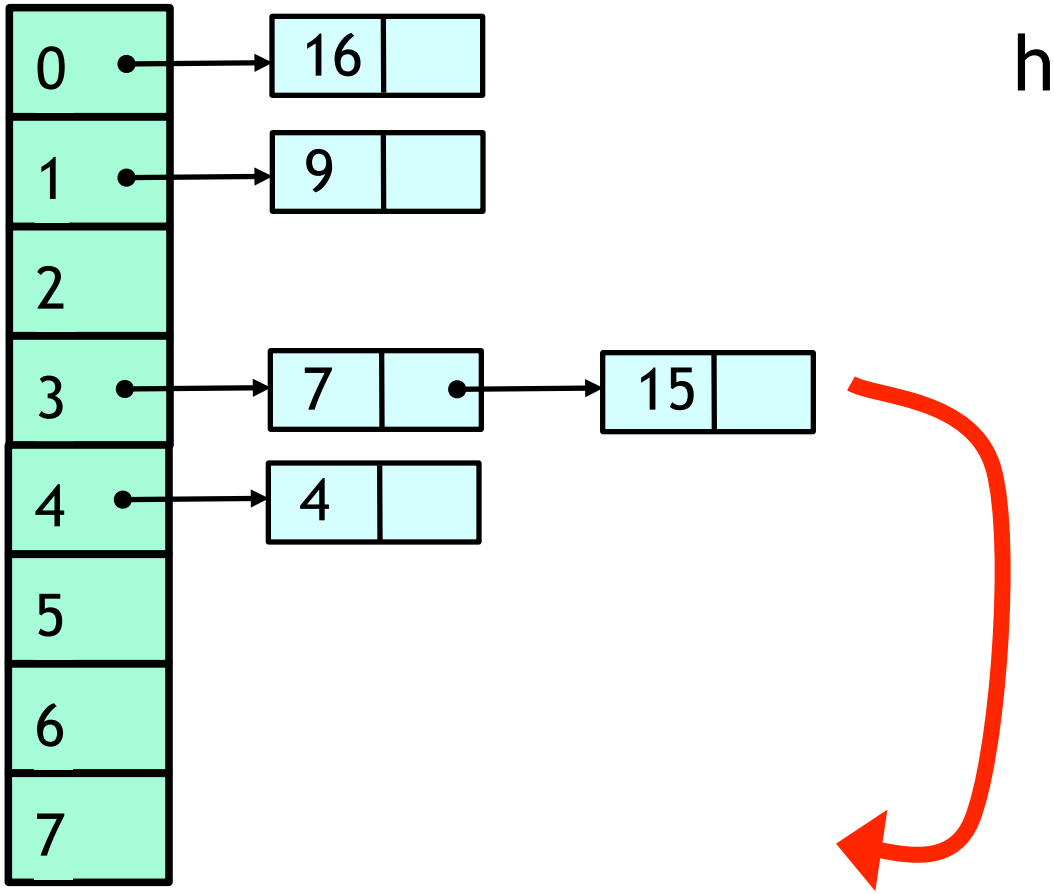


h(k) = k mod 8

Moving the items

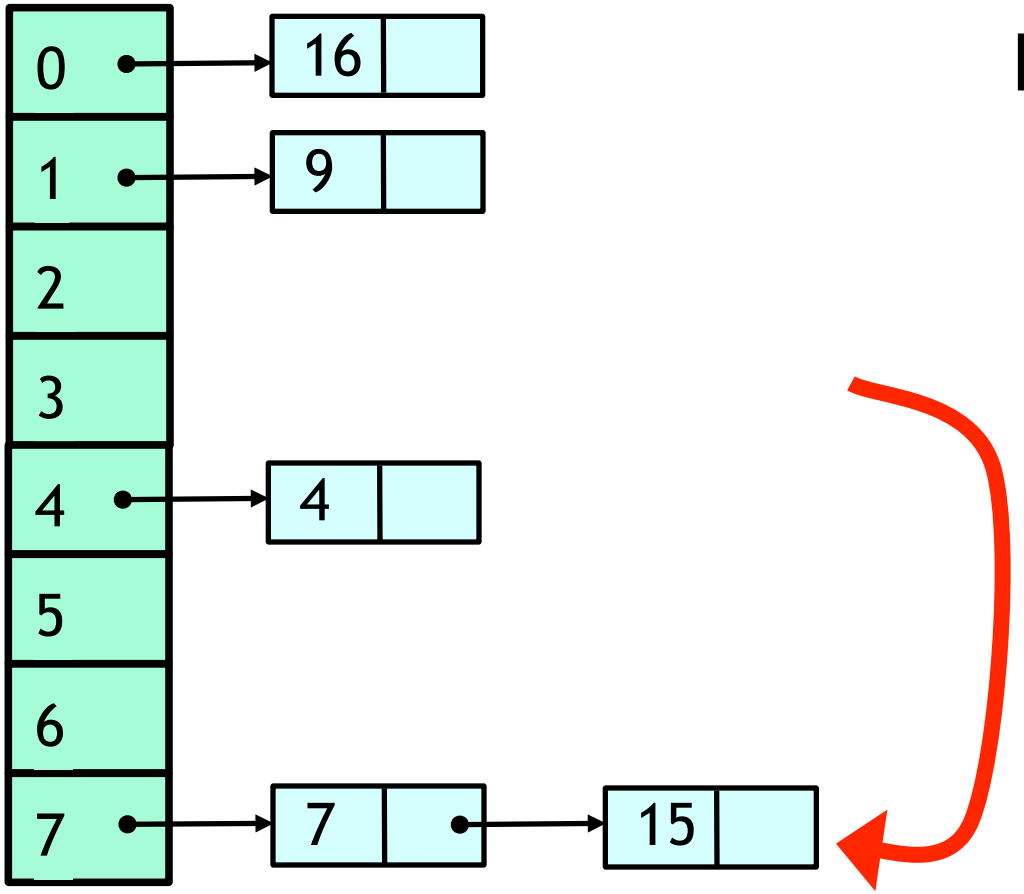


Moving the items



h(k) = k mod 8

Moving the items



h(k) = k mod 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-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 important

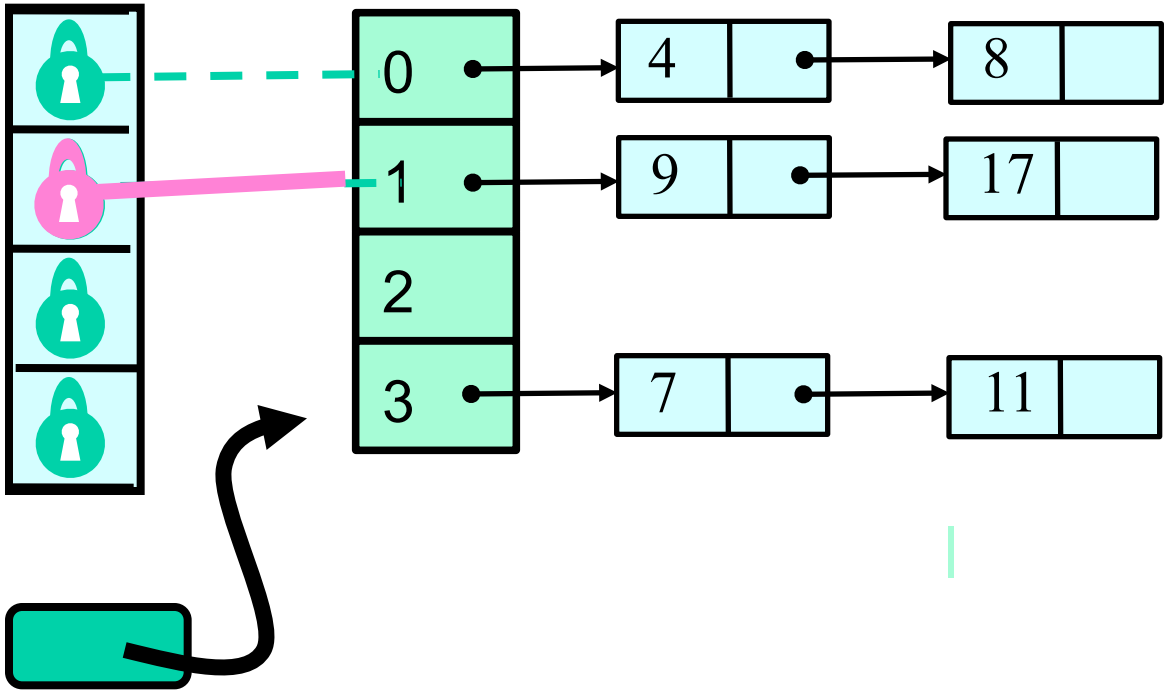
When to Resize?

- Many reasonable policies. Here's one.
- Bucket threshold
  - When ≥ ¼ buckets exceed this value
- Global threshold
  - When any bucket exceeds this value

Coarse-Grained Locking

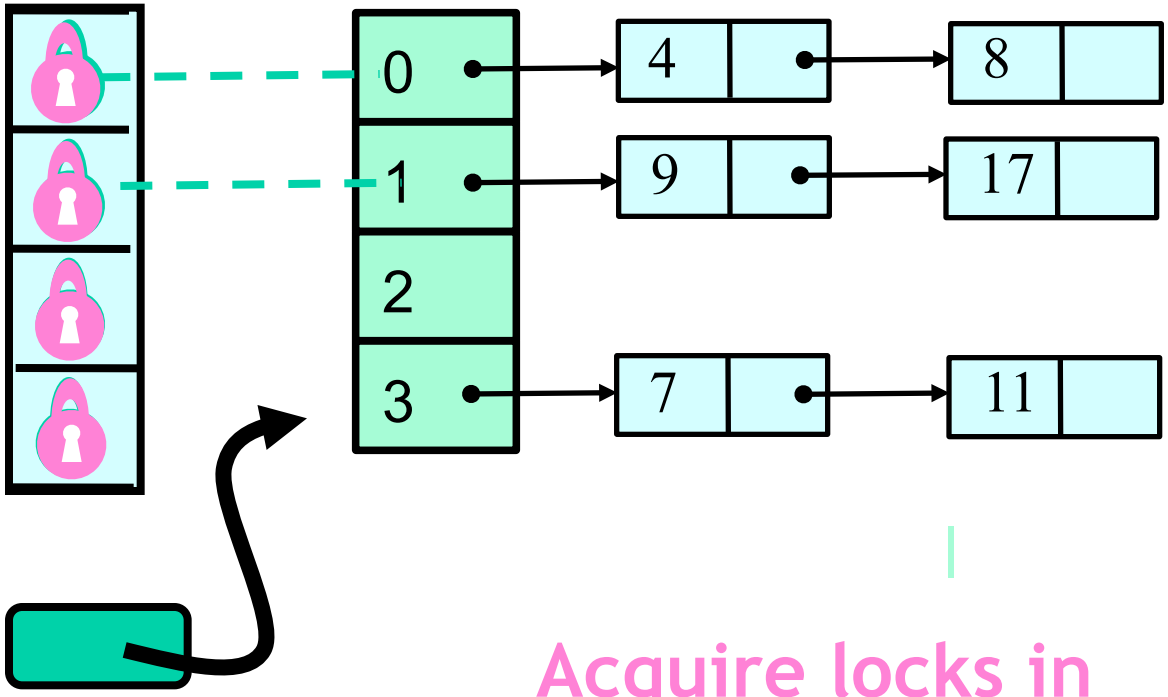
- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck

