Concurrency:
Multi-core Programming & Data Processing

Concurrent Hashing

Prof. P. Felber

Pascal.Felber@unine.ch

http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

List-Based Sets and Beyond

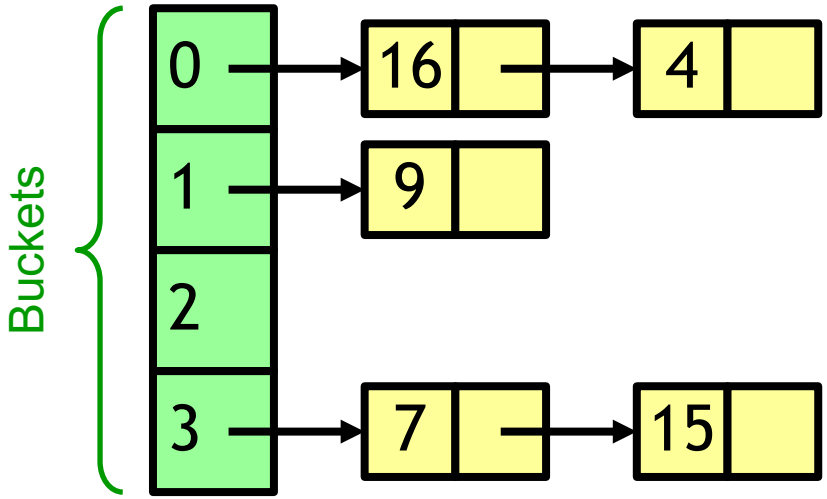
- We looked at a number of ways to make highly-concurrent lists, queues, stacks
  - Fine-grained locks
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- What's missing?
  - `add()`, `remove()`, `contains()` methods all take time linear in set size
- We want constant-time methods (on average)

Hashing

- Hash function
  - `h(object) → integer`
- Uniformly distributed
  - Different objects most likely have different hash values
- Java `hashCode()` method fingerprint of an object

Sequential Hash Table

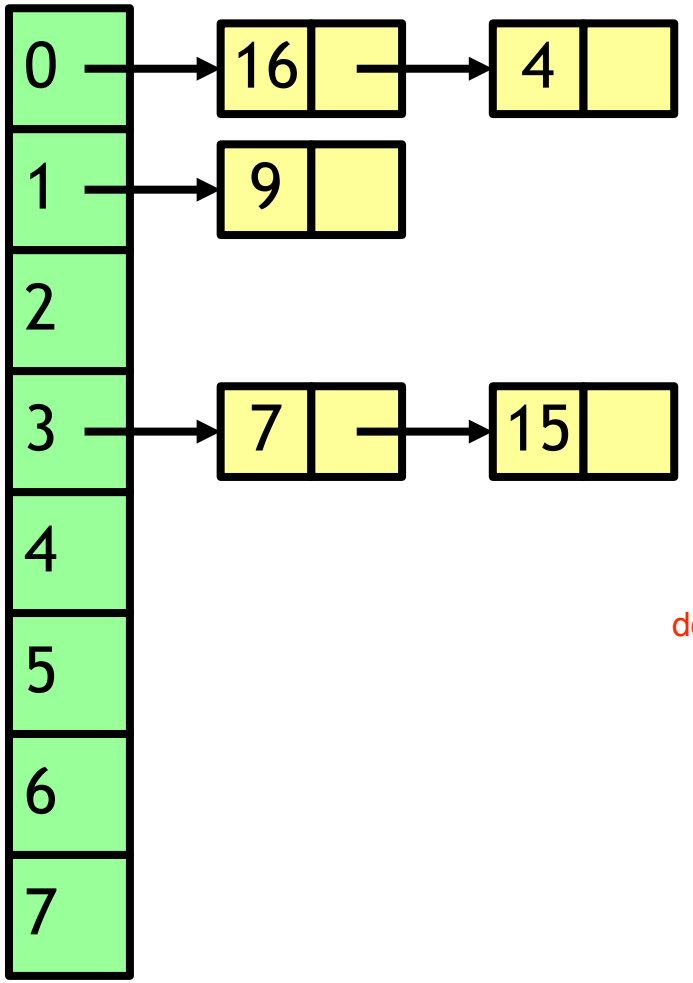
buckets: classical way of implementing hash tables



$h(k) = k \% 4$

Item count: 5

Resizing the Table



$h(k) = k \% 4$

↓

$h(k) = k \% 8$

Must redistribute items!

doubling results in average moving half of the object

Simple Hash Set

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

Array of lock-free lists

Construct with initial size

Use object hash code to pick a bucket

Call bucket's add() method

No Brainer?

