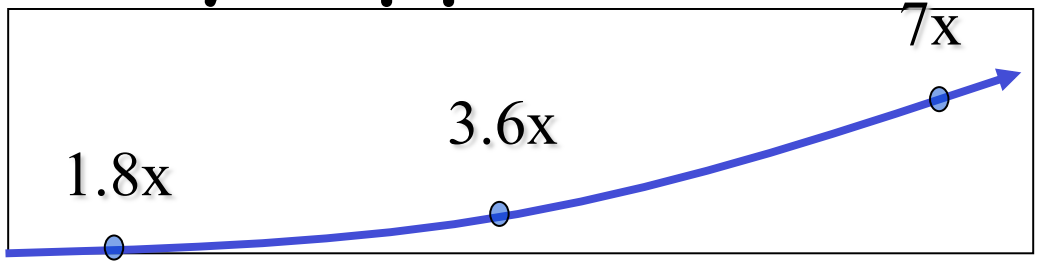
Design and Evaluation of Scalable Software

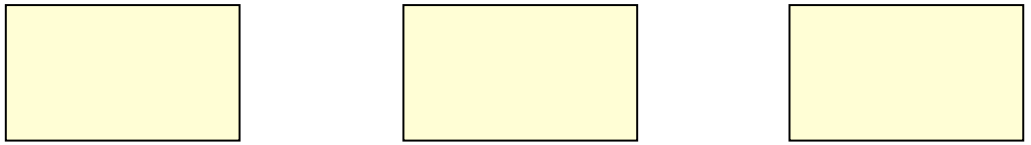
Damian Dechev [1,2]

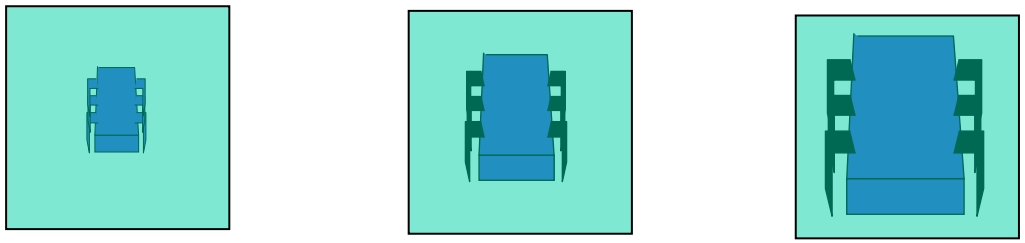
1: University of Central Florida,
CSE S3 Lab, http://cse.eecs.ucf.edu
Orlando, FL

2: Sandia Labs,
Livermore, CA

Traditional Scaling Process (La-Z-Boy Approach)