Fine-grained Locking



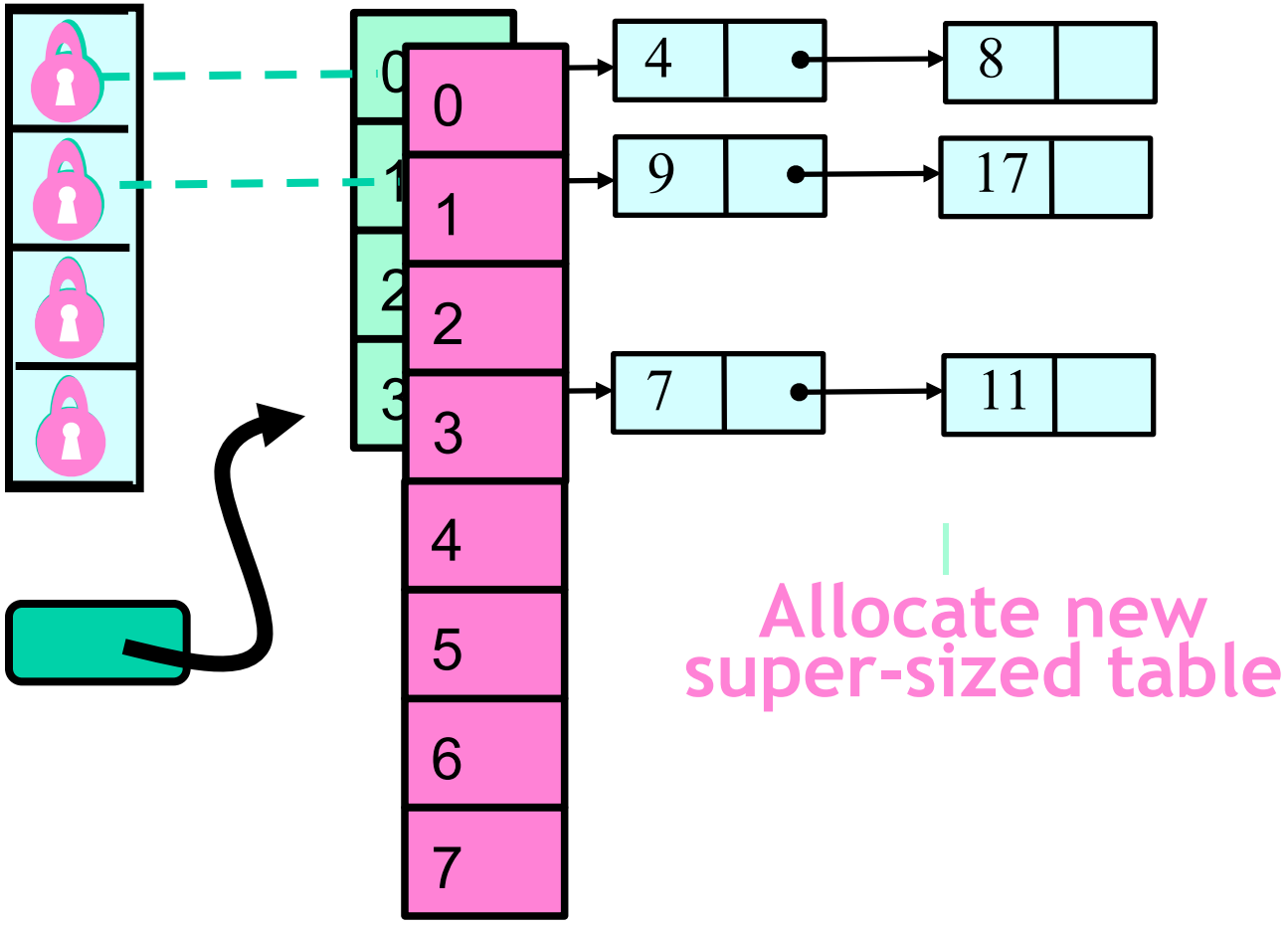
Each lock associated with one bucket

Resizing

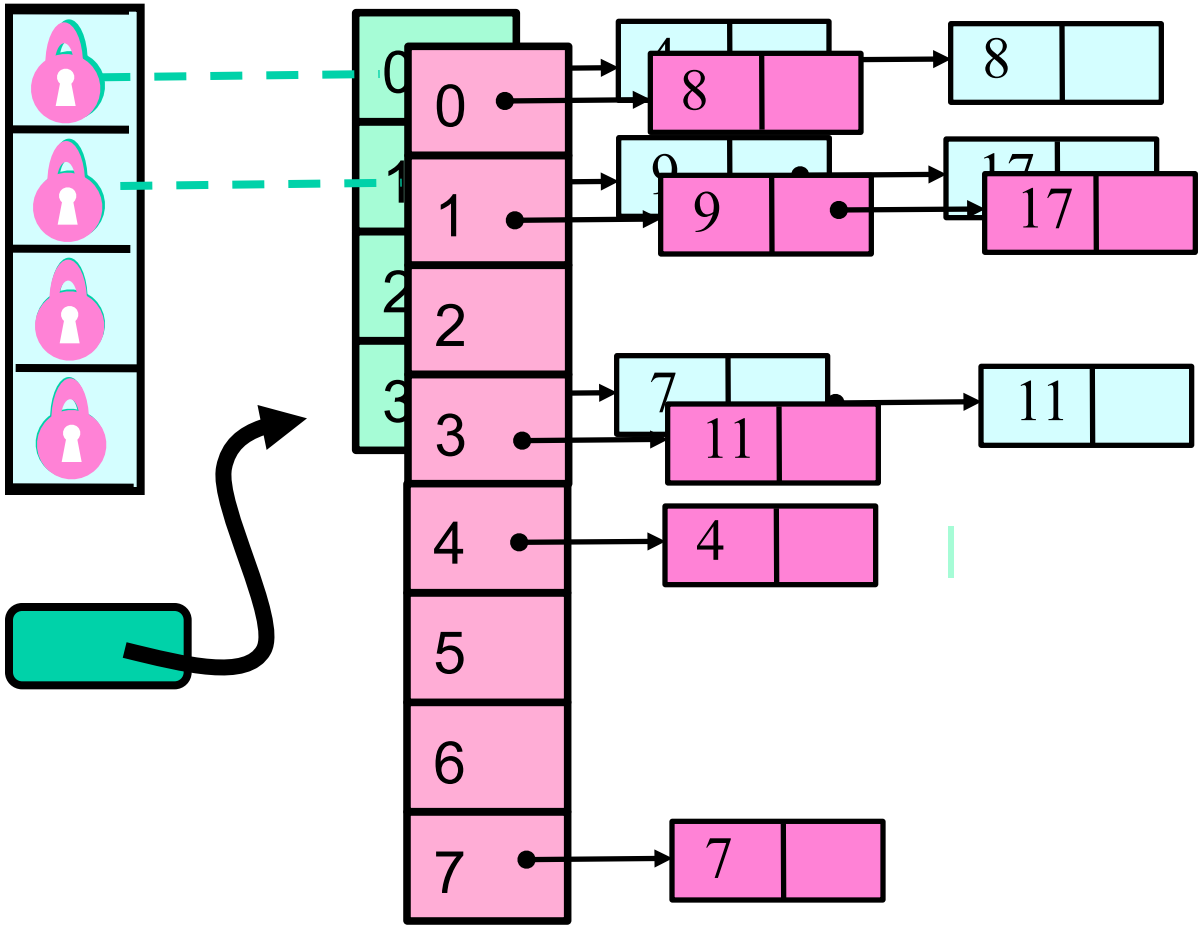


Acquire locks in ascending order

Fine-grained Locking

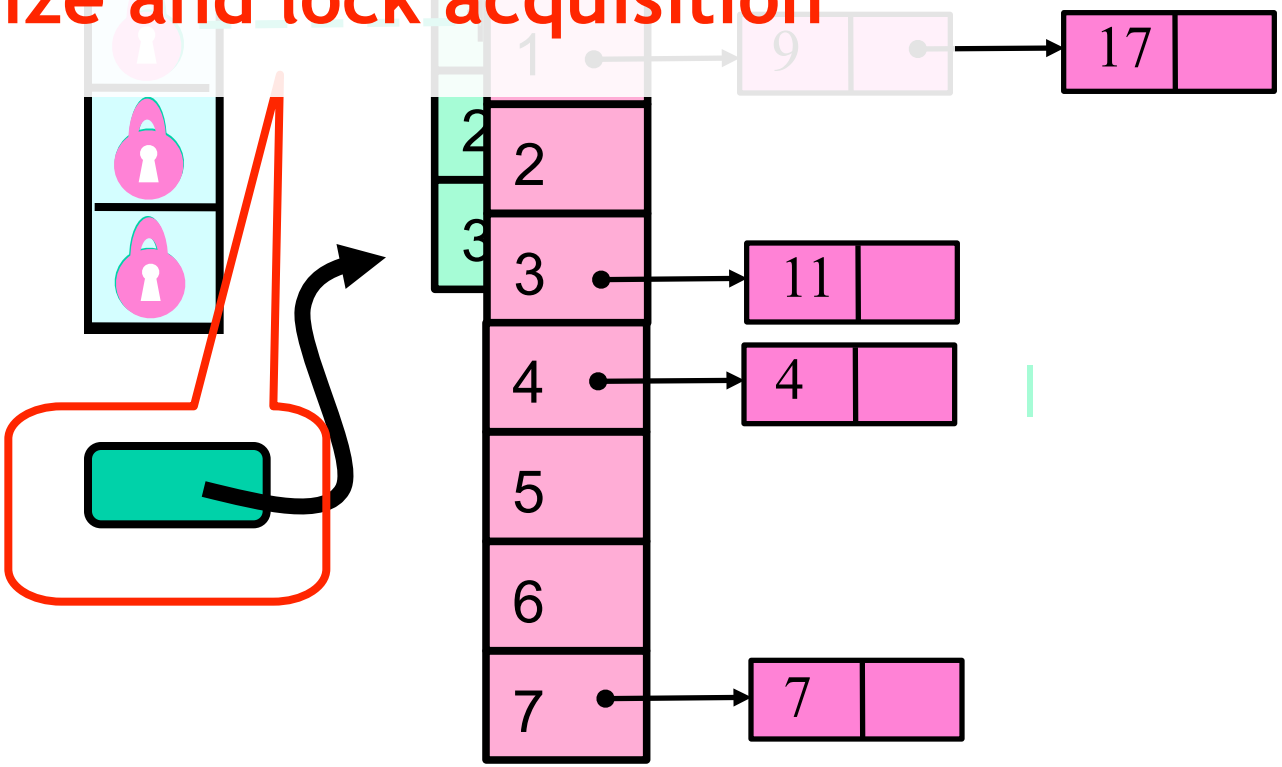


Resizing



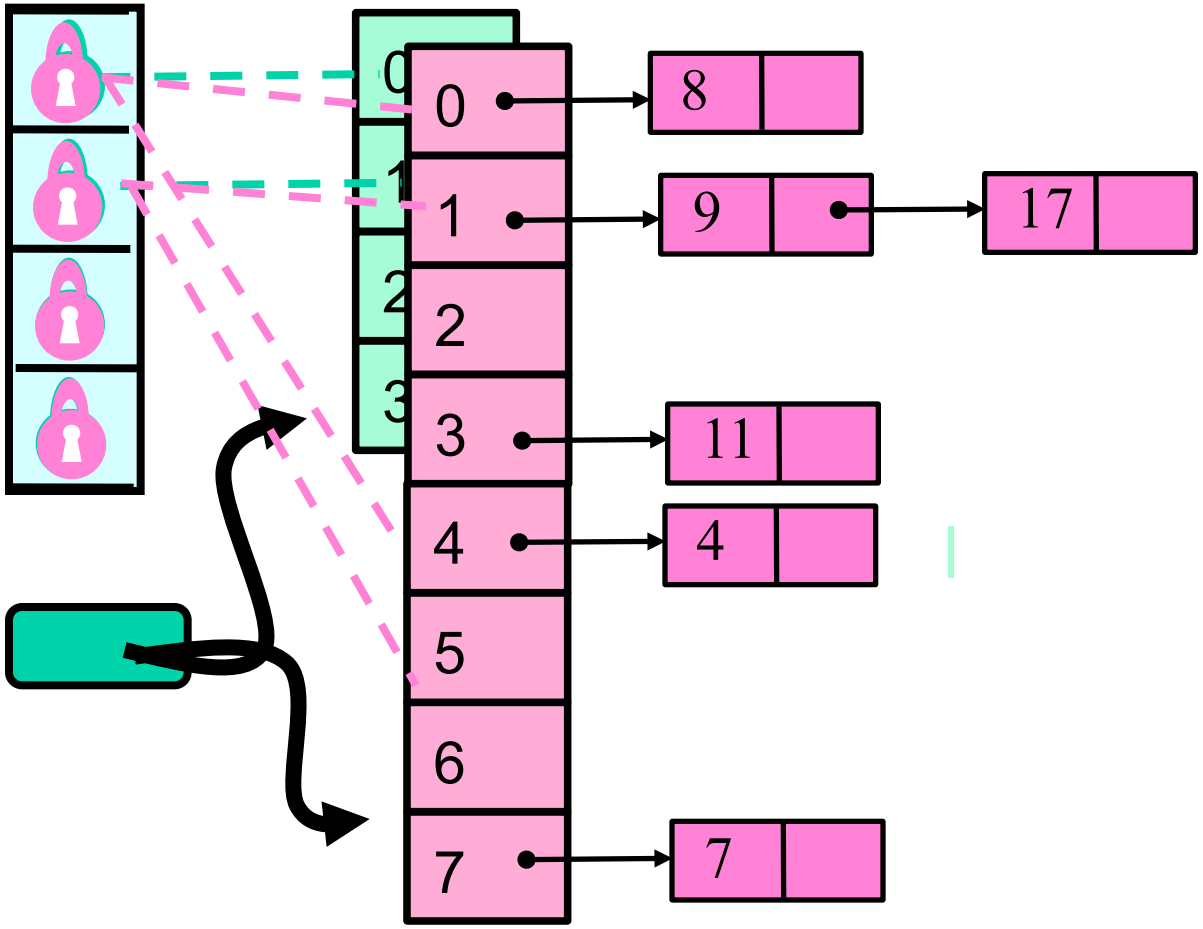
Resizing

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



Striped Locks: each lock associated with two buckets

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

Initially same number of locks and buckets

```
public class FGH
 protected Range
 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 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);
  }
 }
```

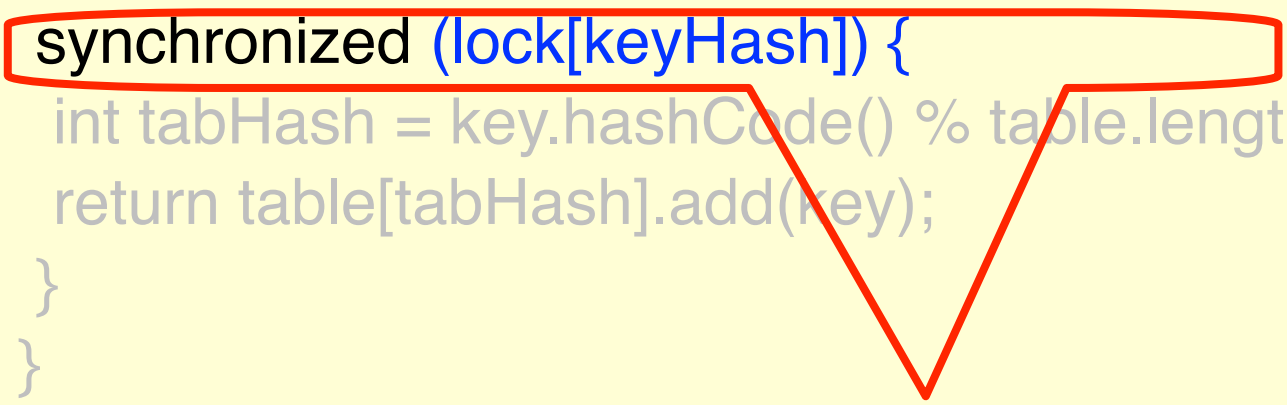
Fine-Grained Locking

```
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?

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);
 }
}
```



Acquire 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);
 }
}
```

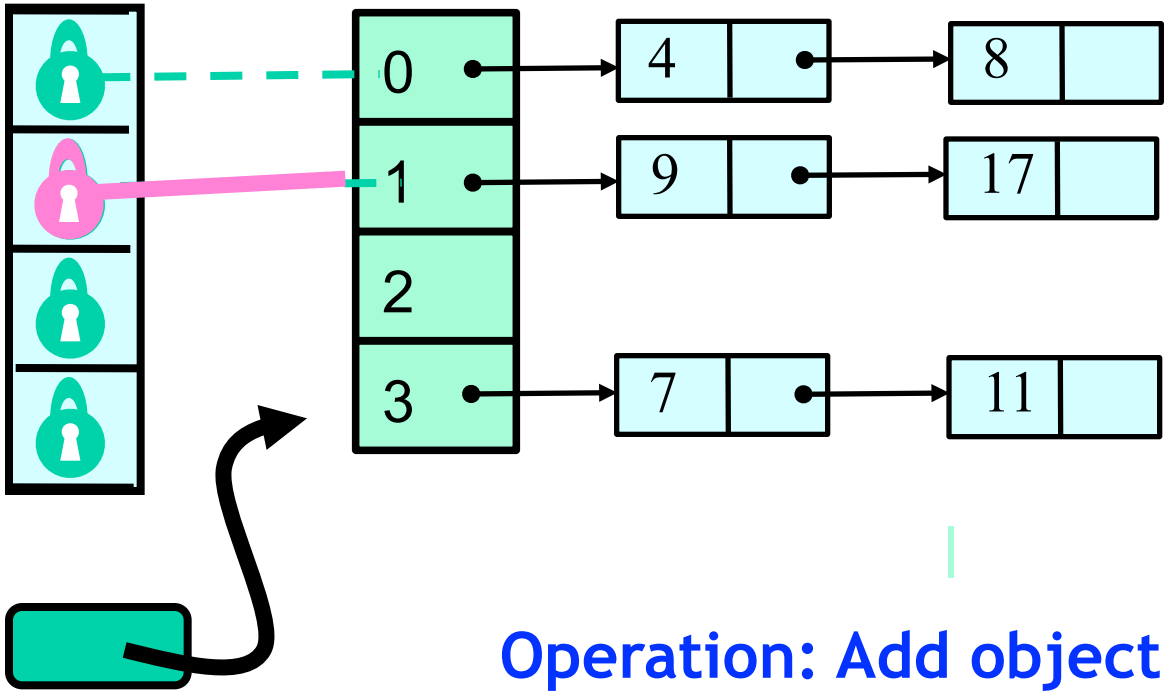
Which bucket?

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);
 }
}
```