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

- We do not 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

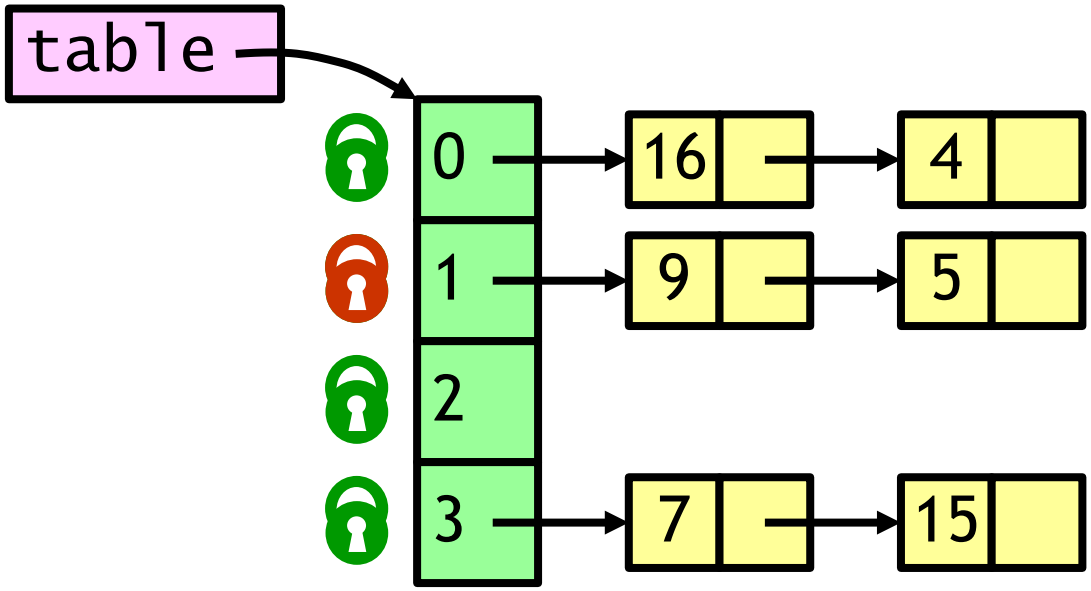
- For instance:
  - Bucket threshold
    - When ≥ ¼ buckets exceed this value
  - Global threshold
    - When any bucket exceeds this value

Coarse-Grained Locking

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

Fine-Grained Locking: Add

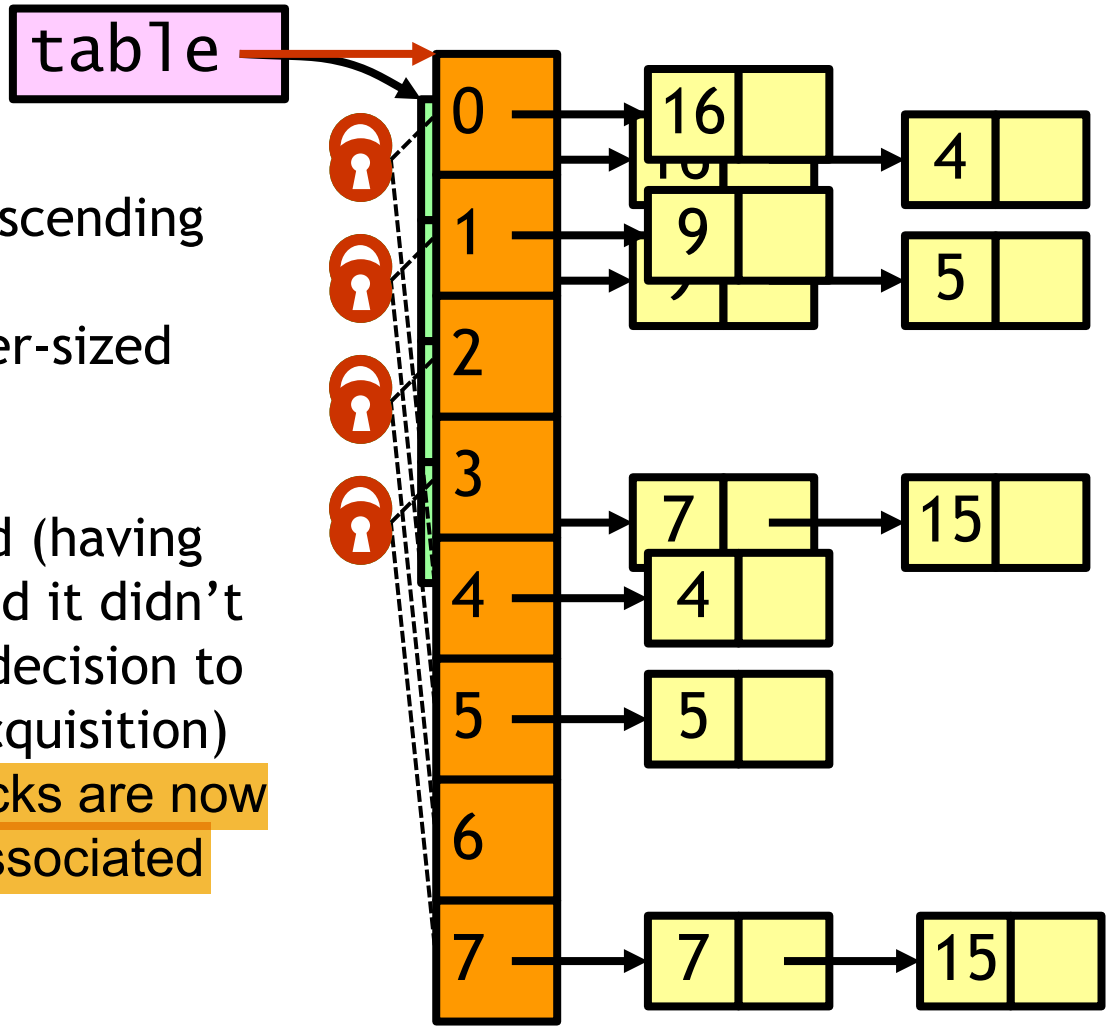
add(5)



when you want to resize you lock everything

Fine-Grained Locking: Resize

1. Acquire locks in ascending order
2. Allocate new super-sized table
3. Move items
4. Update table field (having previously checked it didn't change between decision to resize and lock acquisition)
5. Release locks (locks are now striped: each is associated with two buckets)