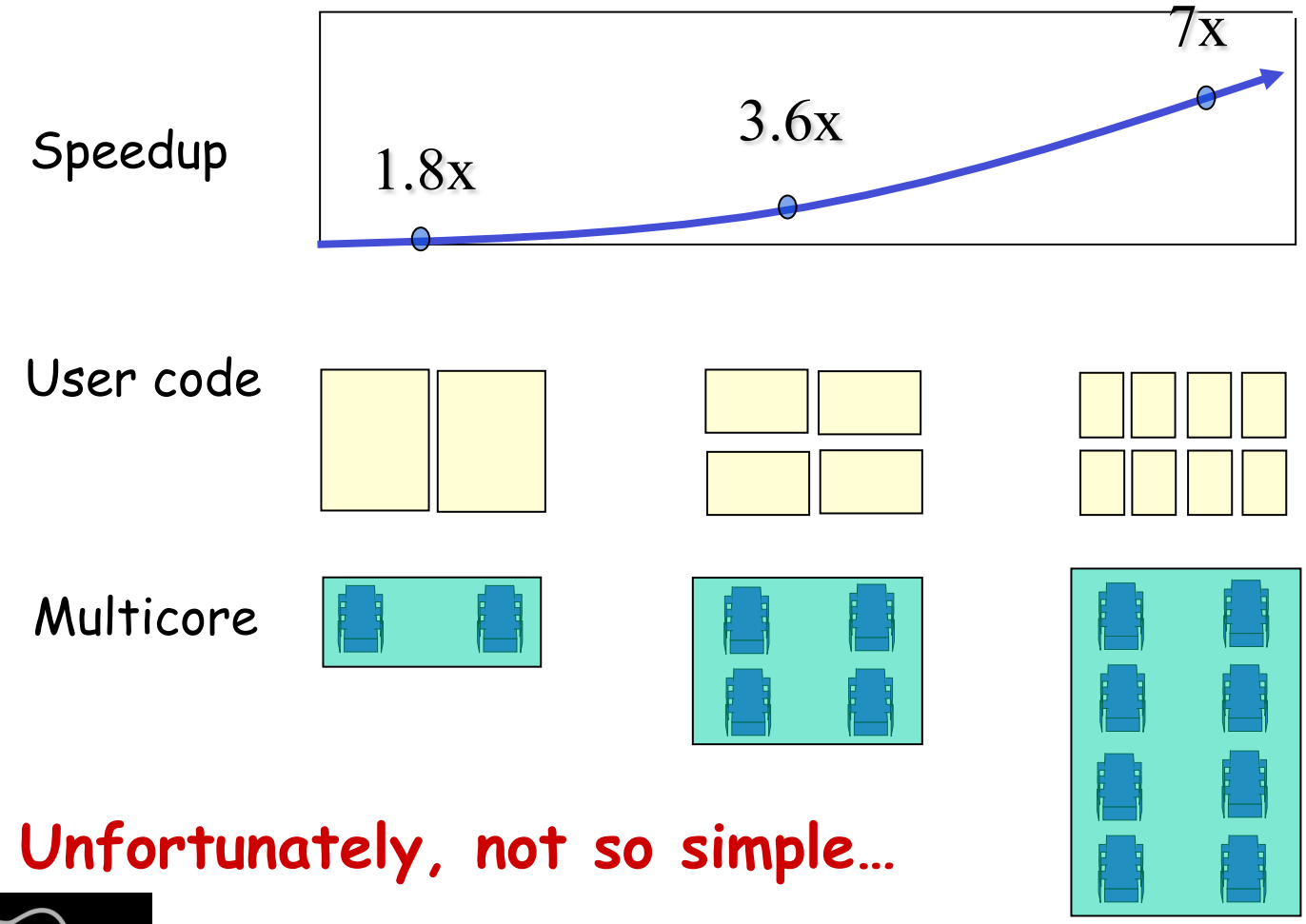






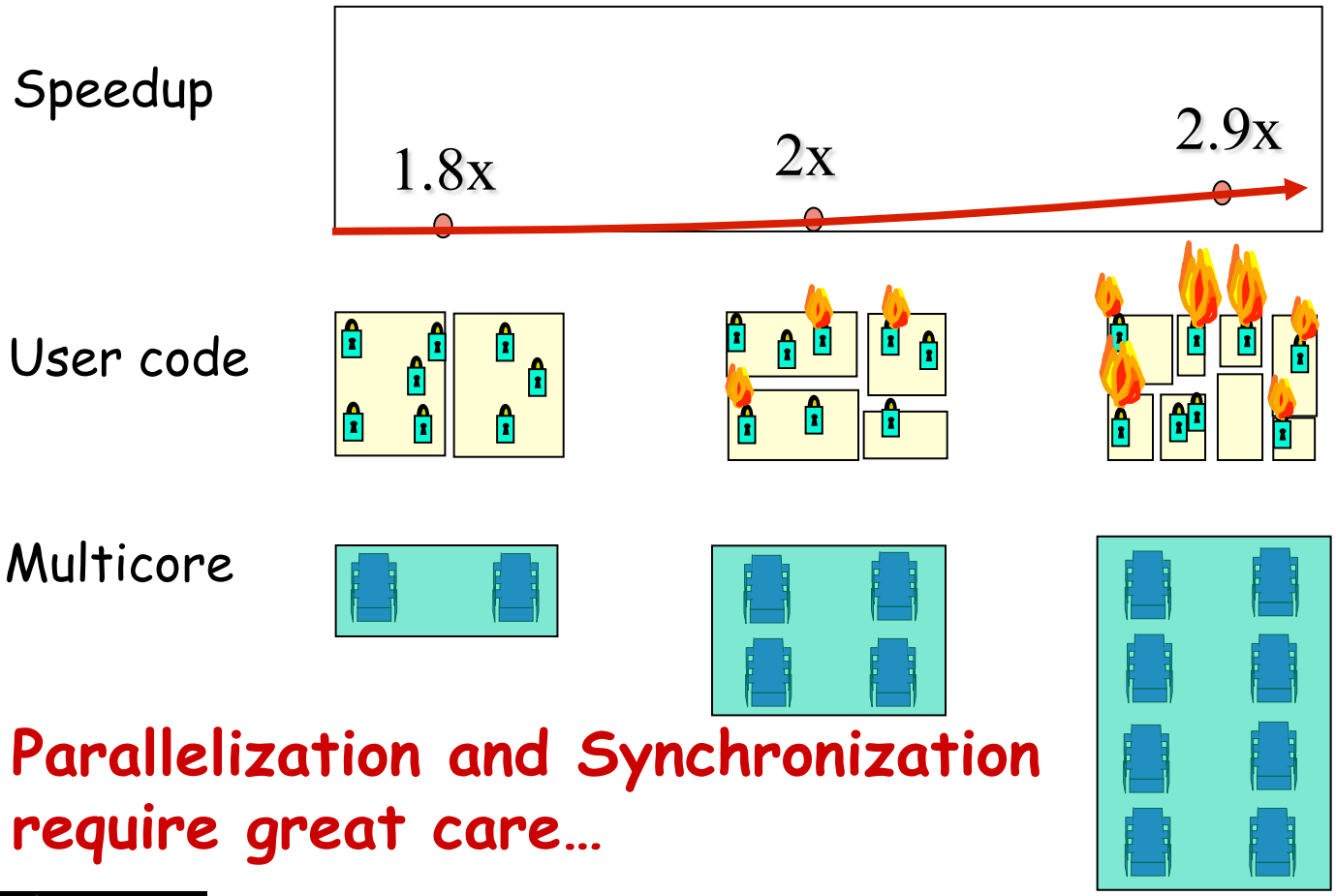
Time: Moore's law

Multicore Scaling Process



Unfortunately, not so simple…

Real-World Scaling Process



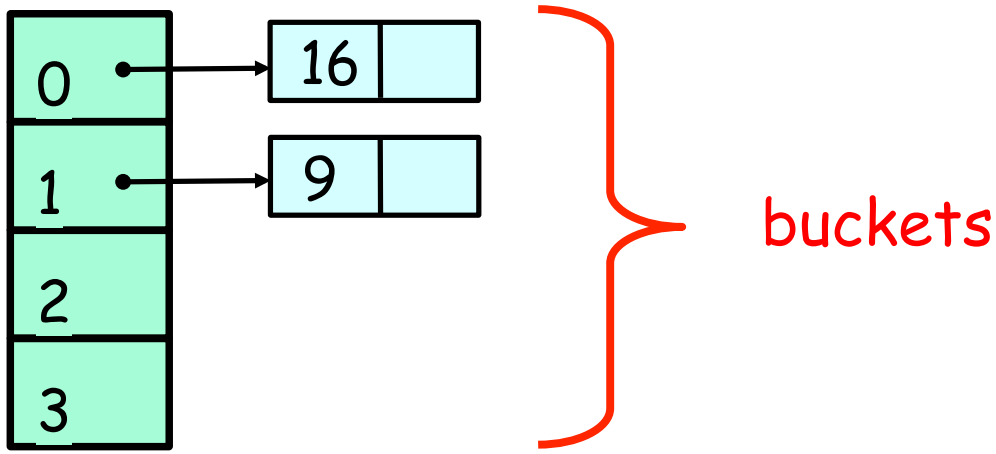
Parallelization and Synchronization require great care…

Problem: Concurrent Hash Table

- Problem is
  - add(), remove(), contains()
  - In a set: take time linear in set size
- We want
  - Constant-time complexity (at least, on average)
  - Hash function (h: items → integers);