Call bucket's add() method

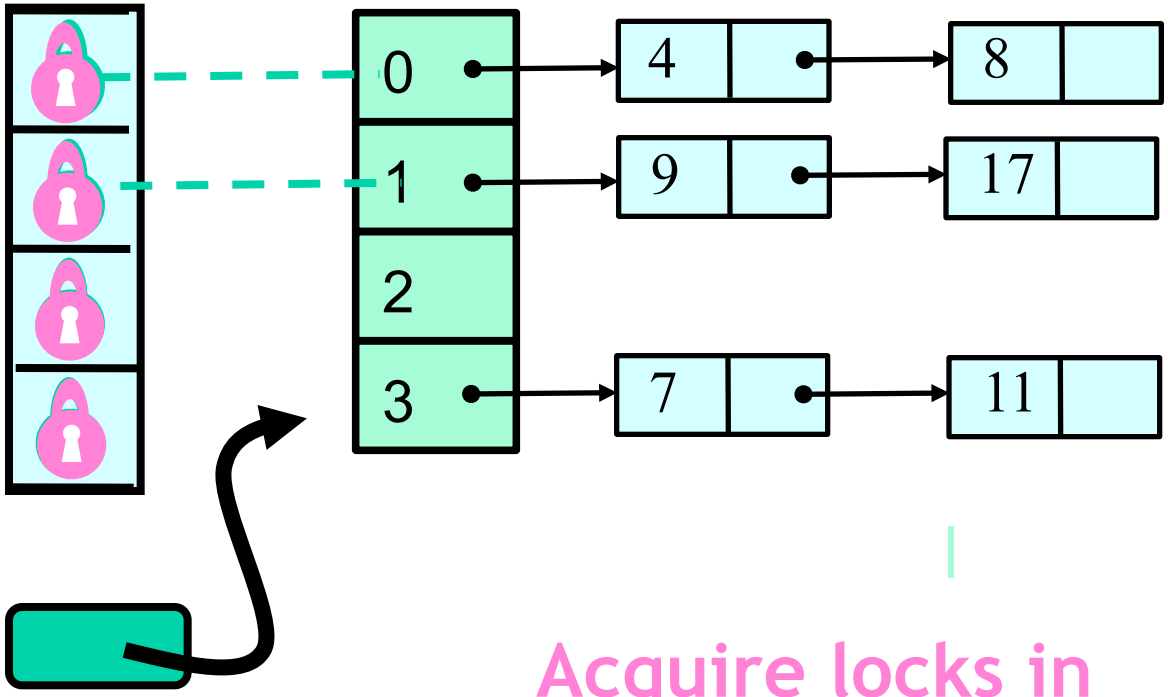
Fine-grained Locking



Operation: Add object 17

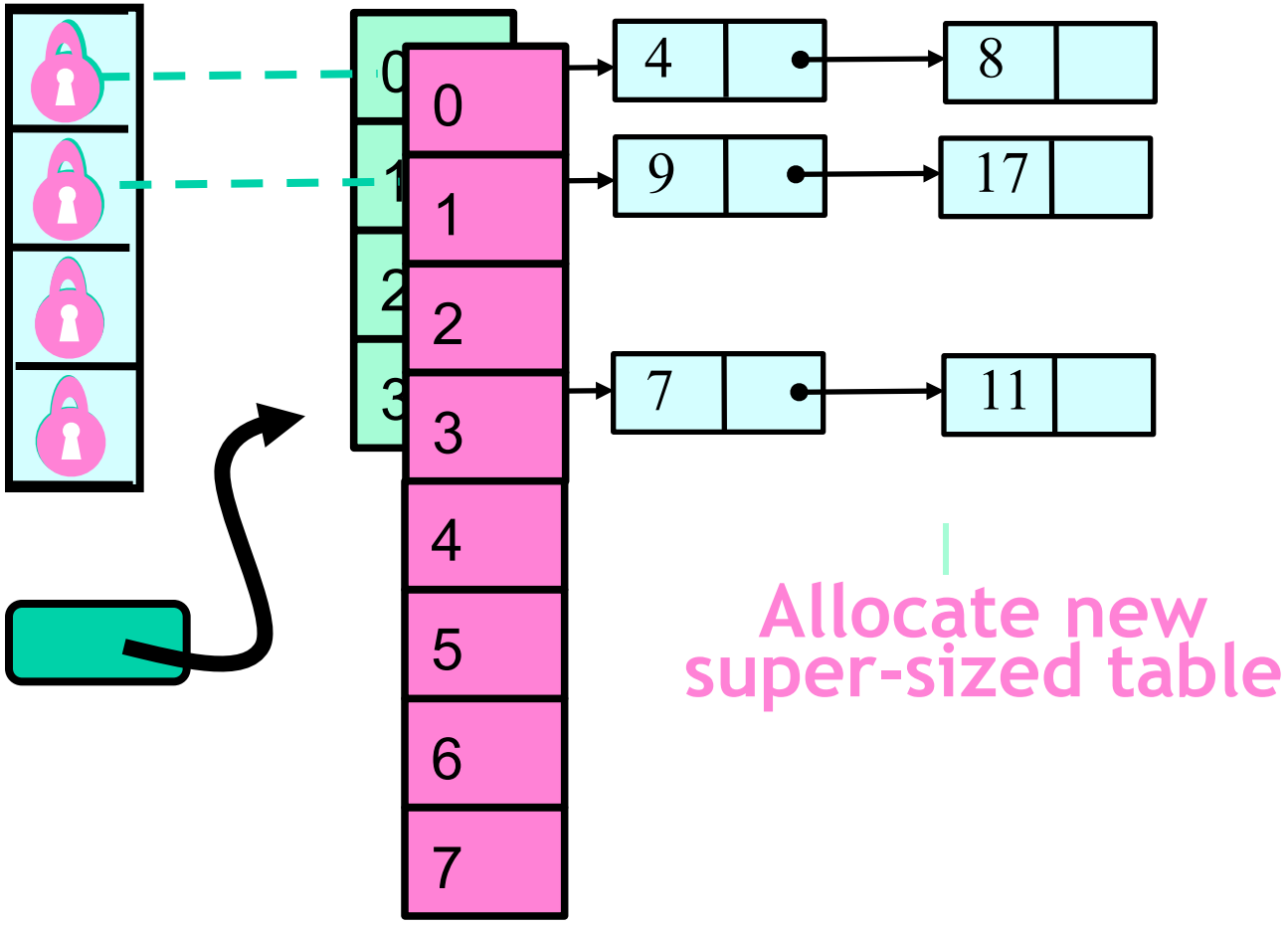
Each lock associated with one bucket

Resizing

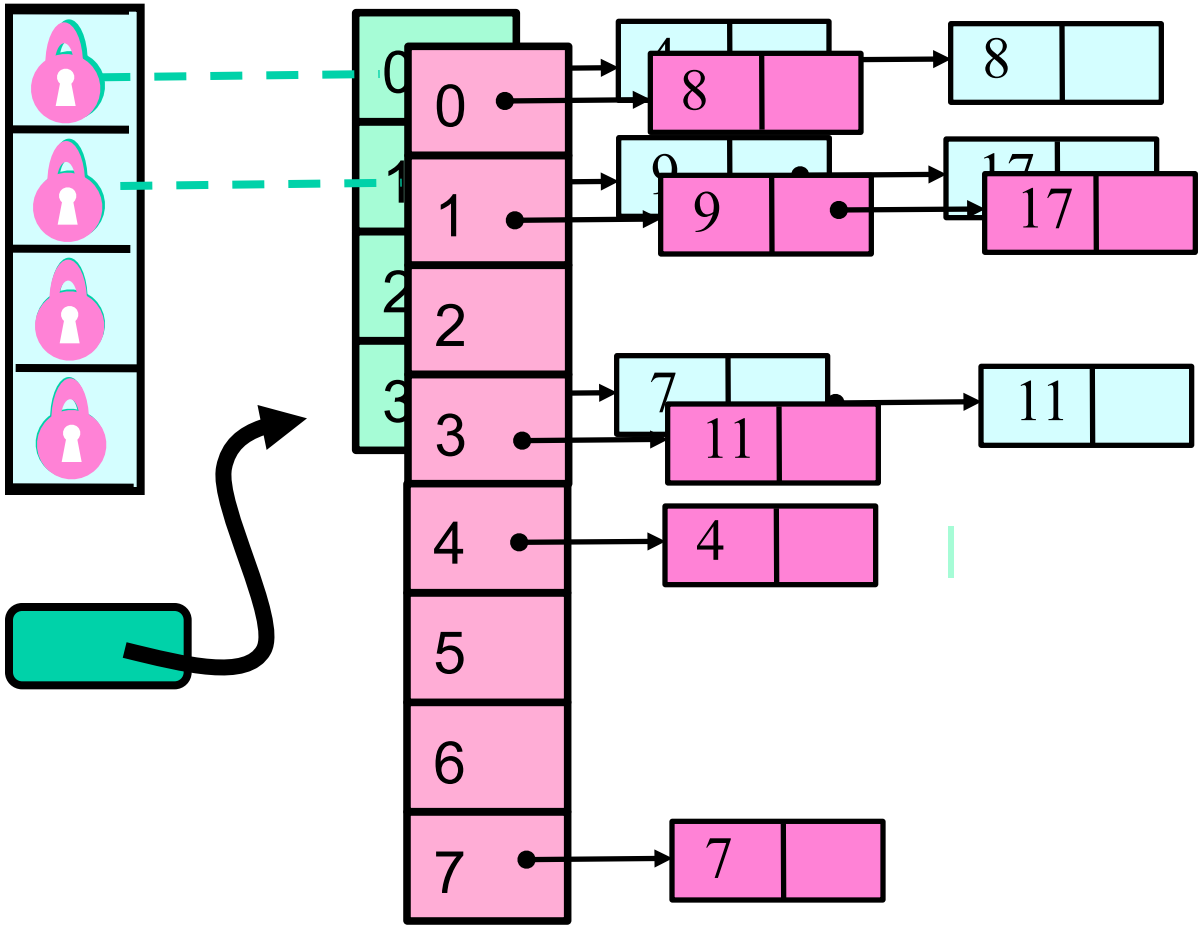


Acquire locks in ascending order

Fine-grained Locking

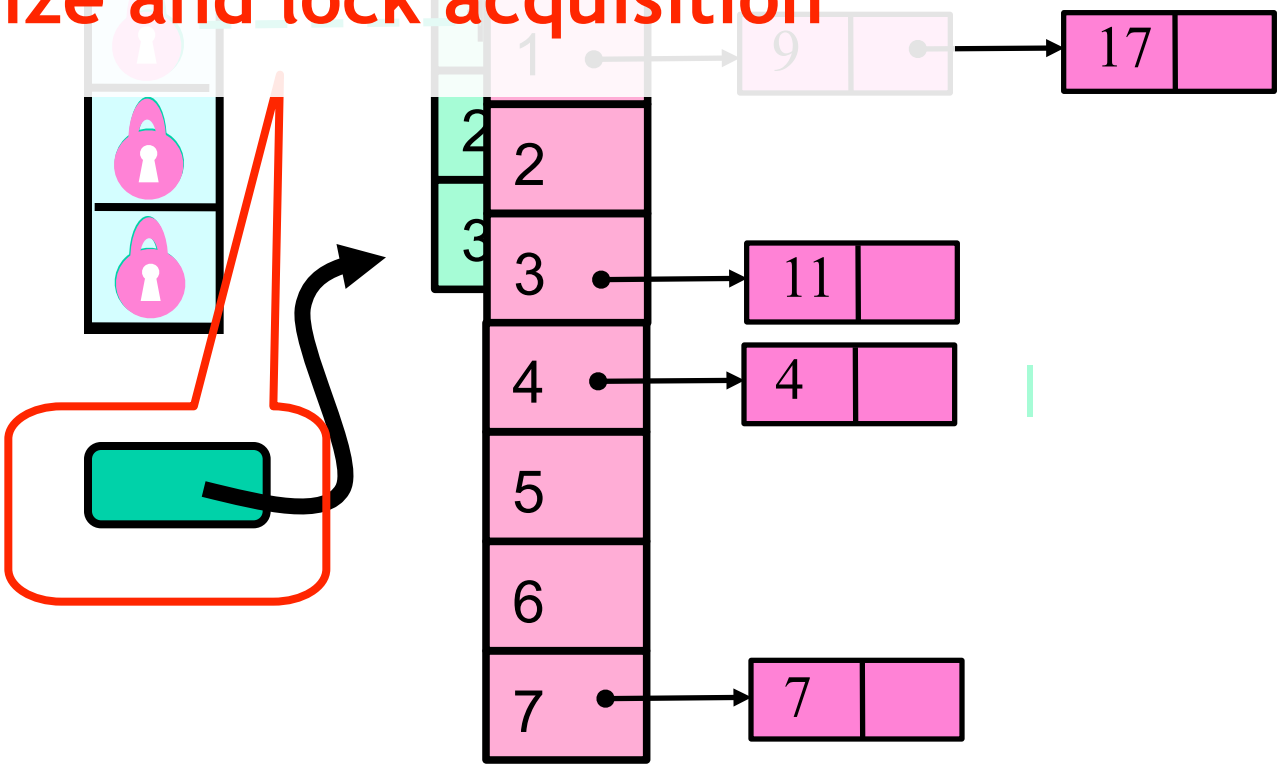


Resizing



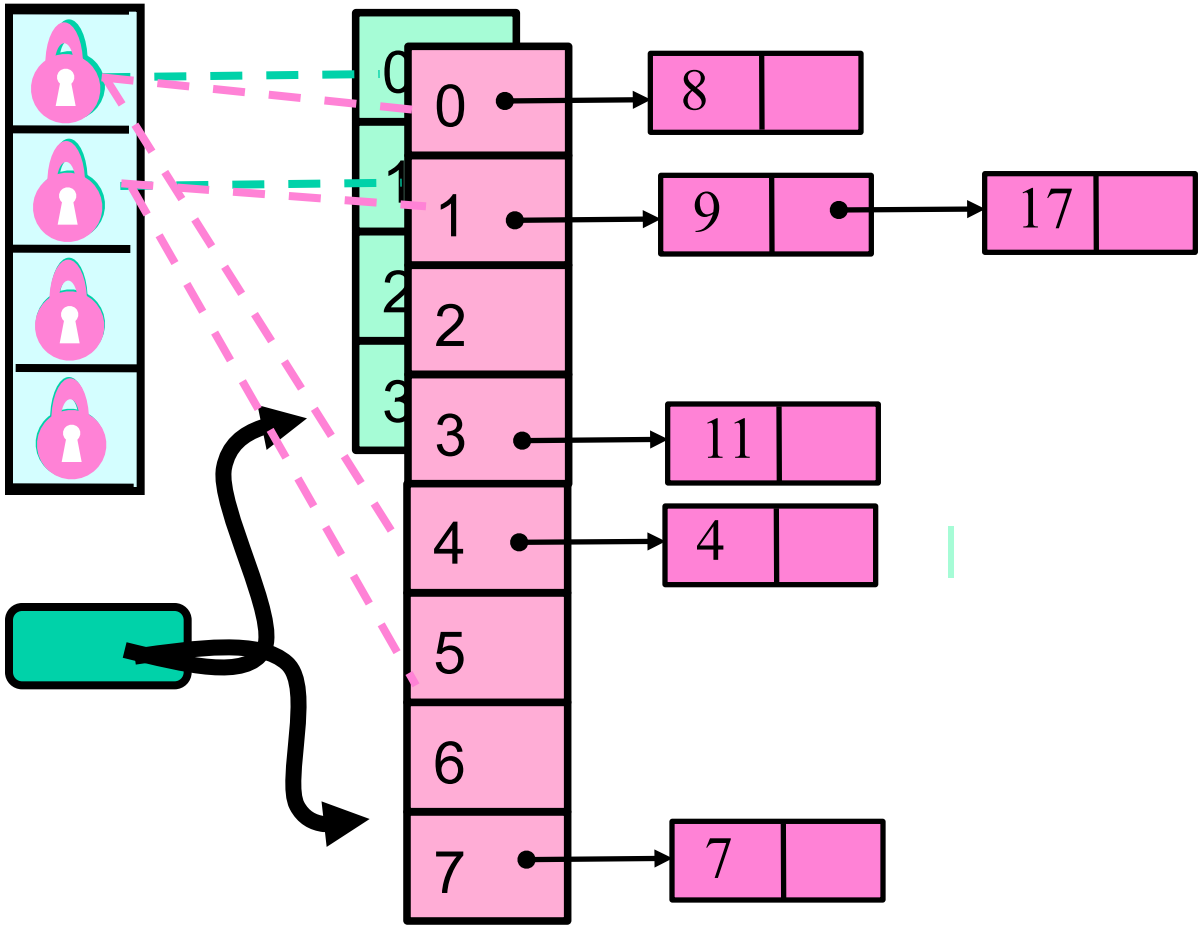
Resizing

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



Striped Locks: each lock associated with two buckets

Resizing



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();
 }}}
```

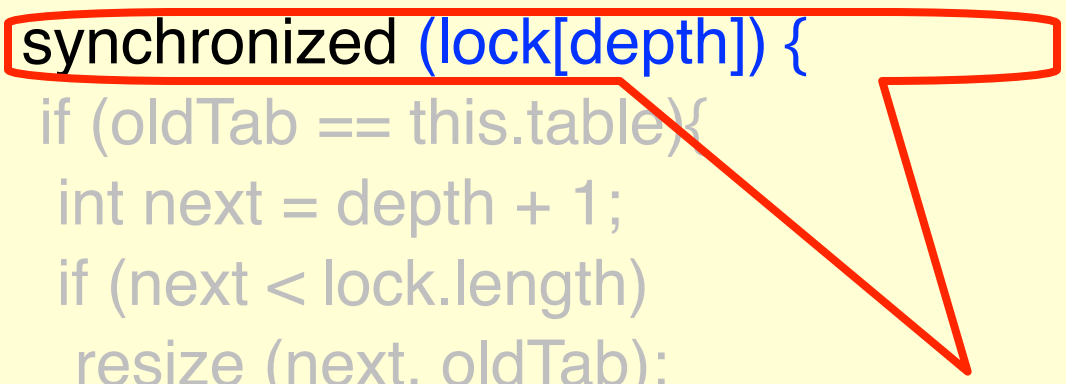
Fine-Grained Locking

```
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
    sequentialResize();
}}}
```

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