Observations

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

Fine-Grained Hash Set

```
public class FGHashSet {
  private Object[] lock;
  private List[] table;
  public FGHashSet(int capacity) {
    table = new List[capacity];
    lock = new Object[capacity];
    for (int i = 0; i < capacity; i++) {
      lock[i] = new Object();
      table[i] = new LinkedList();
    }
  }
  …
```

Array of locks (monitors here)

Array of buckets

Initially same number of locks and buckets

Fine-Grained Hash Set

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

Which bucket?

Which lock?

Acquire lock

Call bucket's add() method

Fine-Grained Hash Set

recursive resize

```
private void resize(int depth, List[] oldTab) {
  synchronized (lock[depth]) {
    if (depth == 0 && oldTab != table)
      return;
    int next = depth + 1;
    if (next < lock.length)
      resize (next, oldTab);
    else
      sequentialResize();
  }
}
public resize() { resize(0, table); }
```

Acquire next lock

Check that no one else has resized

Recursively acquire next lock

Locks acquired, do the work

Fine-Grained Locks

- We can resize the table, but not the locks
- Debatable whether method calls are linear-time in presence of contention…
- The `contains()` method does not modify any fields
  - Why should multiple `contains()` methods conflict?
- Idea: replace locks by read/write locks

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

Basic 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?

issue of fairness: 3 readers/ 1 writer. writer can be oventaken by reader. writer has to wait until there is no reader anymore

FIFO Read/Write Lock

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

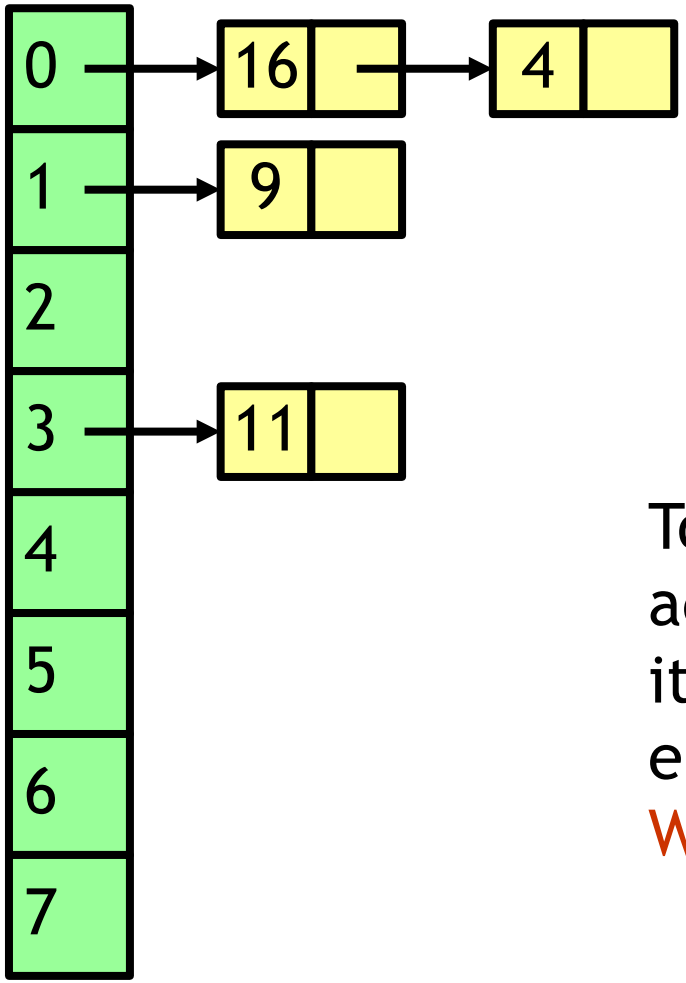
The Story So Far

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

Optimistic Synchronization

- The `contains()` method
  - Scans without locking
- If it finds the key
  - OK to return true
- If it does not 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