  elements uniformly distributed

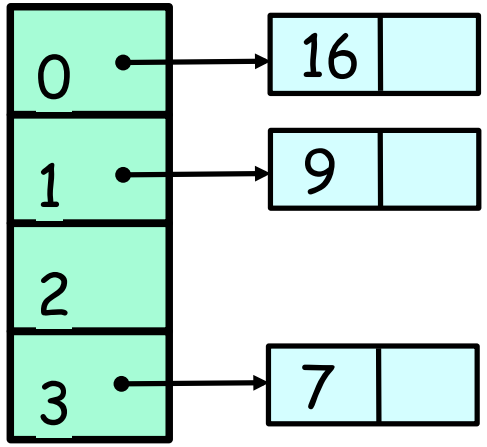
Sequential Hash Map



2 Items

h(k) = k mod 4

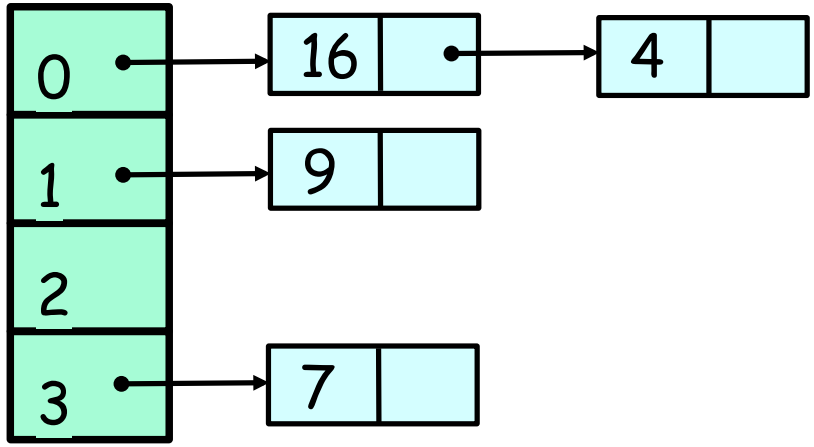
Add an Item