```
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
    sequentialResize();
 }}}
```

Locks acquired, do the work

Fine-Grained Locks

- We can resize the table
- But not the locks
- Debatable whether method calls are constant-time in presence of contention:
  - #locks should grow proportional to the number of concurrent operations

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
  - Striped locks cover a range (not resized)
- Read/Write locks
  - FIFO property enforces fairness

Concurrent Hashing Part 2

Christof Fetzer, TU Dresden

Based on slides by Maurice Herlihy and Nir Shavit

The Story So Far

- Hash table = list of lists
- Book-shelf:
  - Which Row? — hash of the key
  - Within a row: list of objects (books)
  - Why? — Faster access than one large list
  - Goal: Constant access time + concurrency

The Story So Far

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

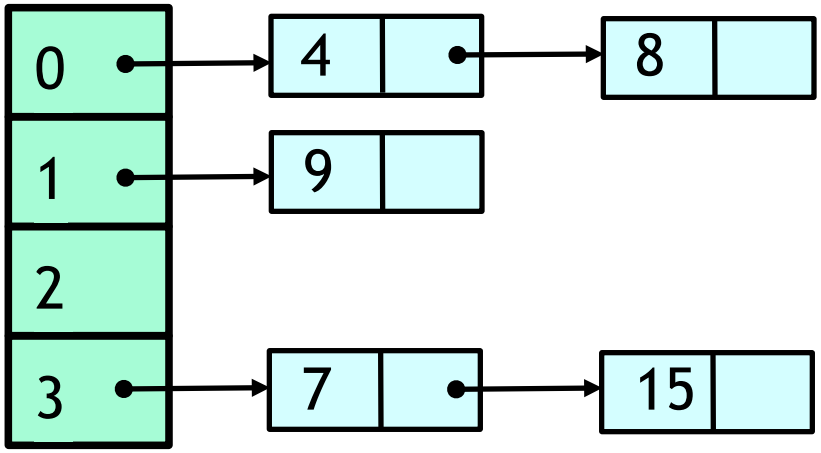
Optimistic Synchronization

- 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

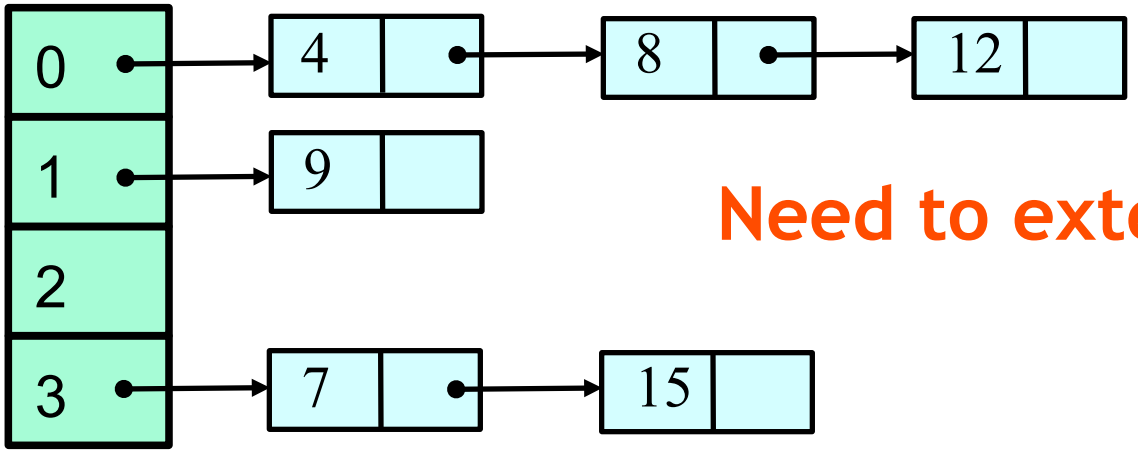
- If it doesn't find the key
  - May be victim of resizing
- Must try again
  - Getting a read lock this time
- Makes sense if
  - Resizes are rare
  - Keys are present

Lock-Free Resizing Problem



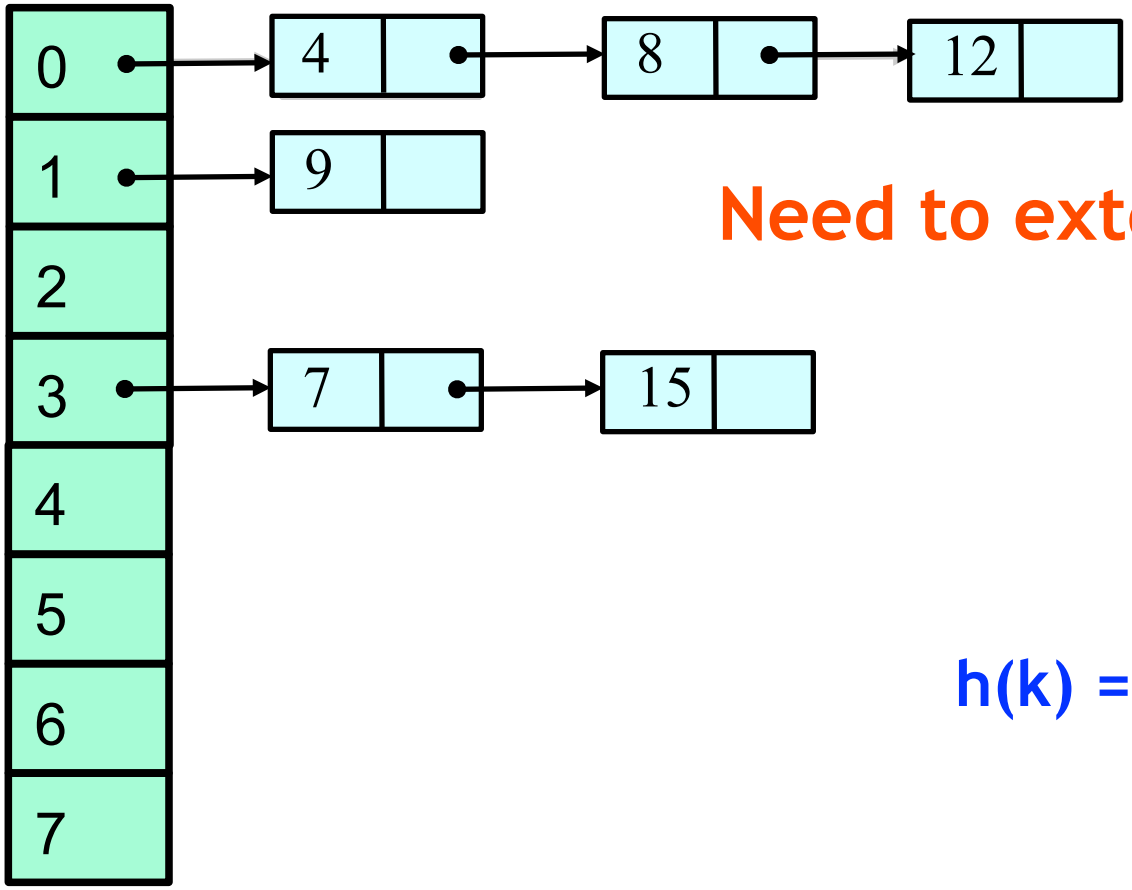
h(k) = k mod 4

Lock-Free Resizing Problem



Need to extend table

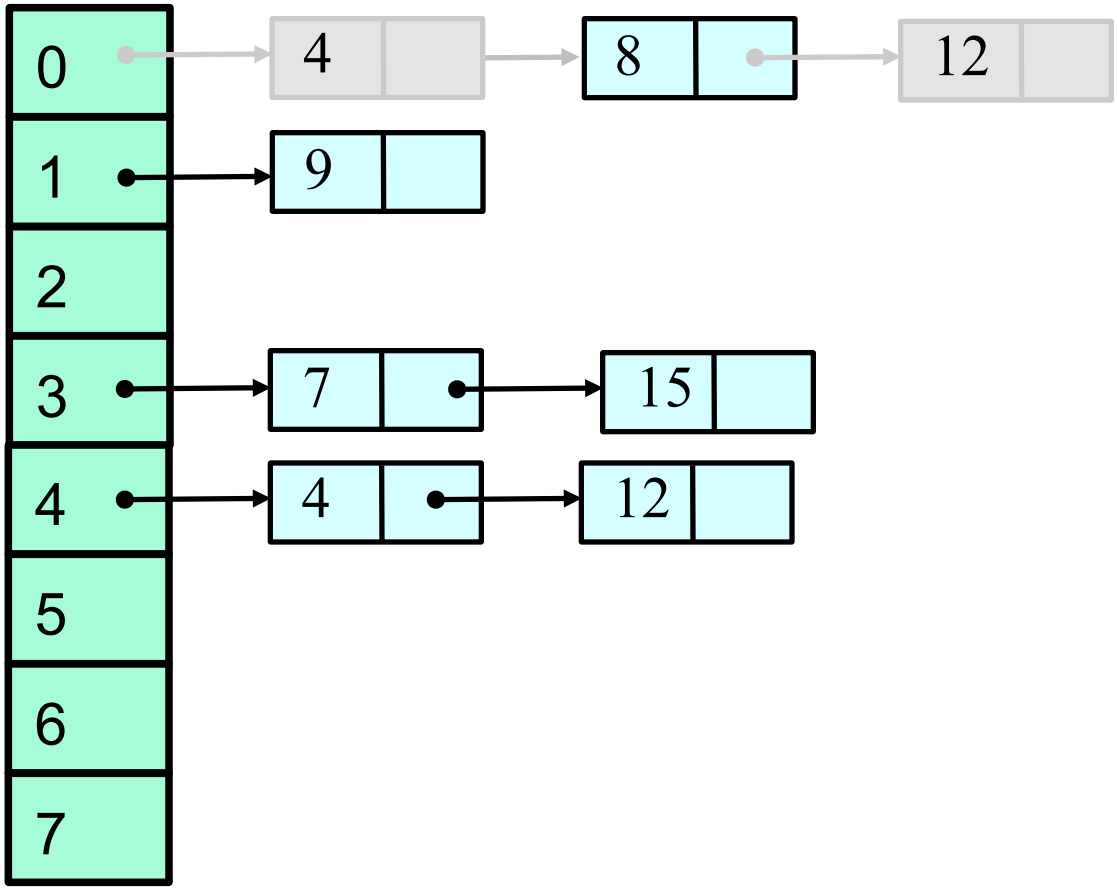
Lock-Free Resizing Problem



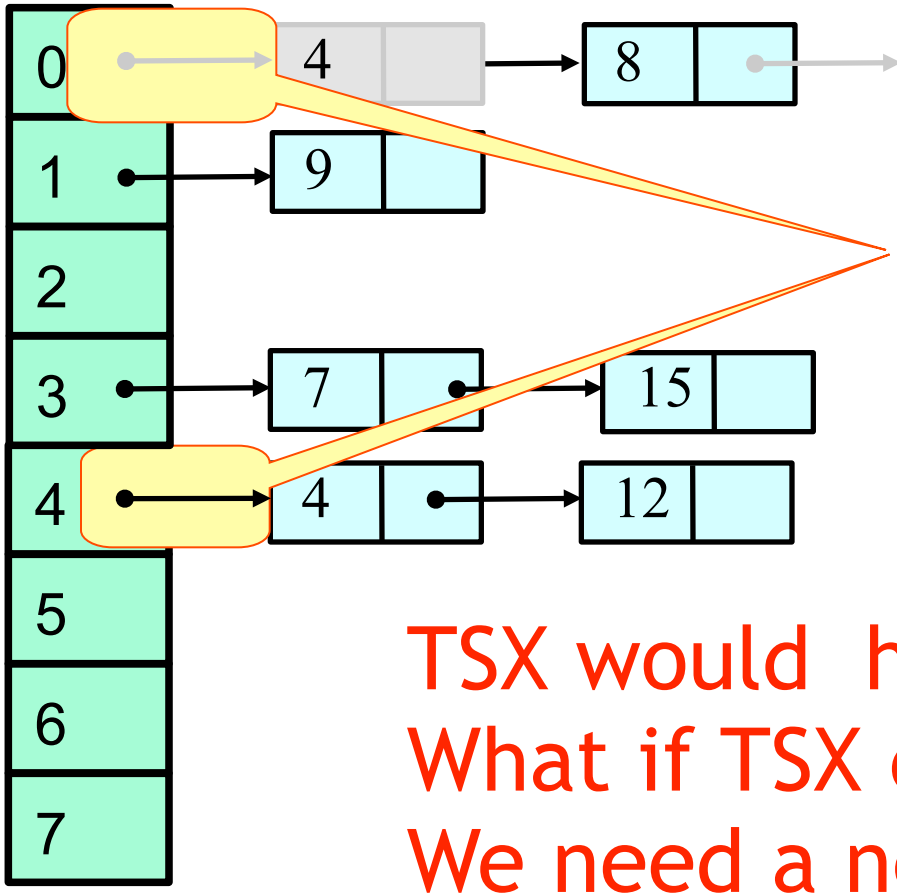
Need to extend table

h(k) = k mod 8

Lock-Free Resizing Problem



Lock-Free Resizing Problem