$h(k) = k \% 4$

$\downarrow$

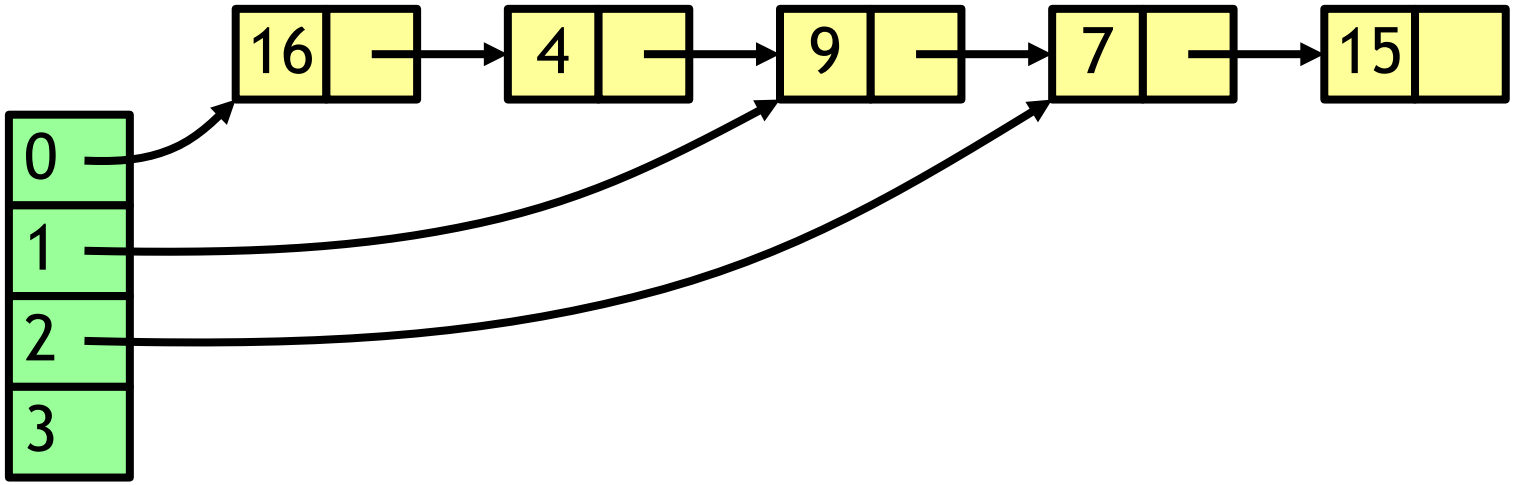
$h(k) = k \% 8$

To remove and then add even a single item, one CAS not enough…

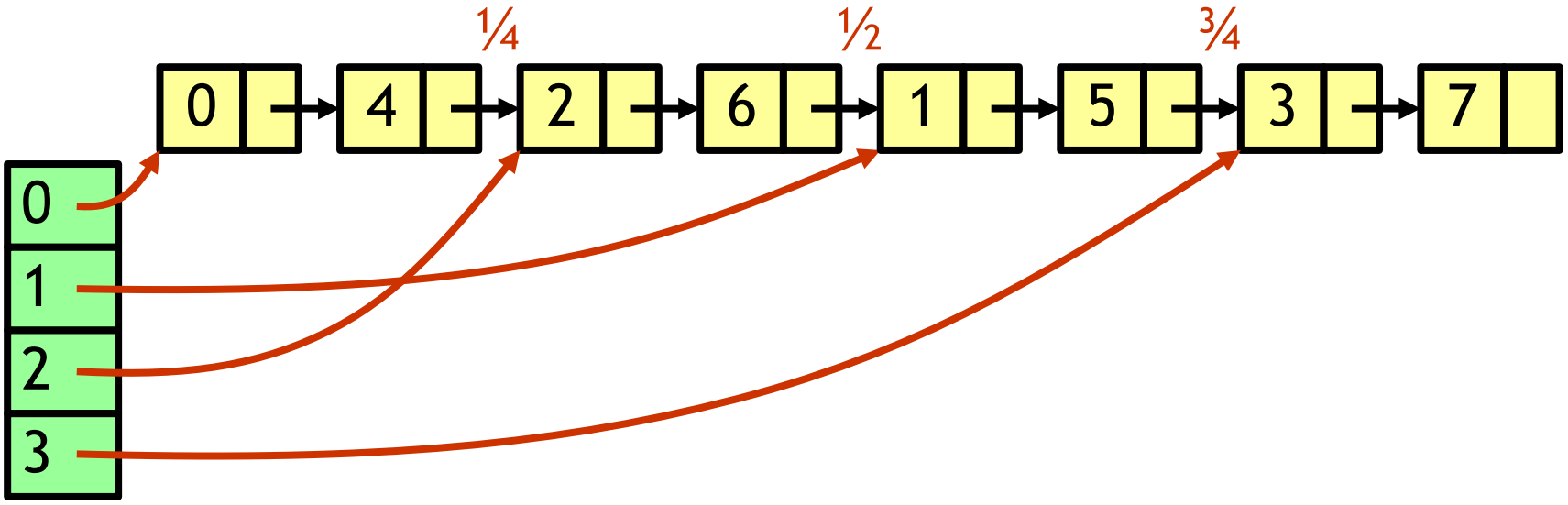
We need a new idea!