3 Items

h(k) = k mod 4

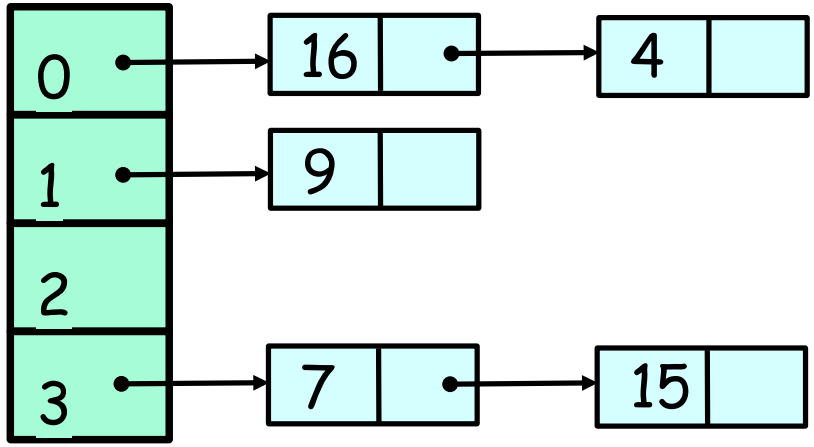
Add Another: Collision



4 Items

h(k) = k mod 4

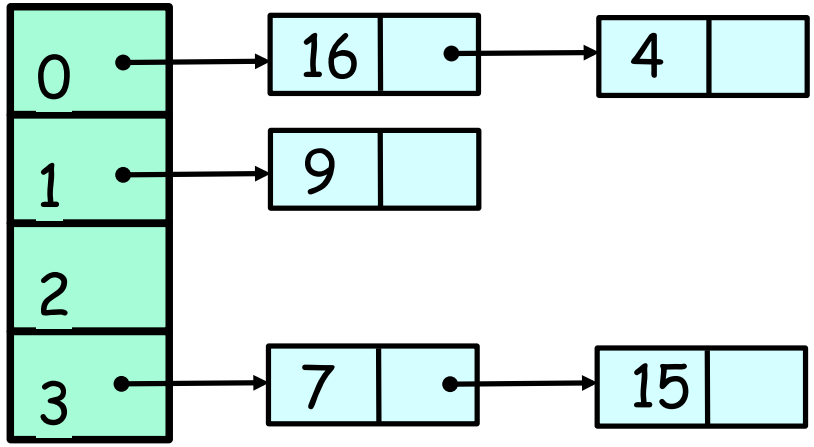
More Collisions



5 Items