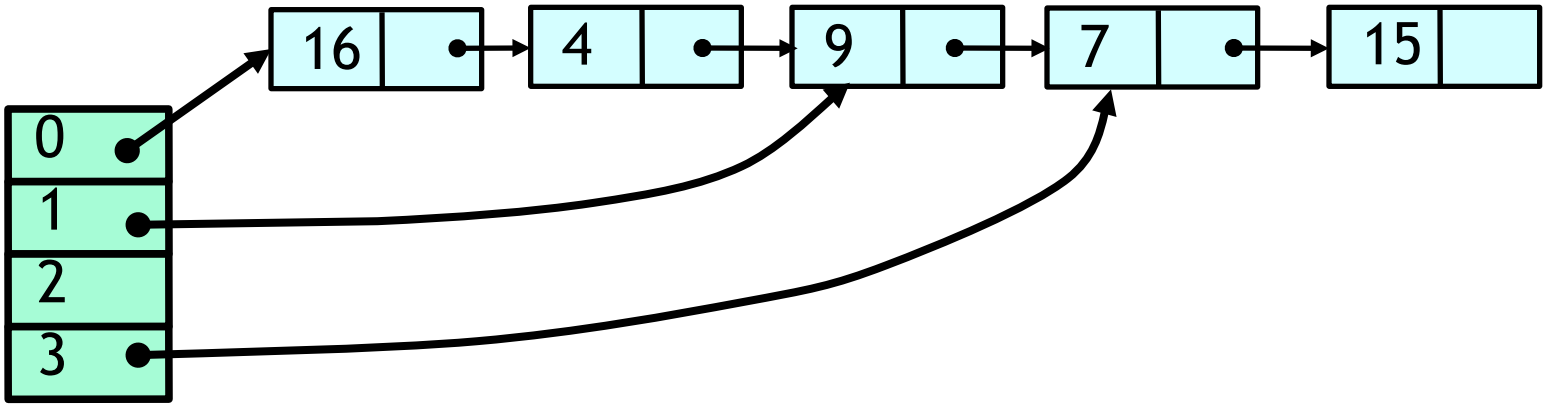


one CAS not enough to remove and then add a single item

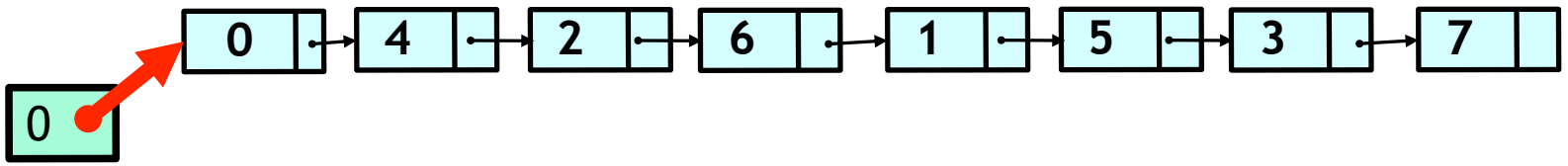
TSX would help.
What if TSX does not work?
We need a new idea…

Don’t move the items

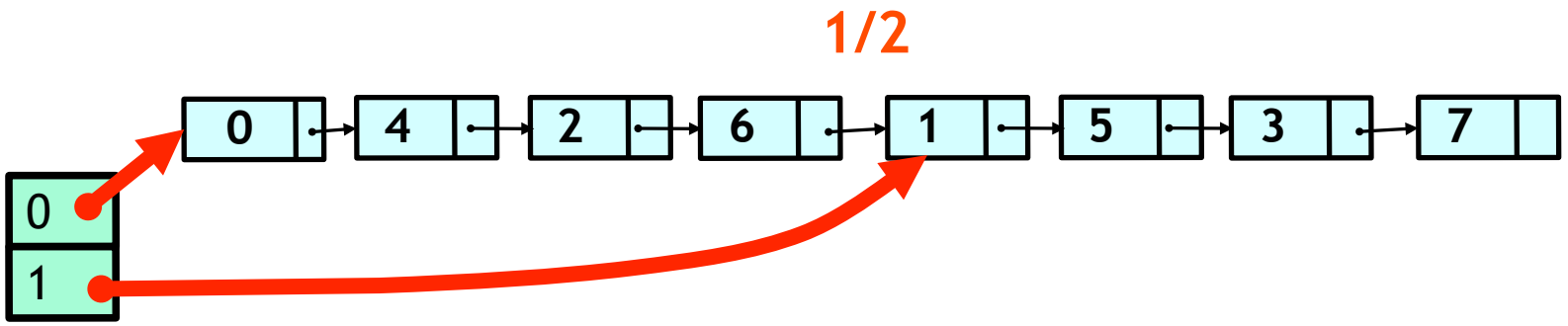
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become “shortcut pointers” into the list



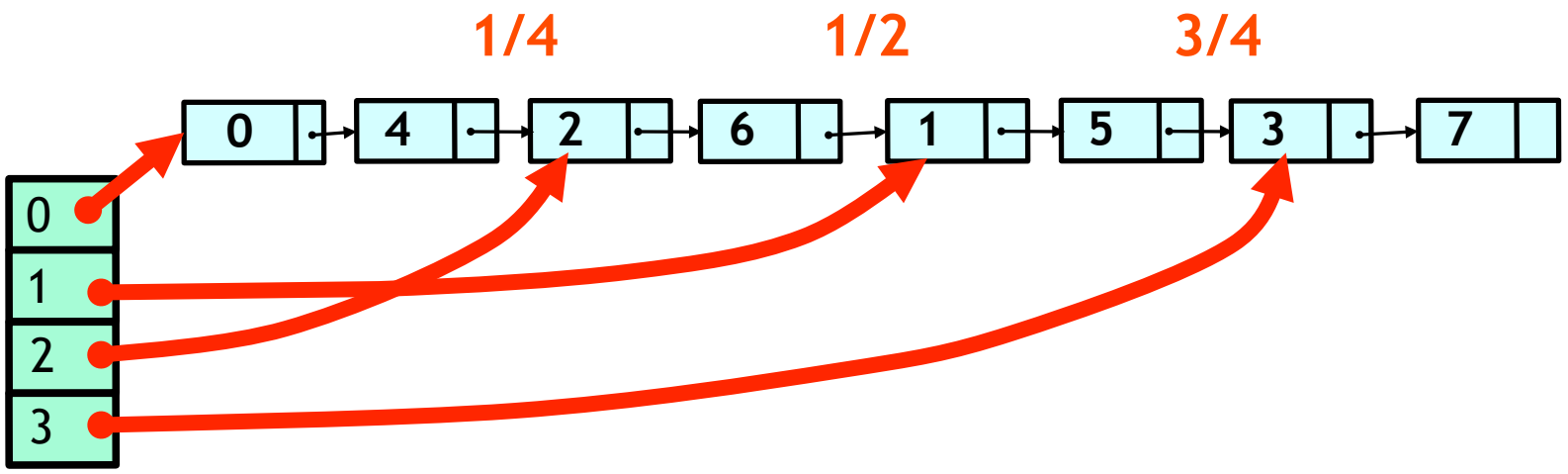
Recursive Split Ordering



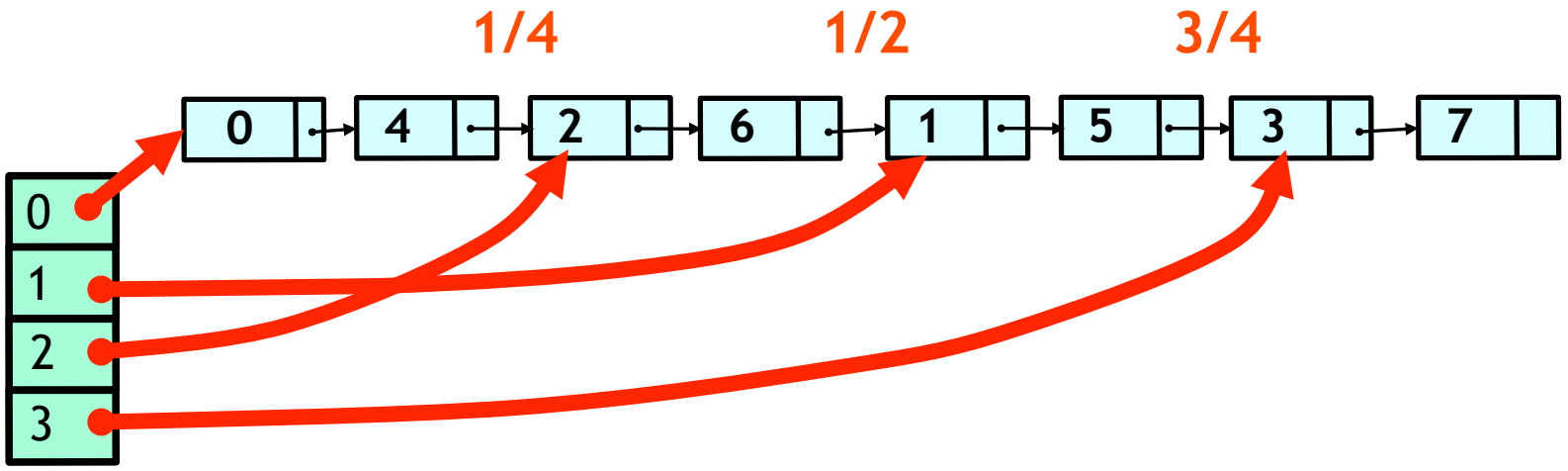
Recursive Split Ordering



Recursive Split Ordering

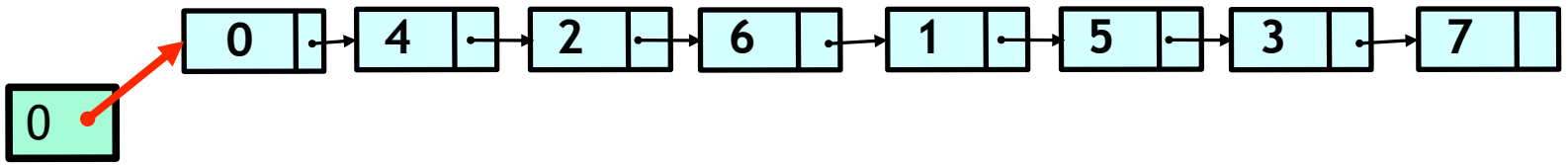


Recursive Split Ordering



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

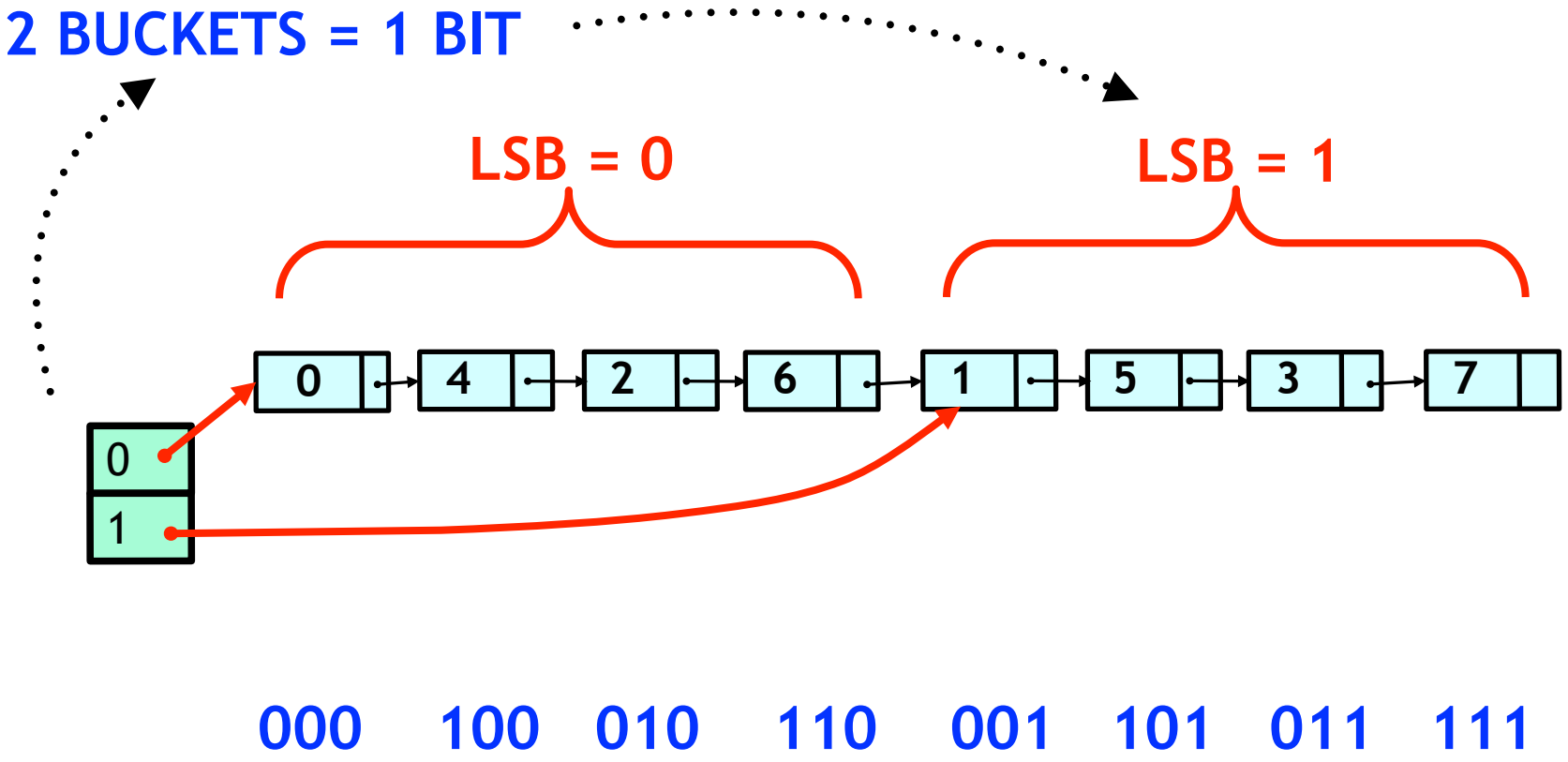
Recursive Split Ordering



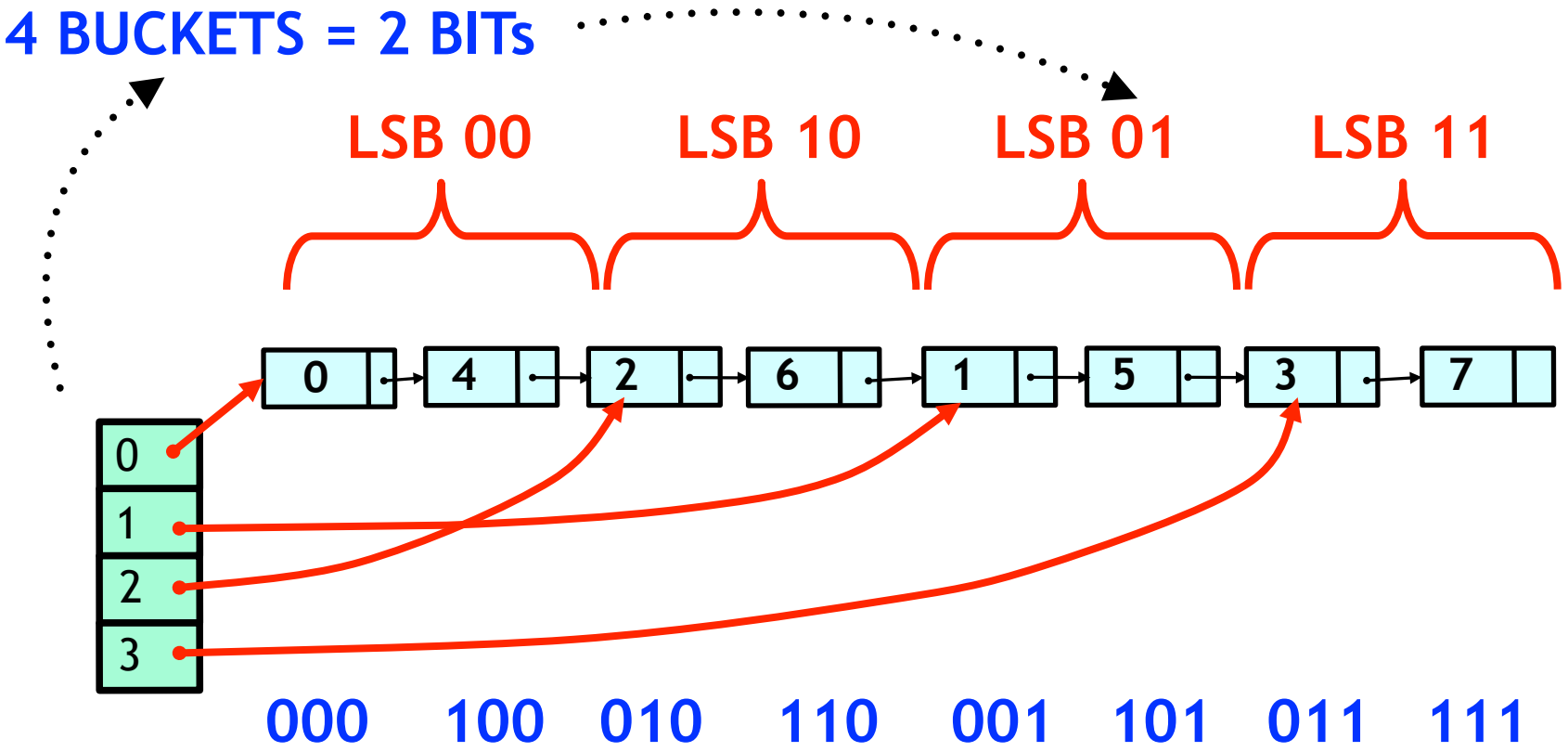
000 100 010 110 001 101 011 111

LSB

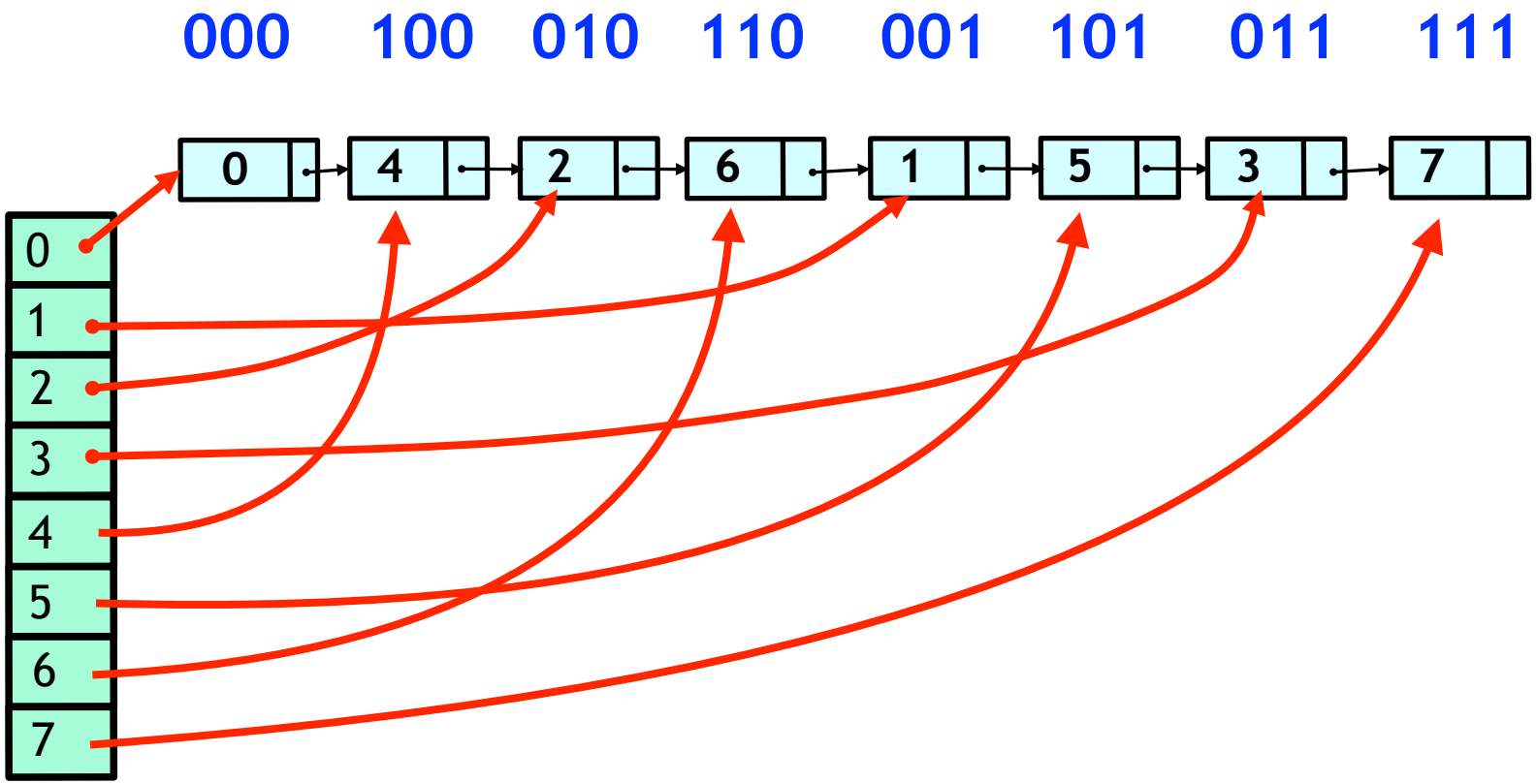
Recursive Split Ordering



Recursive Split Ordering



Recursive Split Ordering



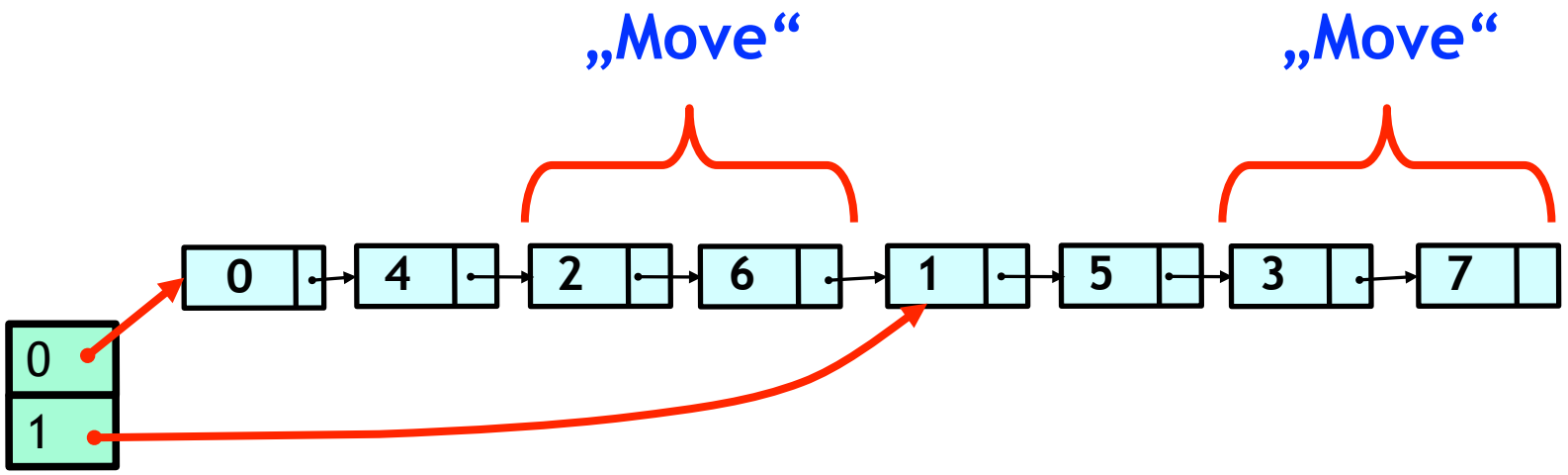
Split-Order

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

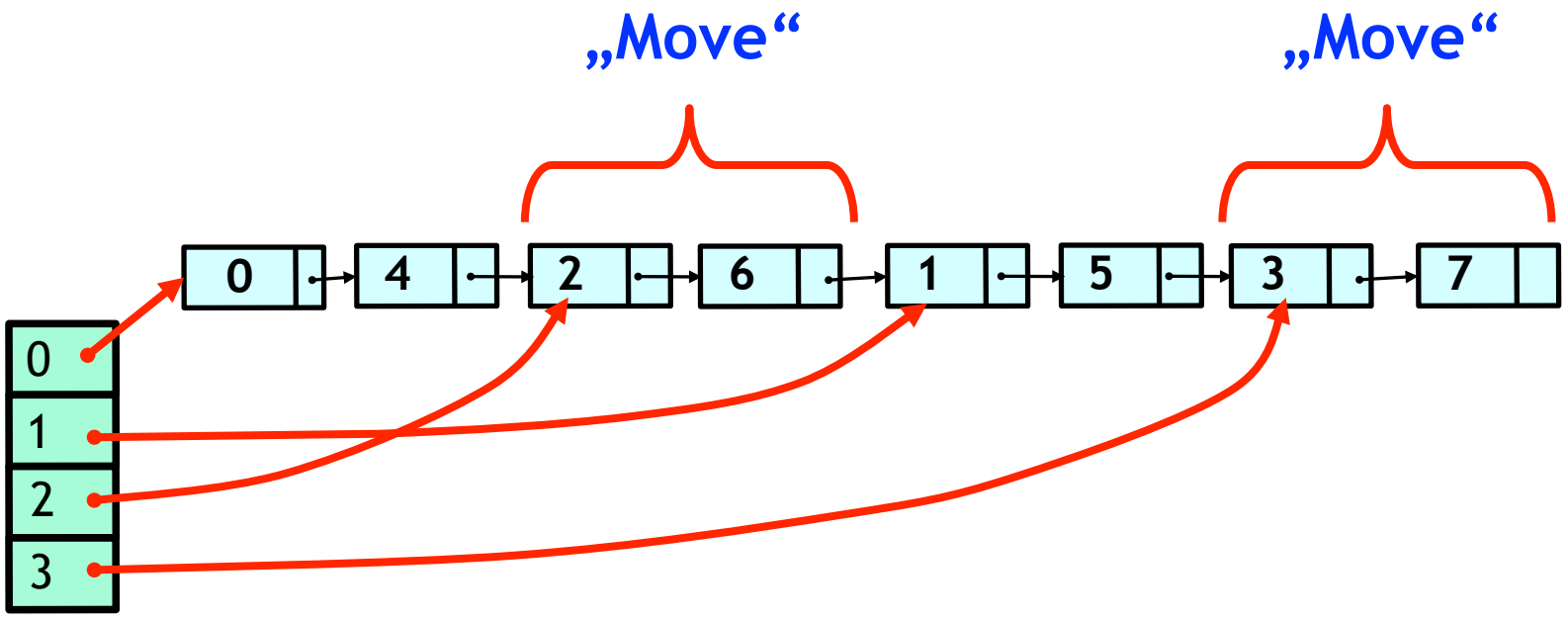
When Table Splits

- 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
- Key must be accessible from both
  - Keys that will move, must come later

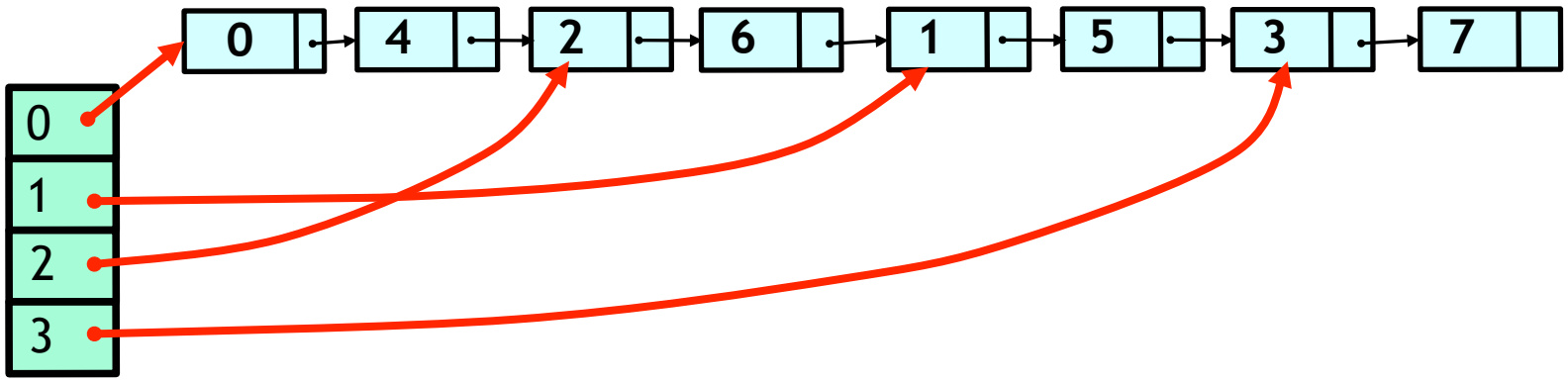
When Table Splits



When Table Splits

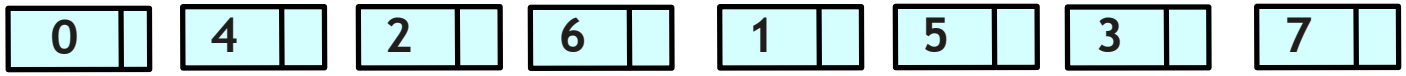


Index of Elements



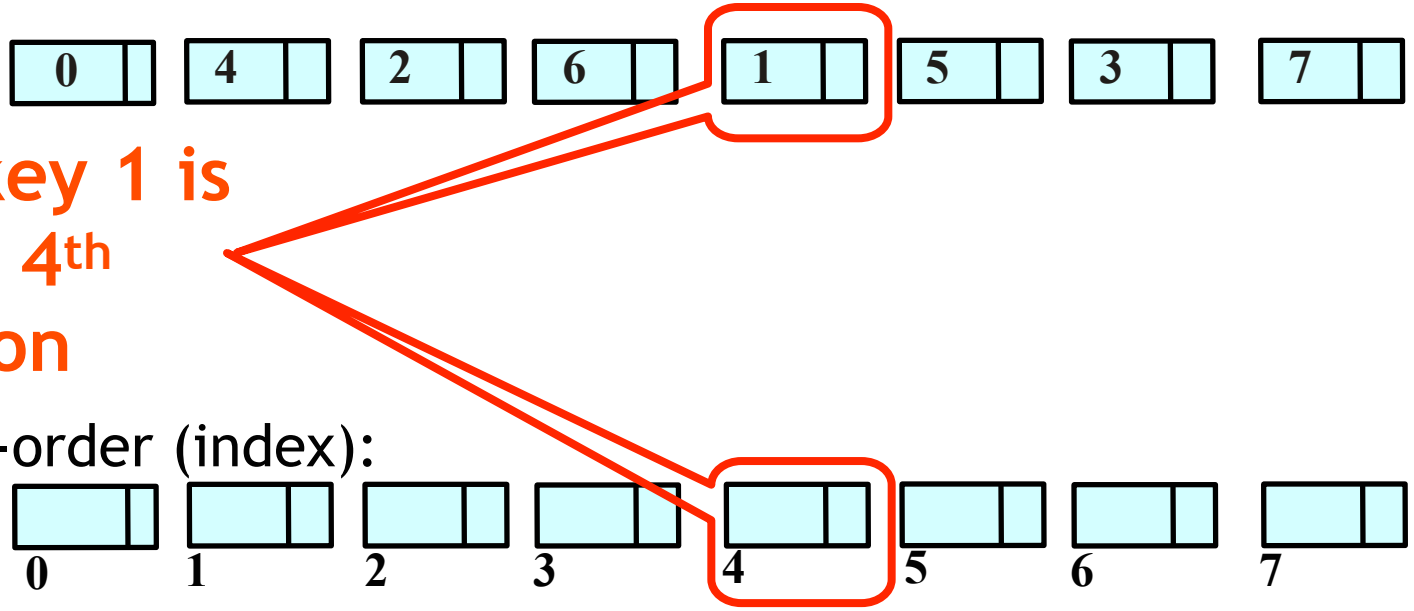
A Bit of Magic

Real keys:

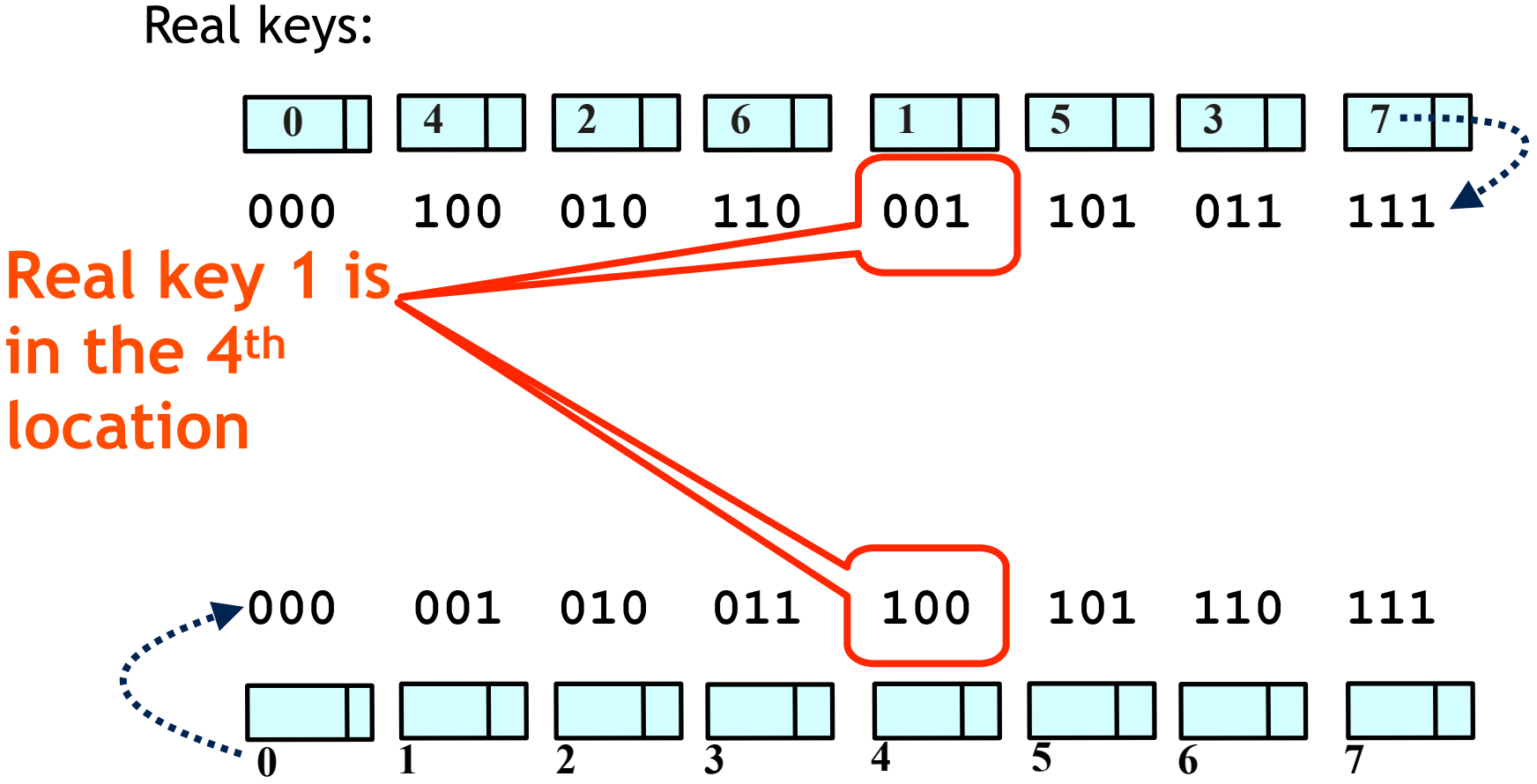


A Bit of Magic

Real keys:



A Bit of Magic



A Bit of Magic

Real keys:

```
000   100   010   110   001   101   011   111
```

Split-order:

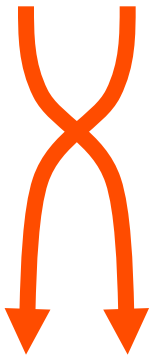
```
000   001   010   011   100   101   110   111
```

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-Order

- Key to bucket mapping
  - LSBs of keys
  - Number of buckets = $2^{(\text{number of LSBs})}$
- Postion of key in list
  - Reverse bit representation of key
  - „6“ -> 110 -> 011 -> position 4

„Homework“

- Postion of key in list
  - „6“ -> 110 -> 011 -> pos. 4
  - „6“ -> 0110 -> 0110 -> pos. 6
  - „6“ -> 00110 -> 01100 -> pos. 12
  - „7“ -> 111 -> 111 -> pos. 7
  - „7“ -> 0111 -> 1110 -> pos.14
  - „7“ -> 00111 -> 11100 -> pos. 28
  - #bits for pos. = #bits of largest key
  - Can #bits change during runtime?

„Homework“

1. Create a list using split-order containing keys „0“ … „7“
   - Use only 3 bits to calc index/position
2. Create a list using split-order containing keys „0“ … „8“
   - Use 4 bits to calculate index/position
3. Compare the order of elements in these lists

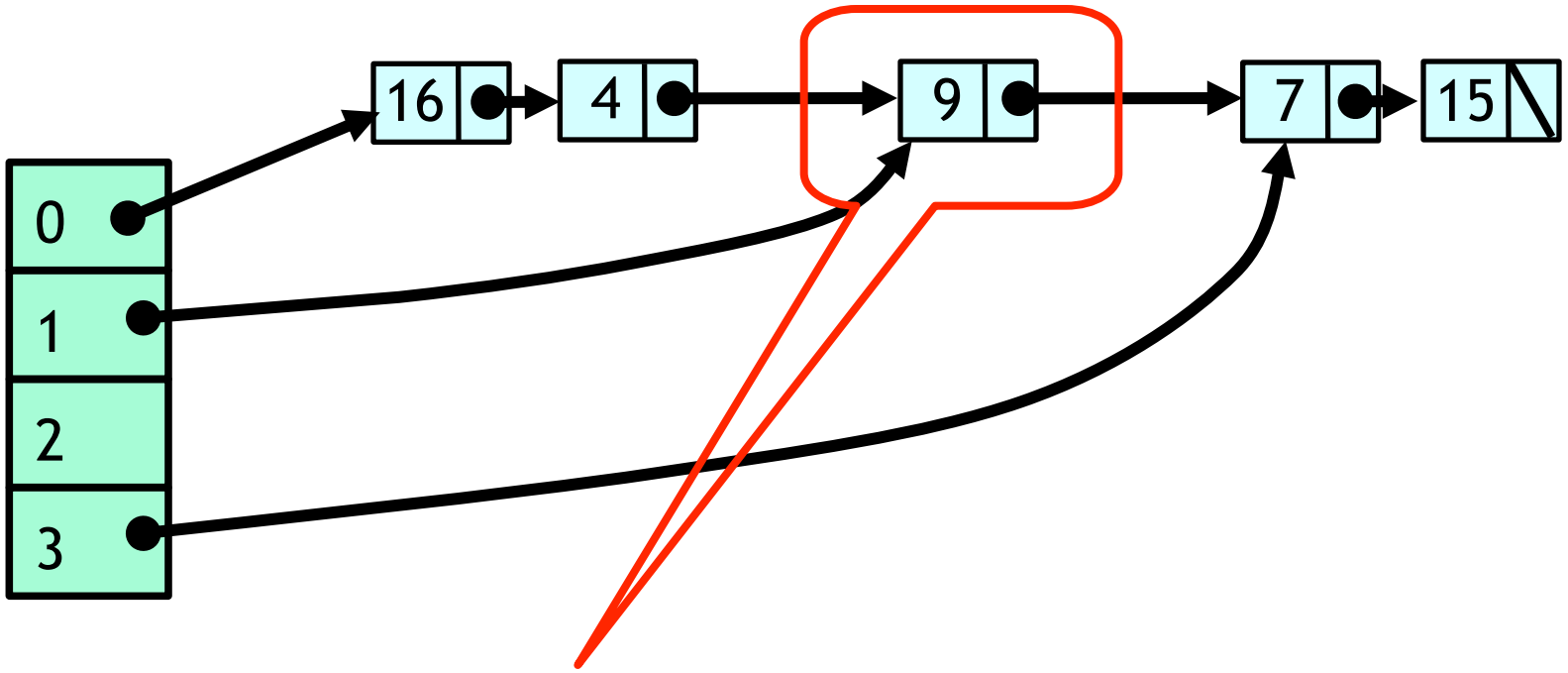
Reverse representations

- „16“ -> 10000 -> 00001 (16 mod 4 = 0)
- „4“ -> 00100 -> 00100 (4 mod 4 = 0)
- „9“ -> 01001 -> 10010 (9 mod 4 = 1)
- „7“ -> 00111 -> 11100 (7 mod 4 = 3)
- „15“ -> 01111 -> 11110 (15 mod 4 = 3)

4 Buckets = 2 LSBs

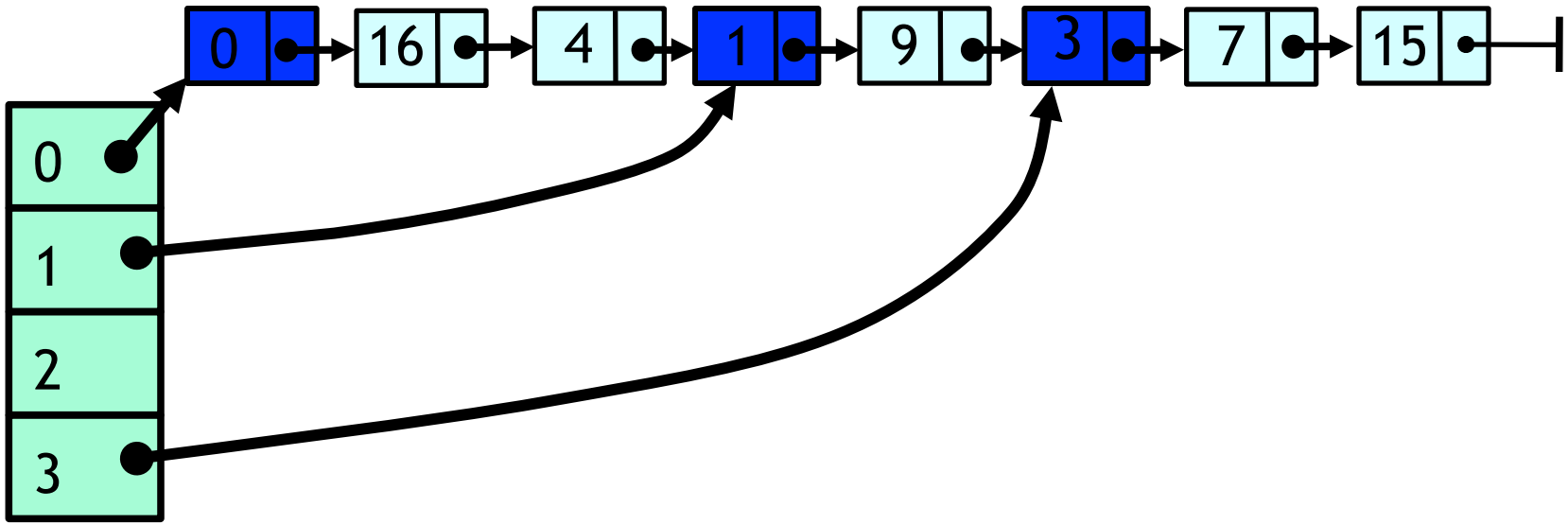
Reverse binary representation is in order

Sentinel Nodes



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

Sentinel Nodes



Solution: use a Sentinel node for each bucket

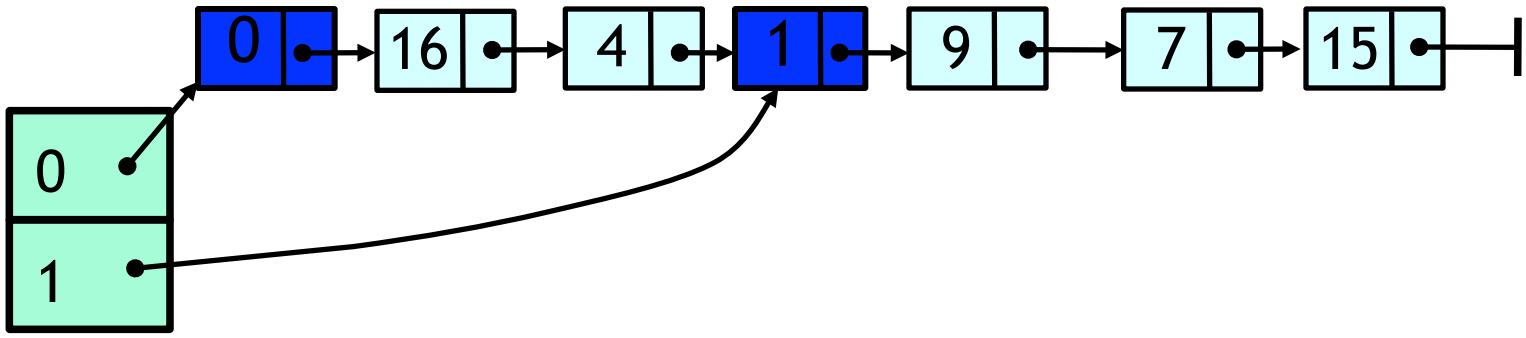
Sentinel vs Regular Keys

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

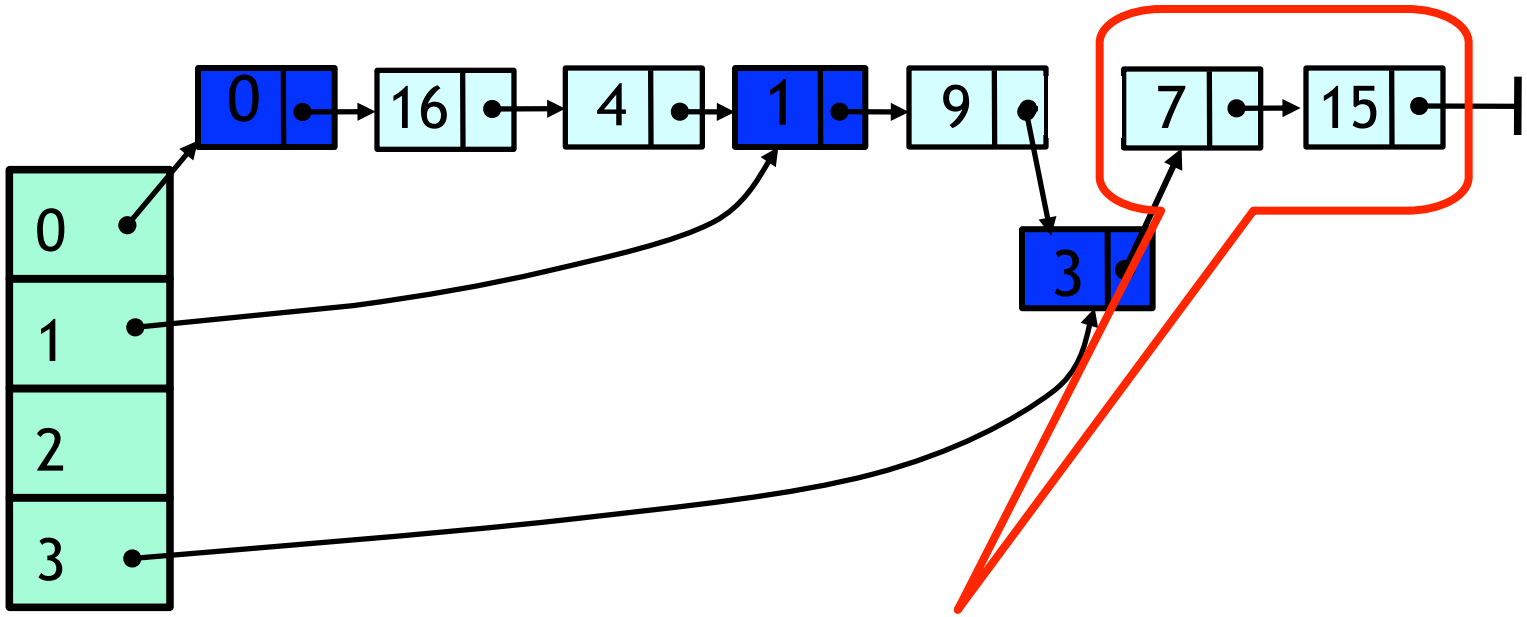
Splitting a Bucket

- We can now split a bucket
- In a lock-free manner
- Using two CAS() method calls
- By initializing the bucket that splits it to point to a new sentinel node

Initialization of Buckets



Initialization of Buckets

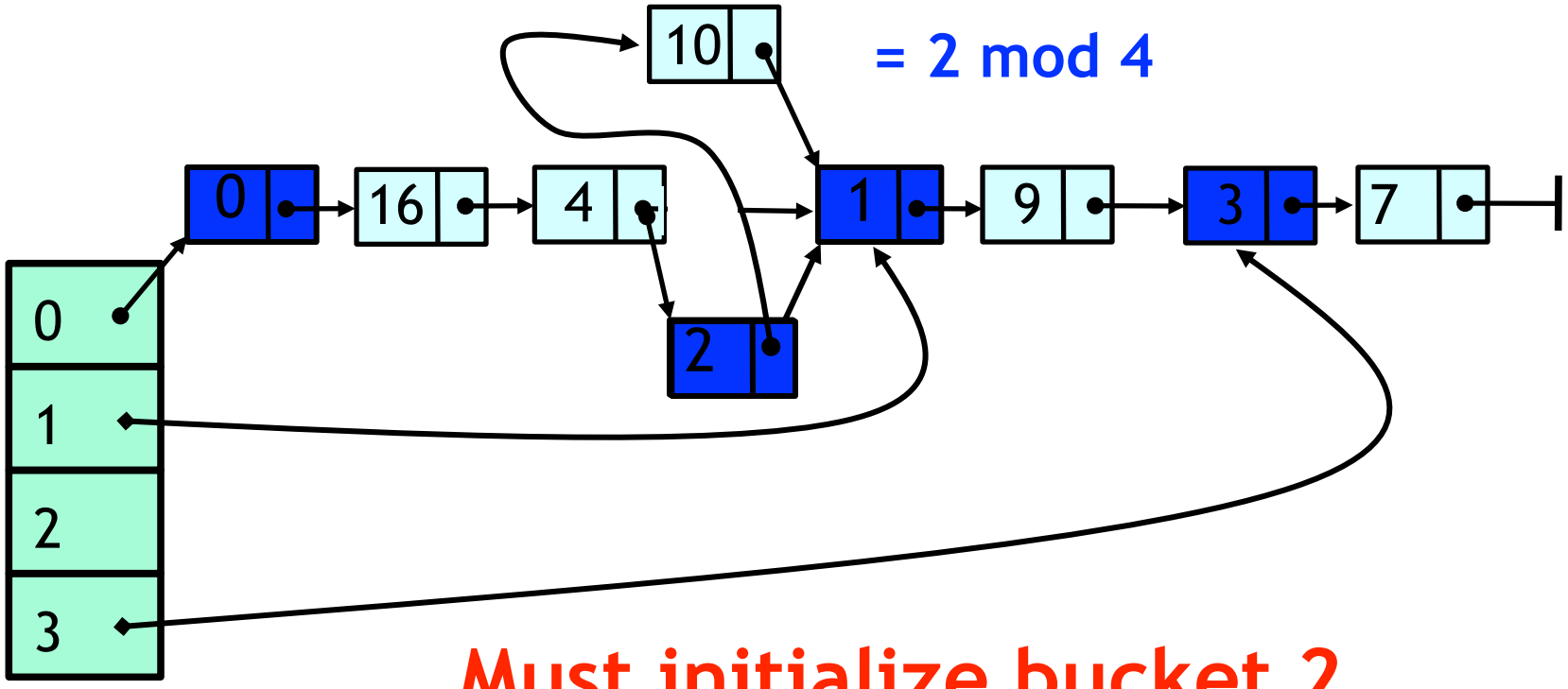


3 in list but not connected to bucket yet

Need to initialize bucket 3 to split bucket 1

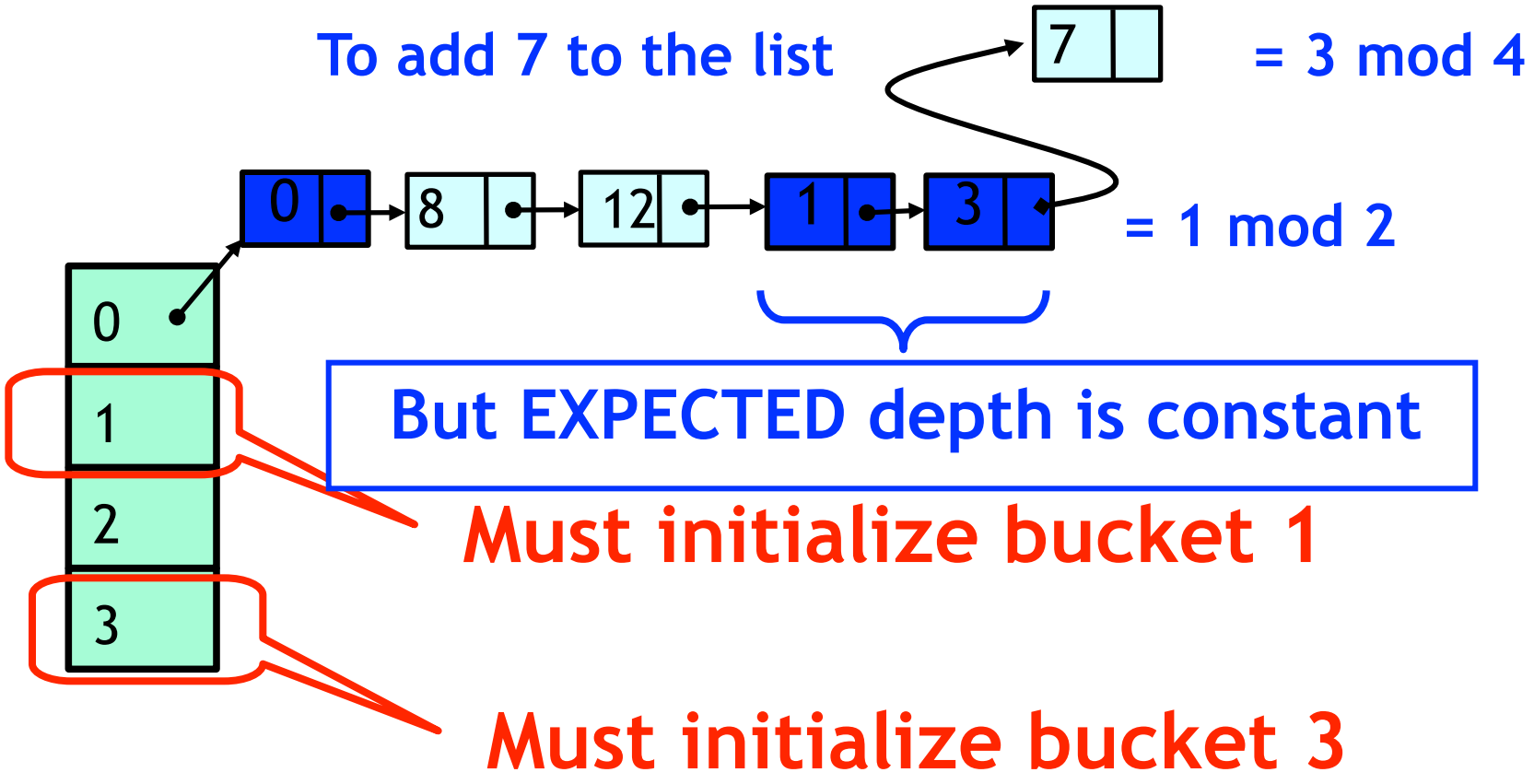
Now 3 points to sentinel – bucket has been split

Adding 10



Must initialize bucket 2
Then can add 10

Recursive Initialization



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
  Lock
                    int key) {...};
}
```