Idea: Do not Move the Items

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



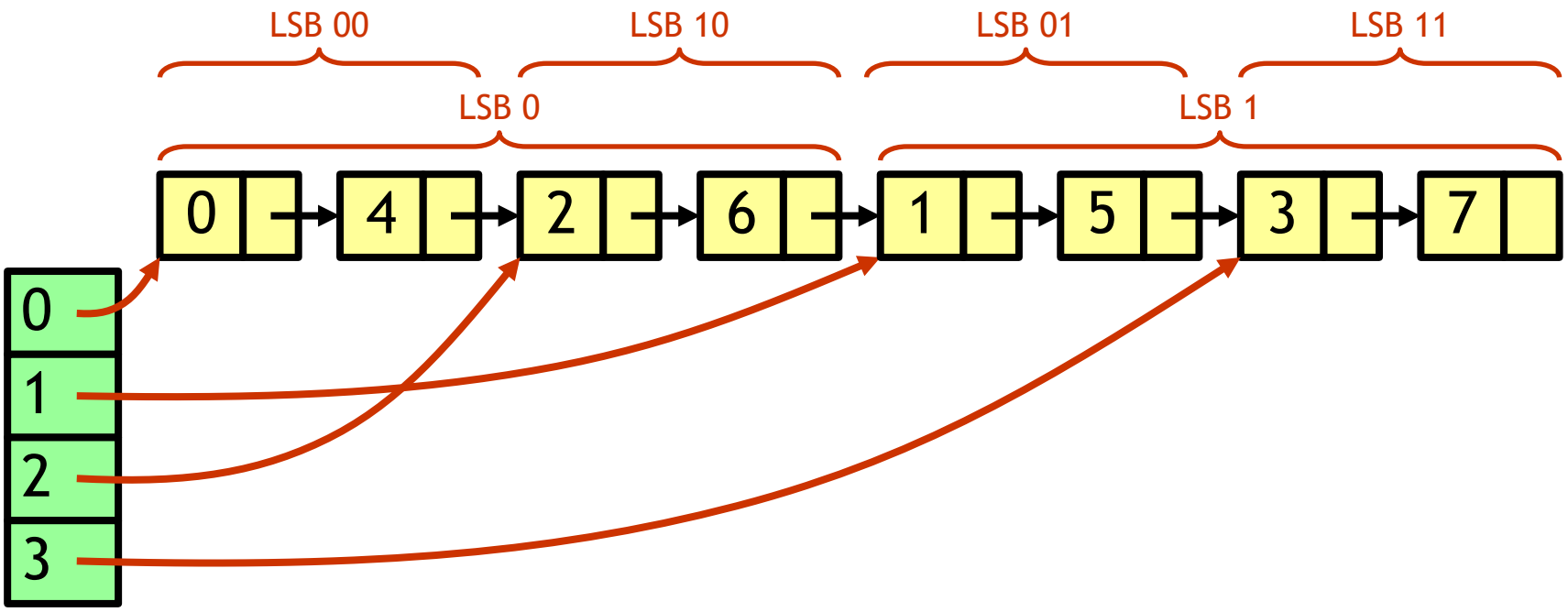
Recursive Split Ordering



List entries sorted in order that allows recursive splitting…

How?

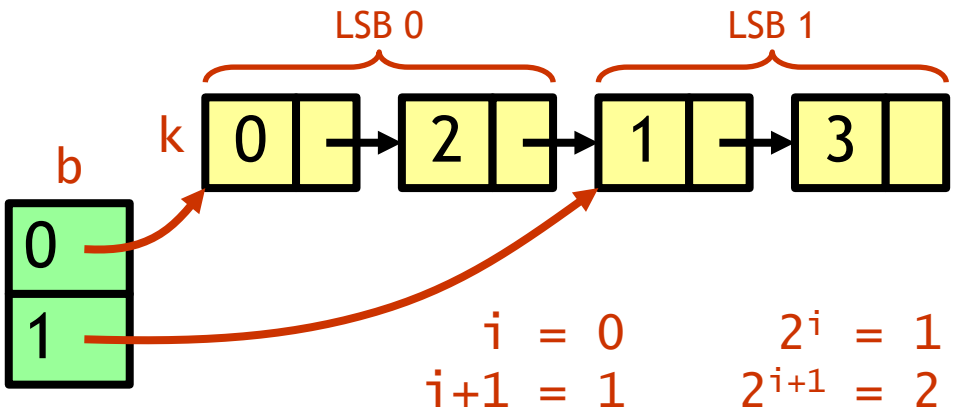
Recursive Split Ordering



- If the table size is 2^i
  - Bucket b contains keys $k = b \% 2^i$
  - Bucket index is key's i LSBs