h(k) = k mod 4

More Collisions

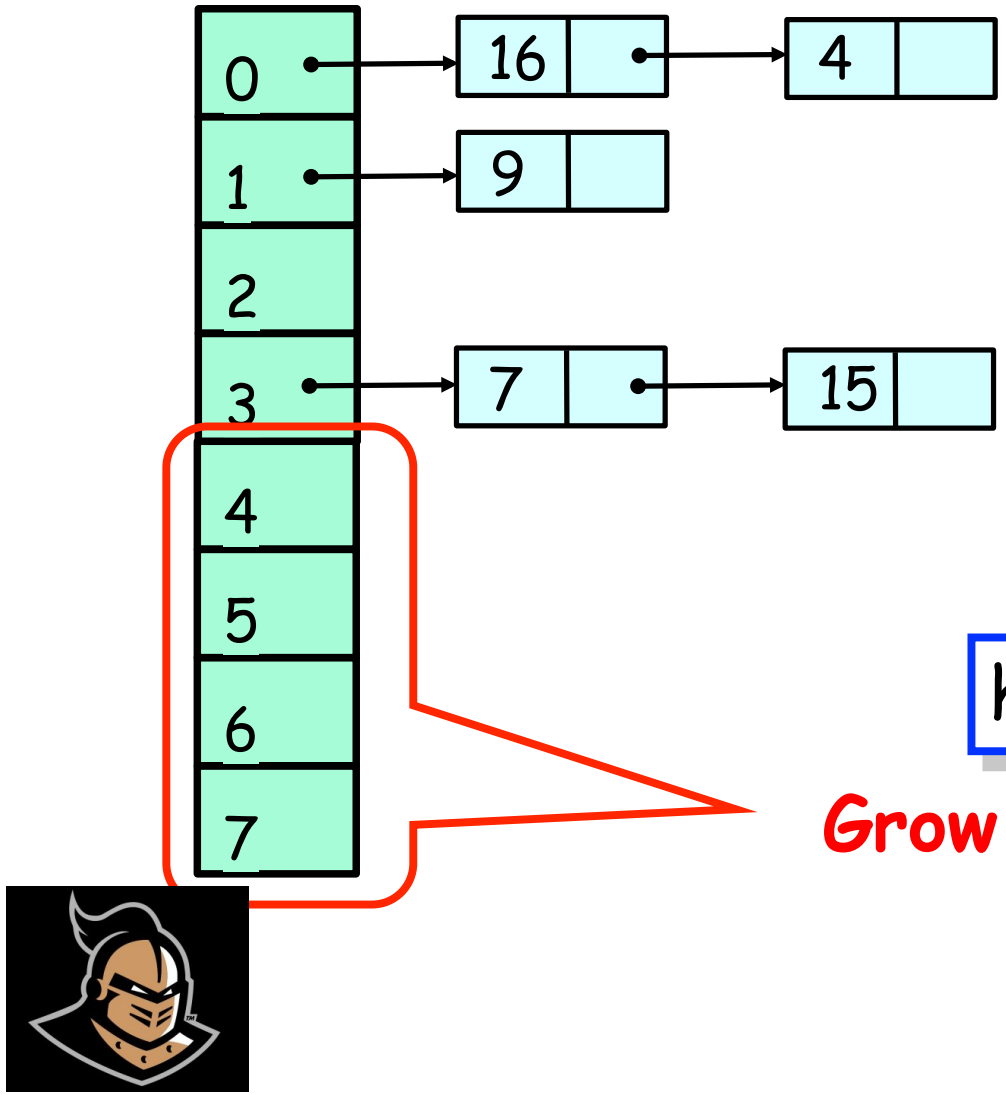


5 Items

h(k) = k mod 4

Problem:
buckets getting too long

Resizing

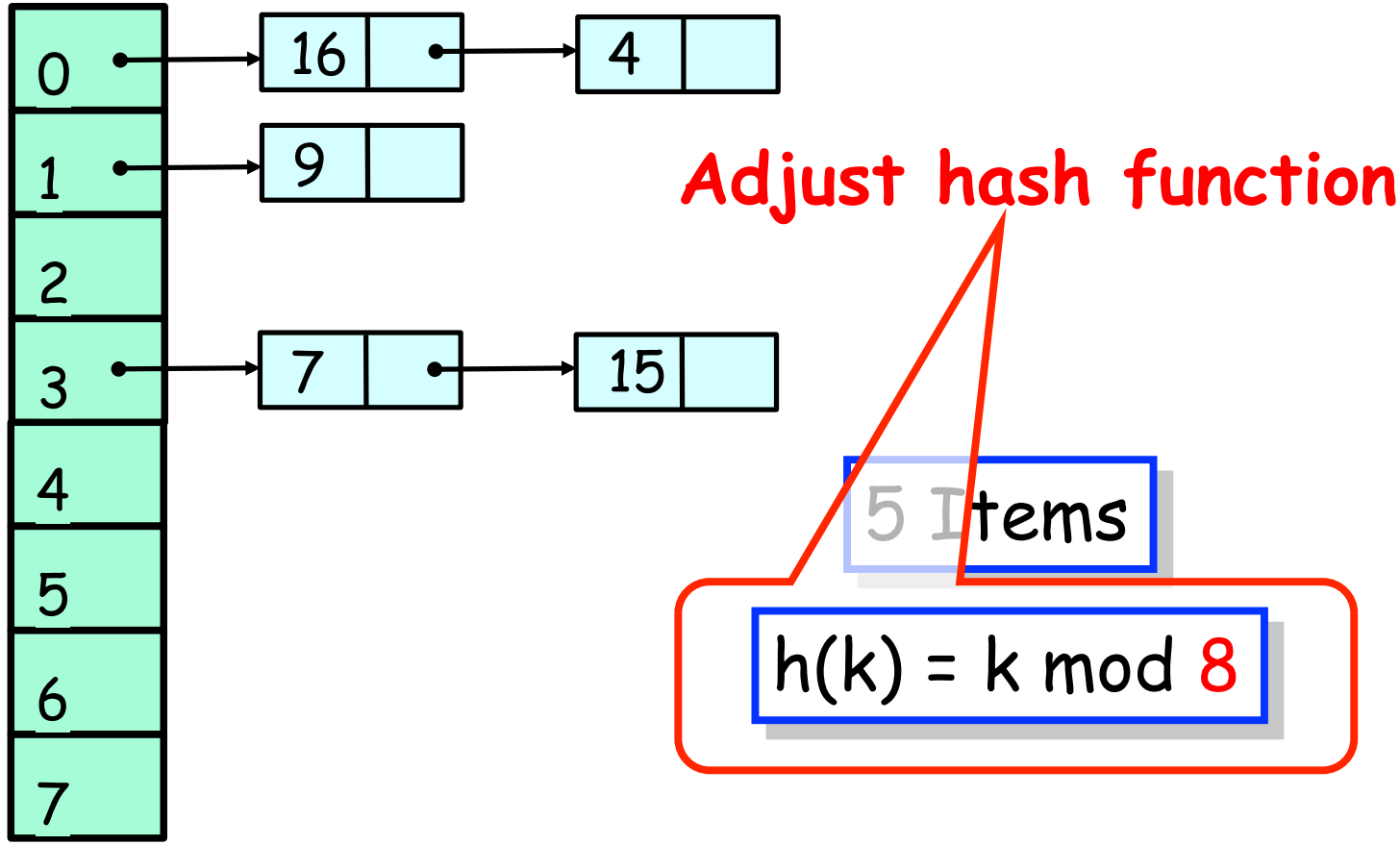