Change: add takes key argument

Lock-Free List

Inserts sentinel with key if not already present …

```
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 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();
    tableSize = new AtomicInteger(2);
    setSize = new AtomicInteger(0);
  }
```

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(2);
    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(2);
    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 1 bucket and size is zero

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

Get pointer to bucket's sentinel, initializing if necessary

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;
}
```

Call bucket's add() method with reversed key

Add() Method

No change? We're done.

```
public boolean add(T 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

Time to resize?

```
public boolean add(T 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;
}
```

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

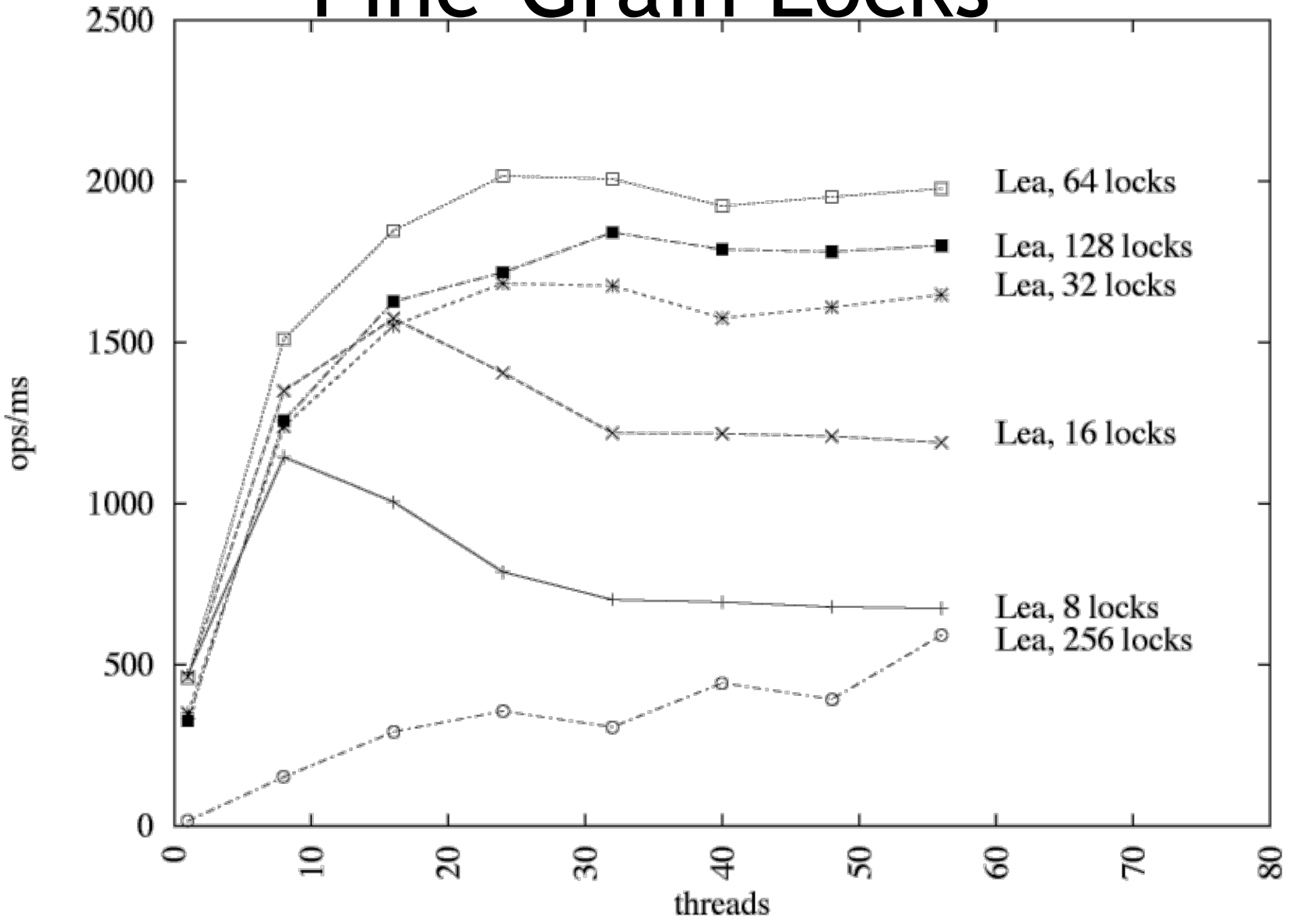
```
void initializeBucket(int bucket) {
 int parent = getParent(bucket);
 if (table[parent] == null)
  initializeBucket(parent);
 int key = makeSentinelKey(bucket);
 LockFreeList list =
   new LockFreeList(table[parent],
                    key);
 }
```

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

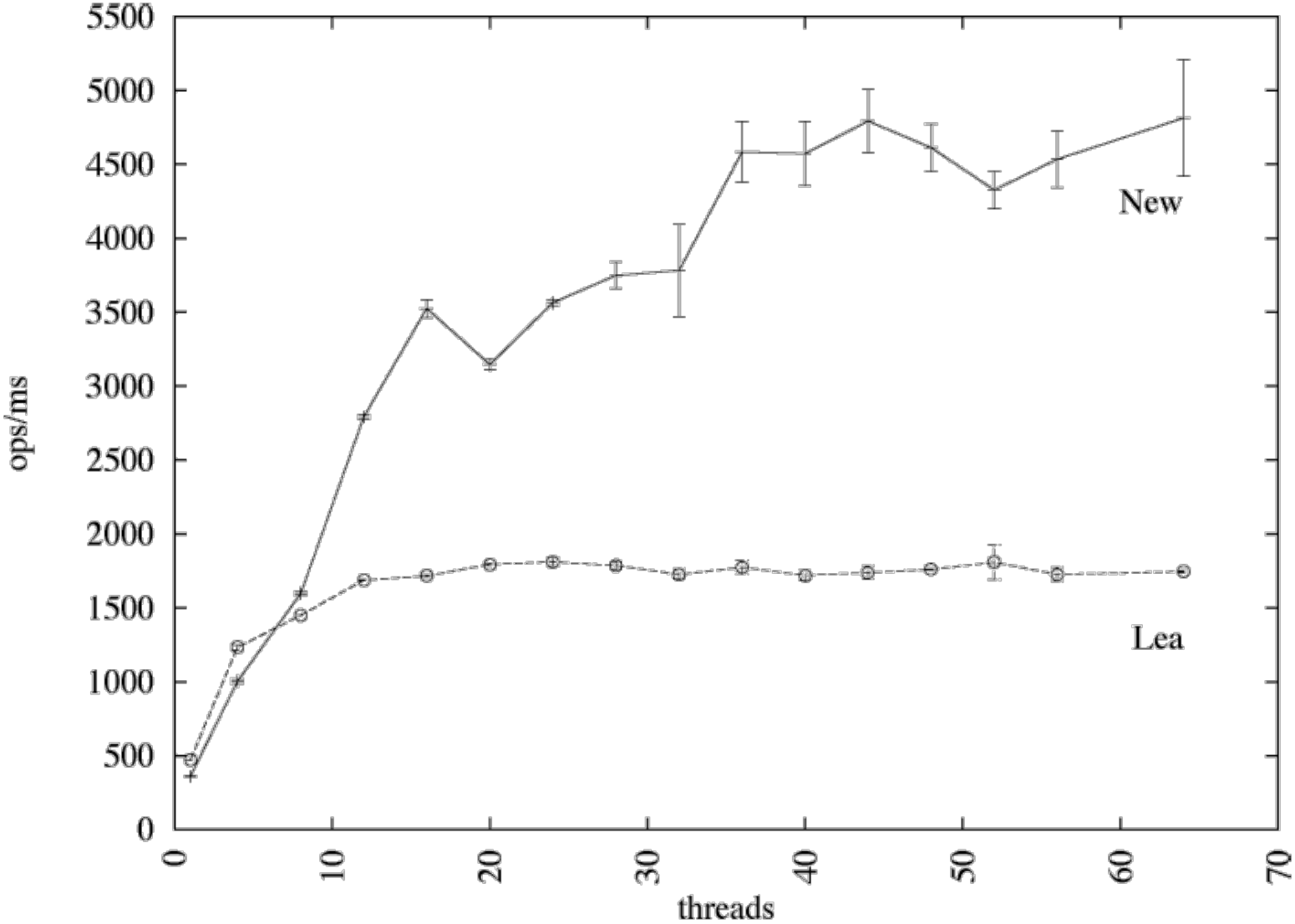
Empirical Evaluation

- On a 30-processor Sun Enterprise 3000
- Lock-Free vs. fine-grained (Lea) optimistic
- In a non-multiprogrammed environment
- 10^6 operations: **88%** contains(), **10%** add(), **2%** remove()