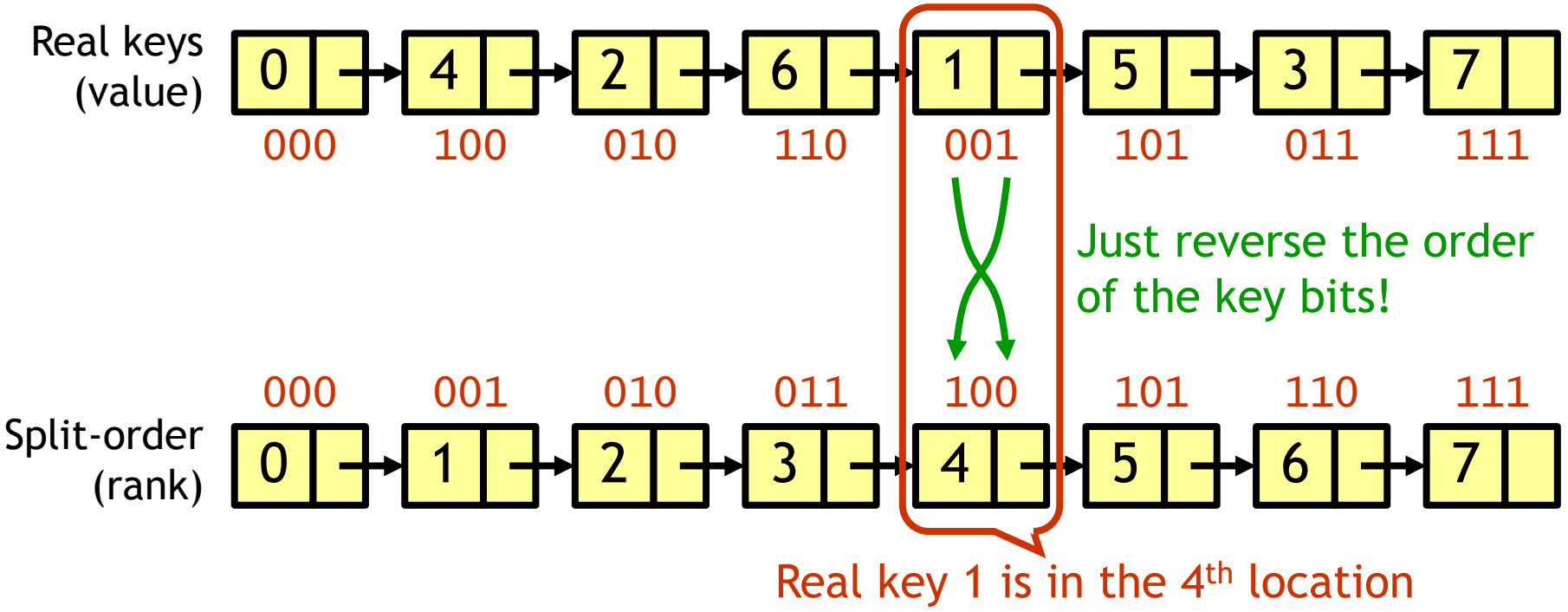
When Table Splits

- When resizing
  - $2^i \rightarrow 2^{i+1}$
- Some keys stay
  - $b = k \% 2^{i+1}$
- Some move
  - $b + 2^i = k \% 2^{i+1}$
- Determined by the bit at position $i+1$
  - Counting backwards (from LSB)
- Keys must be accessible from both
  - Keys that will move must come later



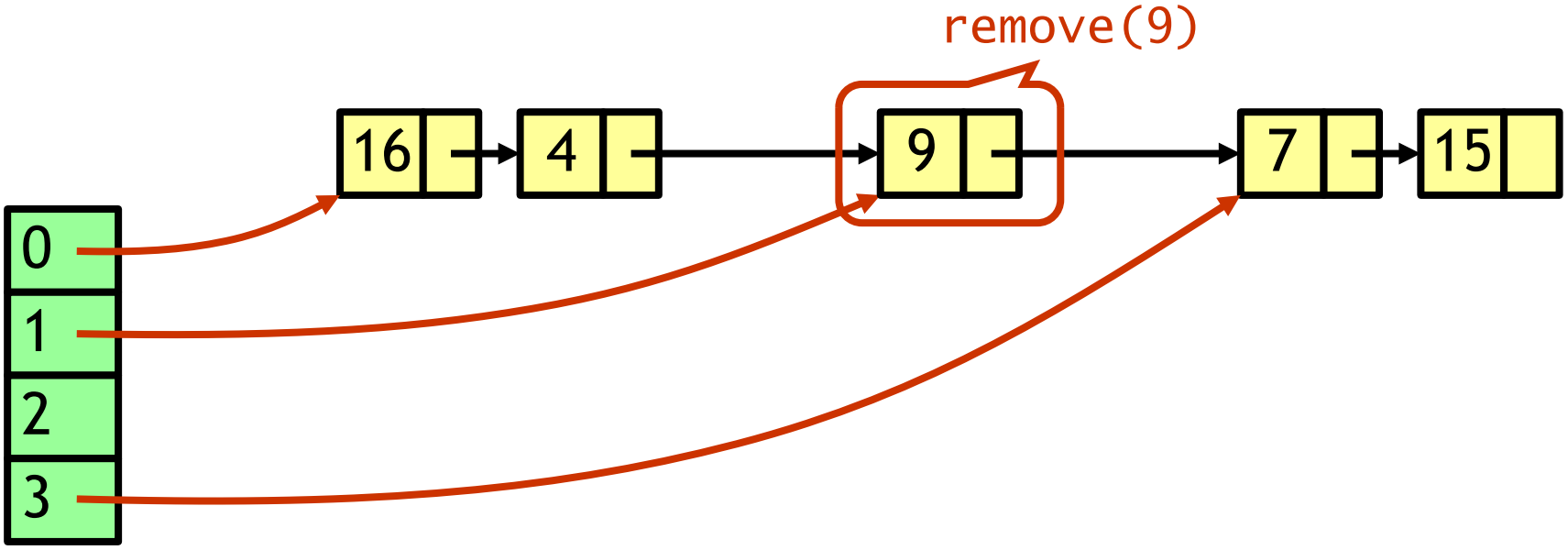
A Bit of Magic

sort them by the reverse of the bits. allows recursive splits



Given two keys a and b, a precedes b if and only if the bit-reversed value of a is smaller than the bit-reversed value of b

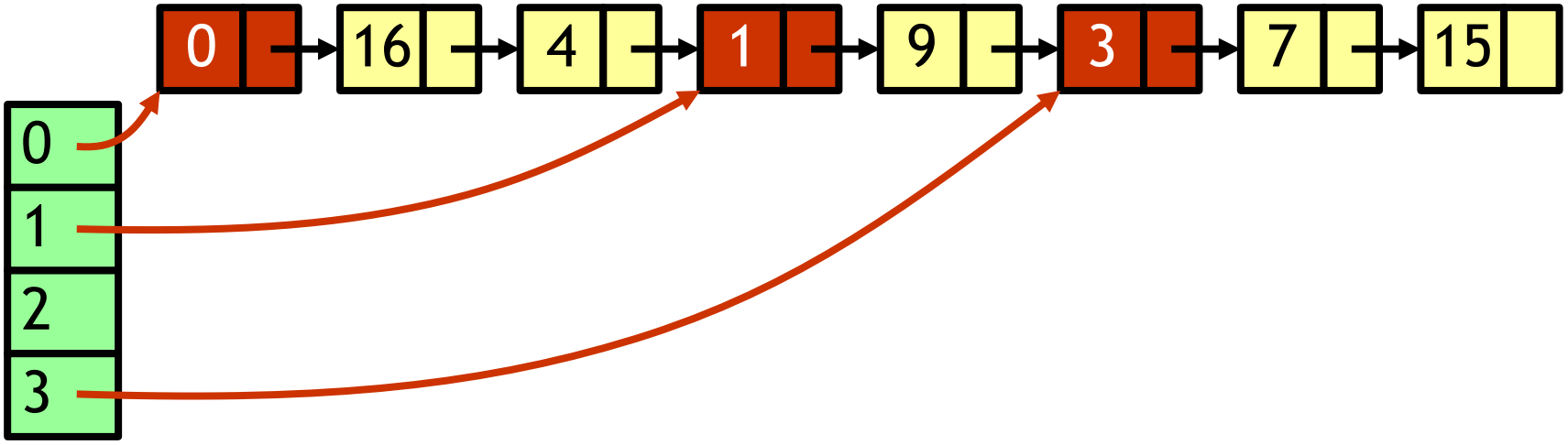
Sentinel Nodes



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

one CAS is not enough to switch 2 pointers

Sentinel Nodes



Solution: use a sentinel node for each bucket (never removed from list)

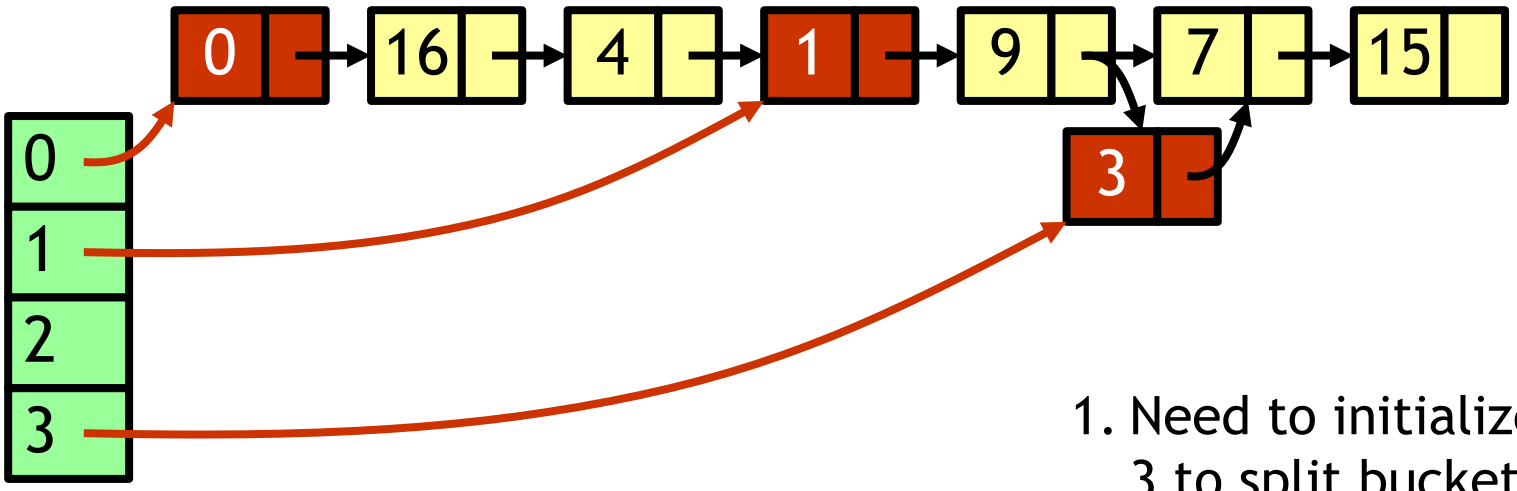
Sentinel key `i` comes before all keys that hash to bucket `i` and after all keys that hash to bucket `(i-1)`