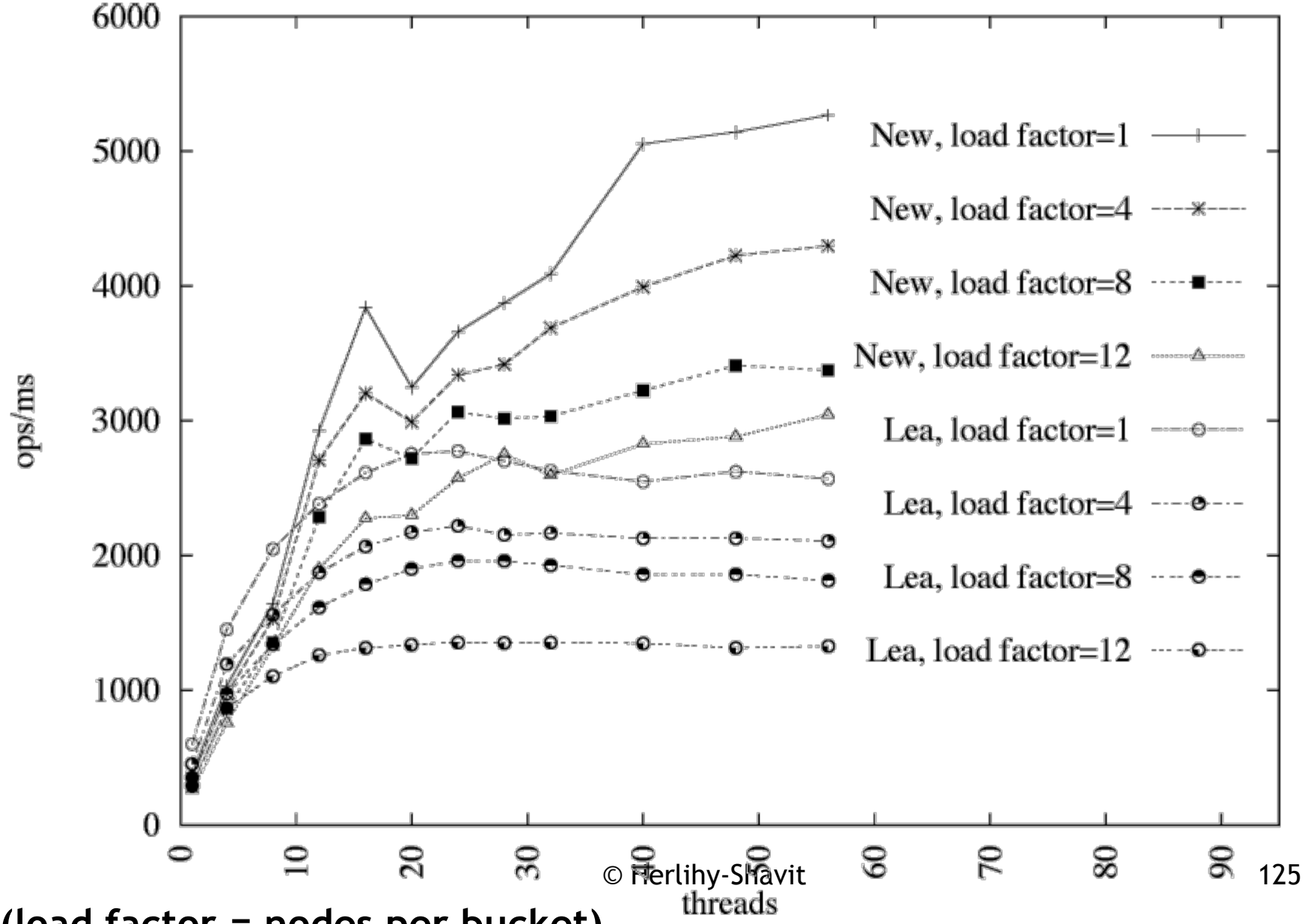
Fine-Grain Locks



Fine-Grain vs Lock-Free

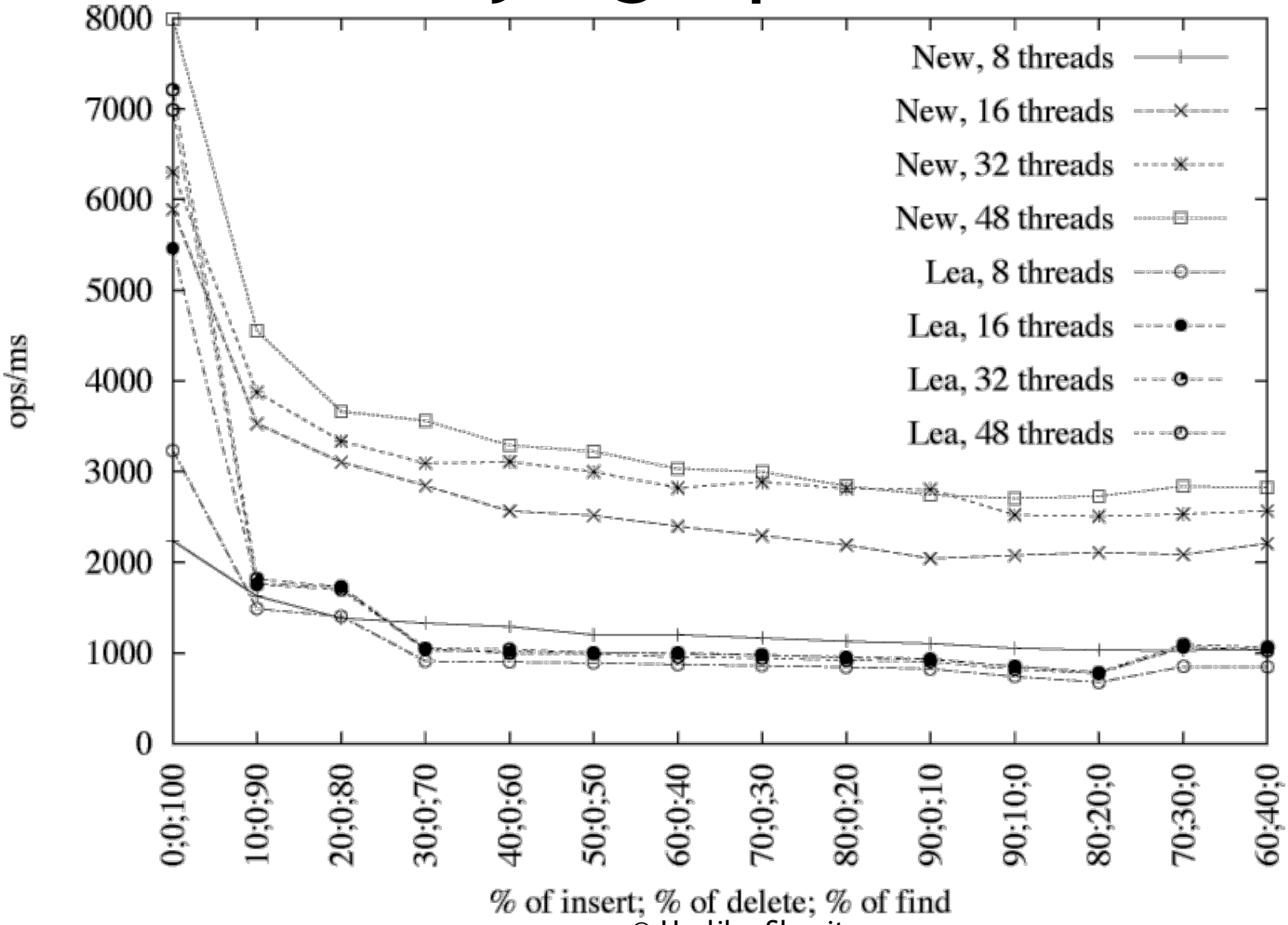


Hash Table Load Factor



(load factor = nodes per bucket)

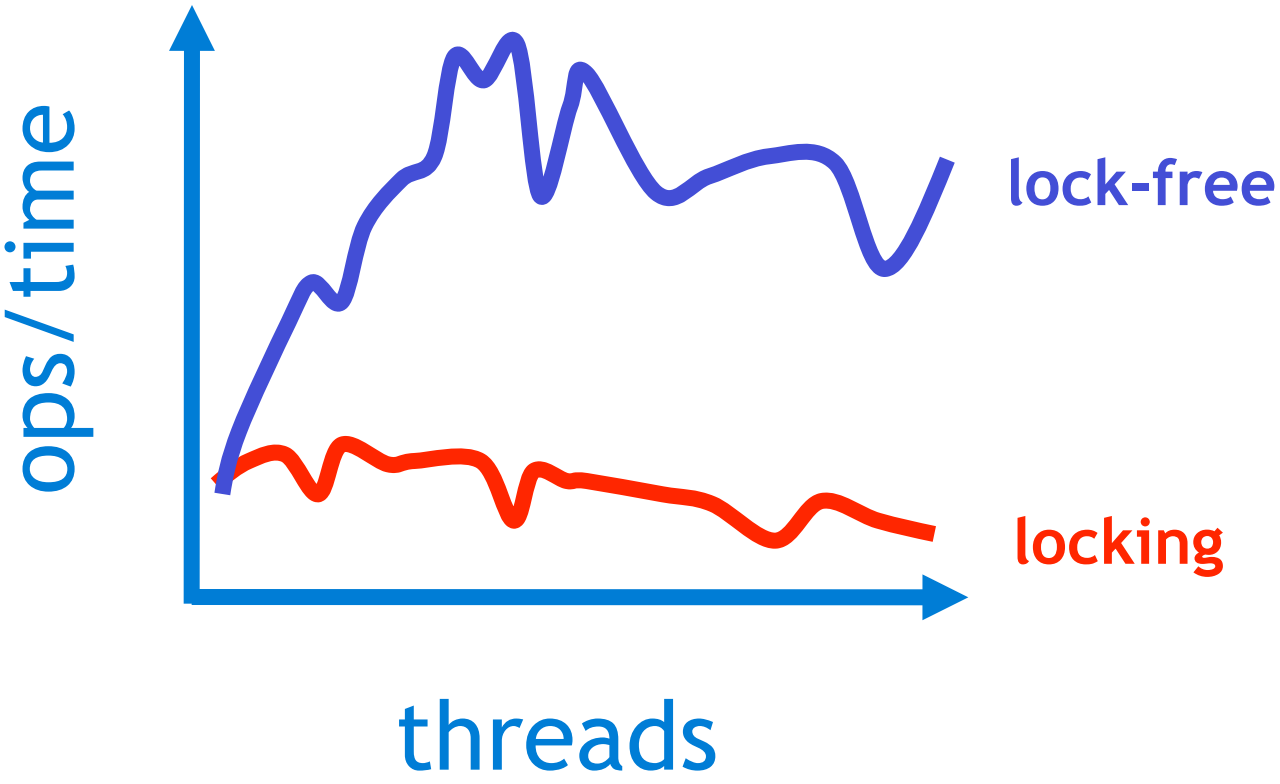
Varying Operations



Summary

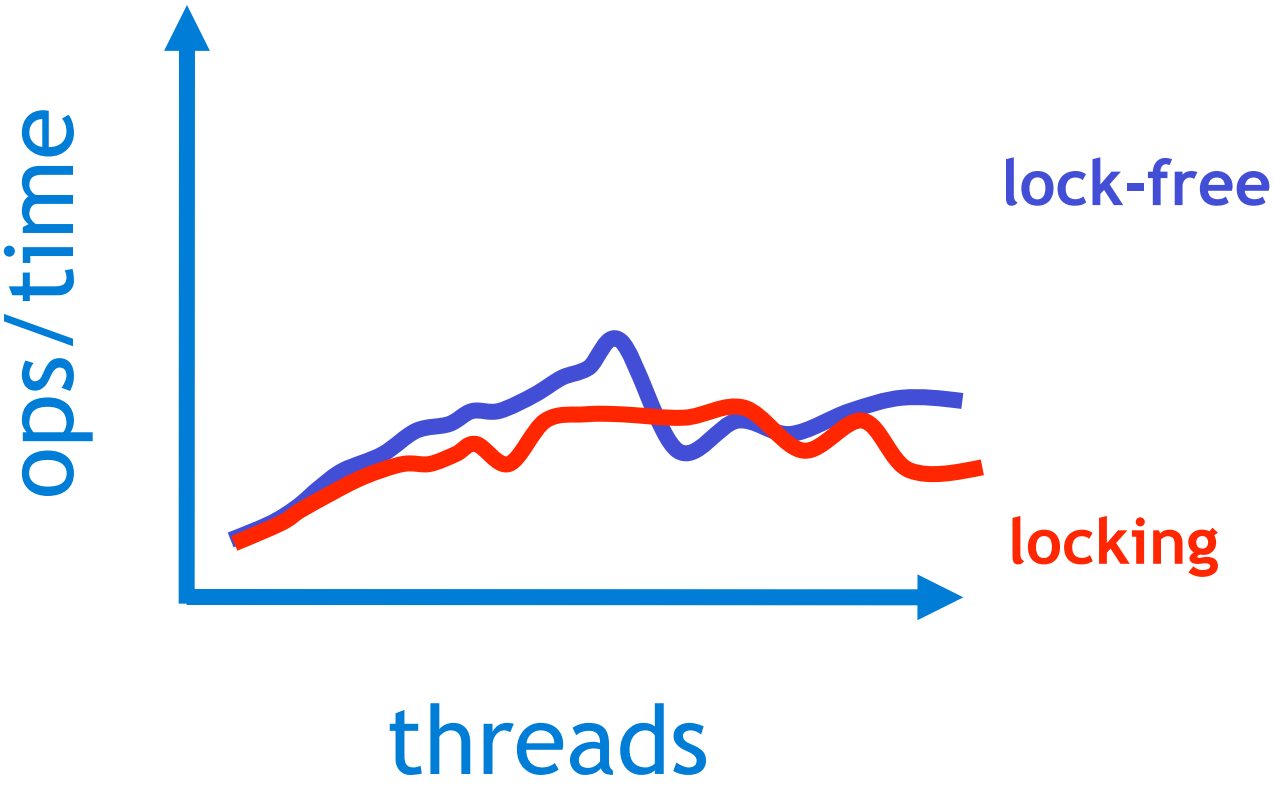
- Concurrent resizing is tricky
- Lock-based
  - Fine-grained
  - Read/write locks
  - Optimistic
- Lock-free
  - Builds on lock-free list

Work = 0



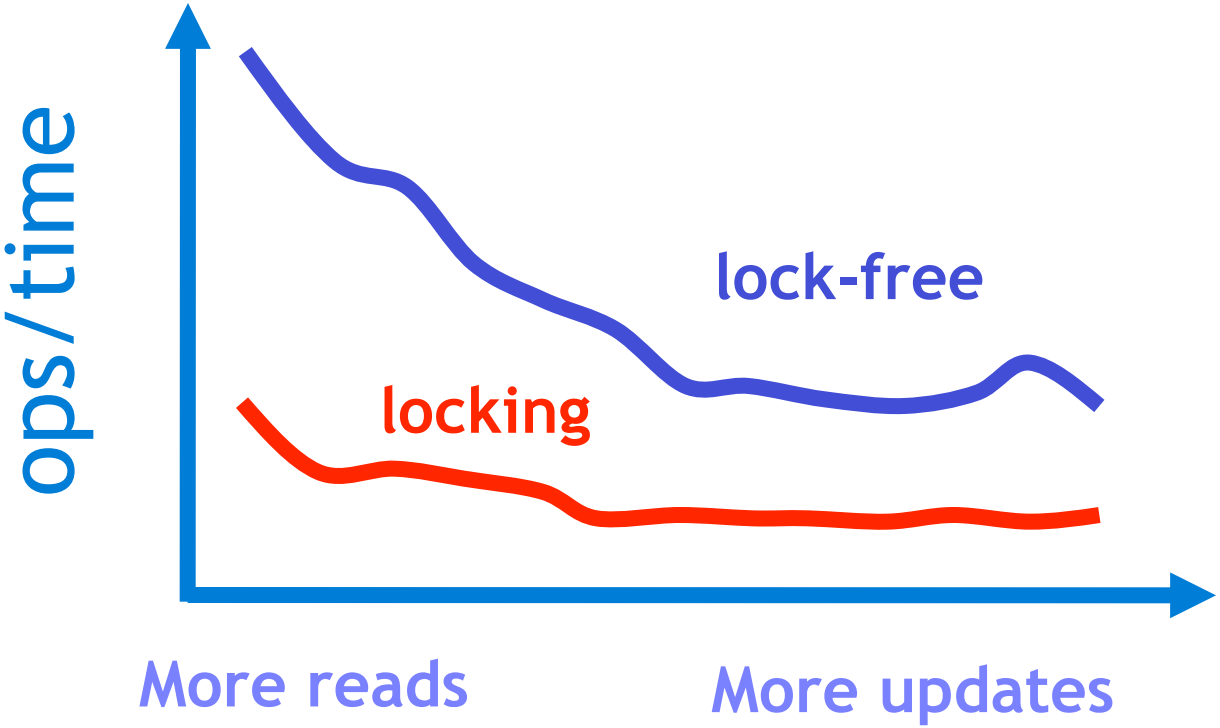
Adapted from Shalev & Shavit 2003

Work = 500



Adapted from Shalev & Shavit 2003

Varying The Mix



64 threads

Adapted from Shalev & Shavit 2003