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

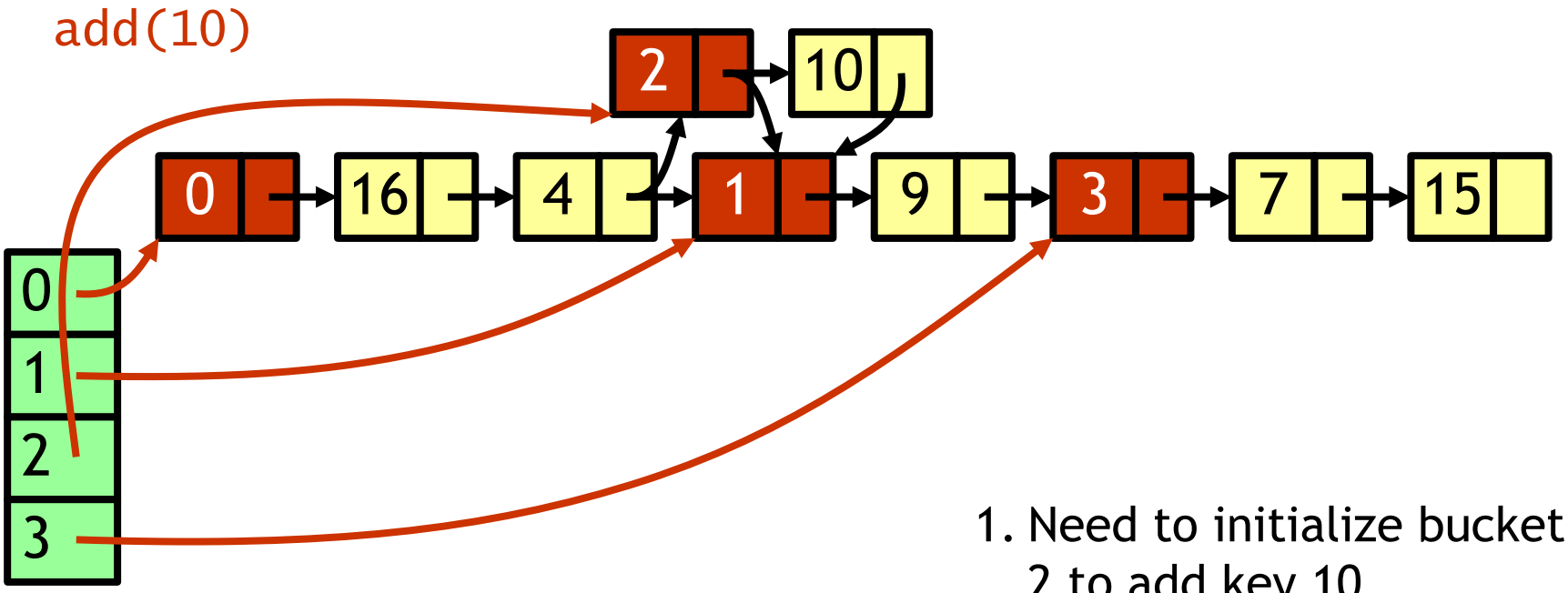
Initialization of Buckets

`resize()`



1. Need to initialize bucket 3 to split bucket 1
2. Add sentinel node to list
3. Sentinel 3 in list but not connected to bucket yet
4. Now 3 points to sentinel, bucket has been split

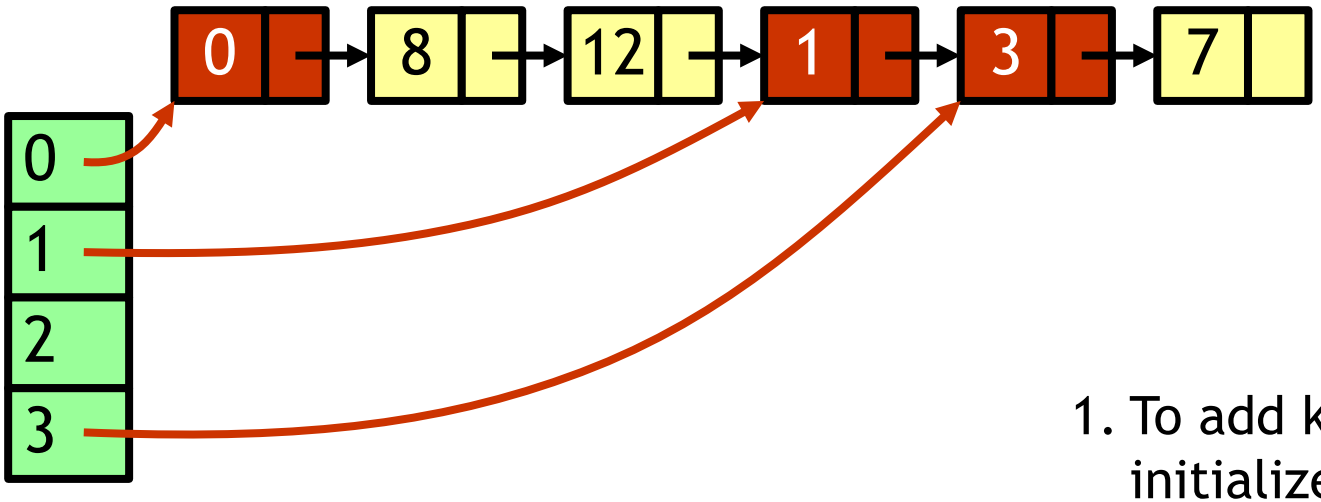
Adding Element



1. Need to initialize bucket 2 to add key 10
2. Then we can add 10

Recursive Initialization

`add(7)`



No more than `log(n)` recursive initializations, but expected depth is constant

1. To add key 7, we must initialize bucket 3 (`7 % 4 = 3`)
2. To initialize bucket 3, we must initialize bucket 1 (`3 % 2 = 1`)
3. Then we can add 7

Summary

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

For further details…

EXTRA SLIDES

Lock-Free List

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

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

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

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

Lock-Free List

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

Insert sentinel with key if not already present and return new list starting with sentinel

Split-Ordered Set

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

How much of array are we using?

Track set size so we know when to resize

Initially use 2 buckets and size is 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;
}
```

Pick a bucket

Non-sentinel split-ordered key

Get bucket's sentinel, initializing if necessary

Call bucket's add() method with reversed key, upon no change we are done

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);
}
```

Find parent, recursively initialize if needed

Prepare key for new sentinel

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