5 Items

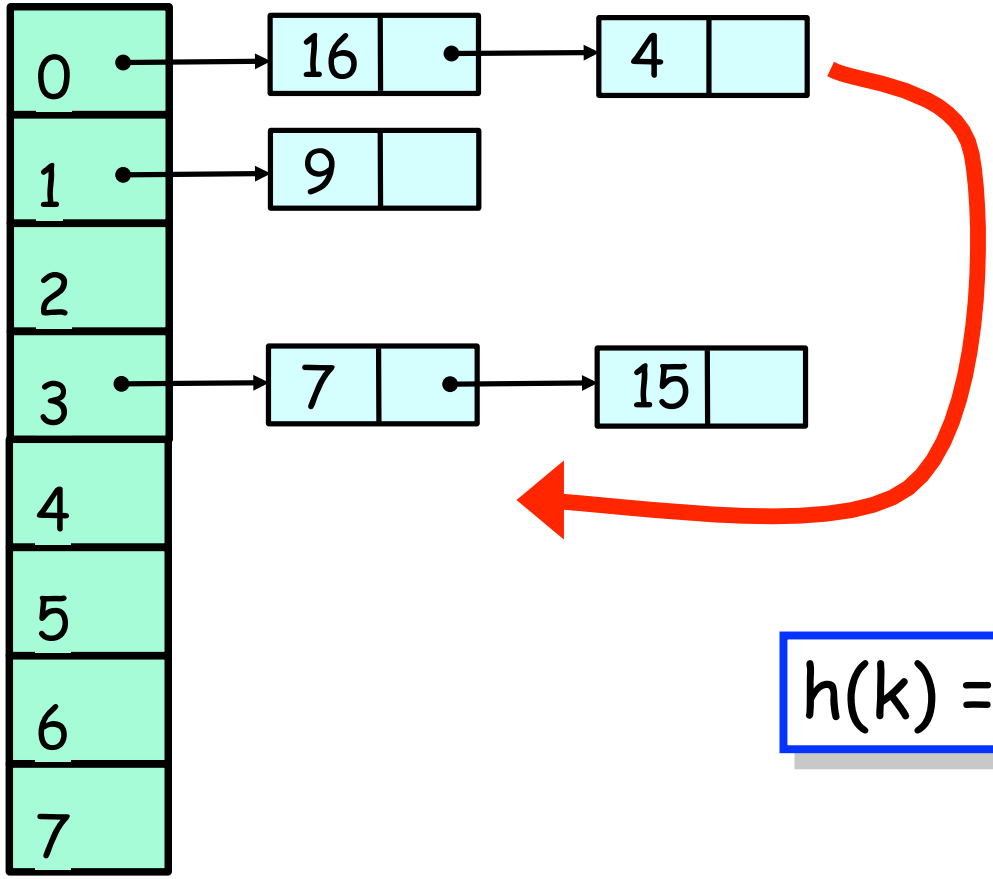
h(k) = k mod 8

Grow the array

Resizing



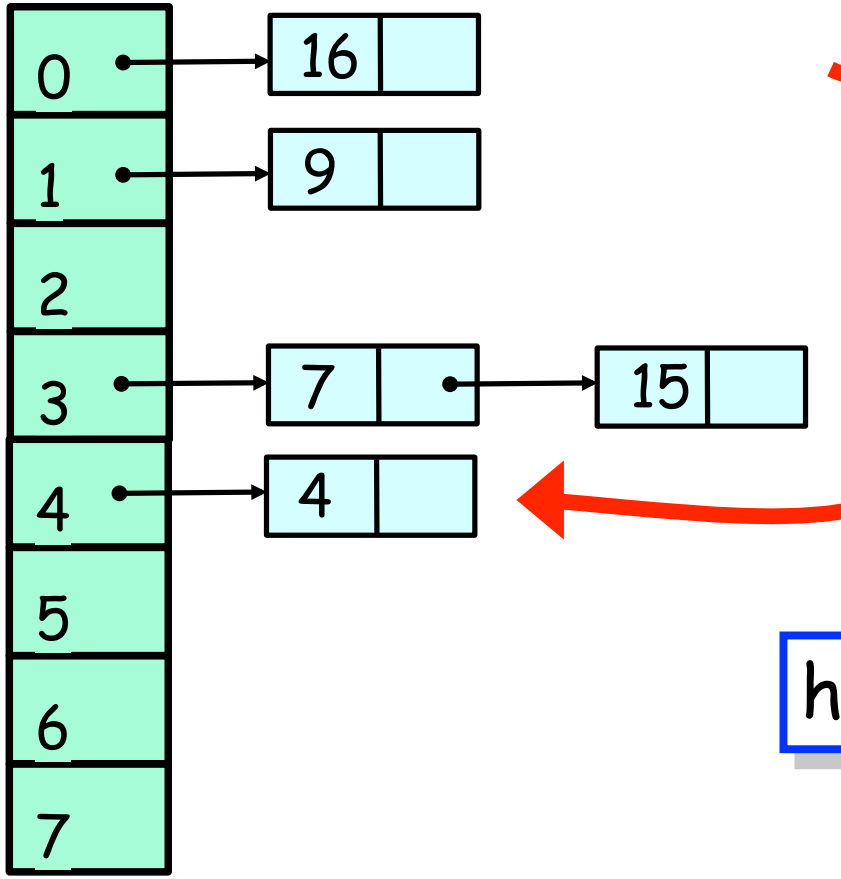
Resizing



h(4) = 0 mod 8

h(k) = k mod 8

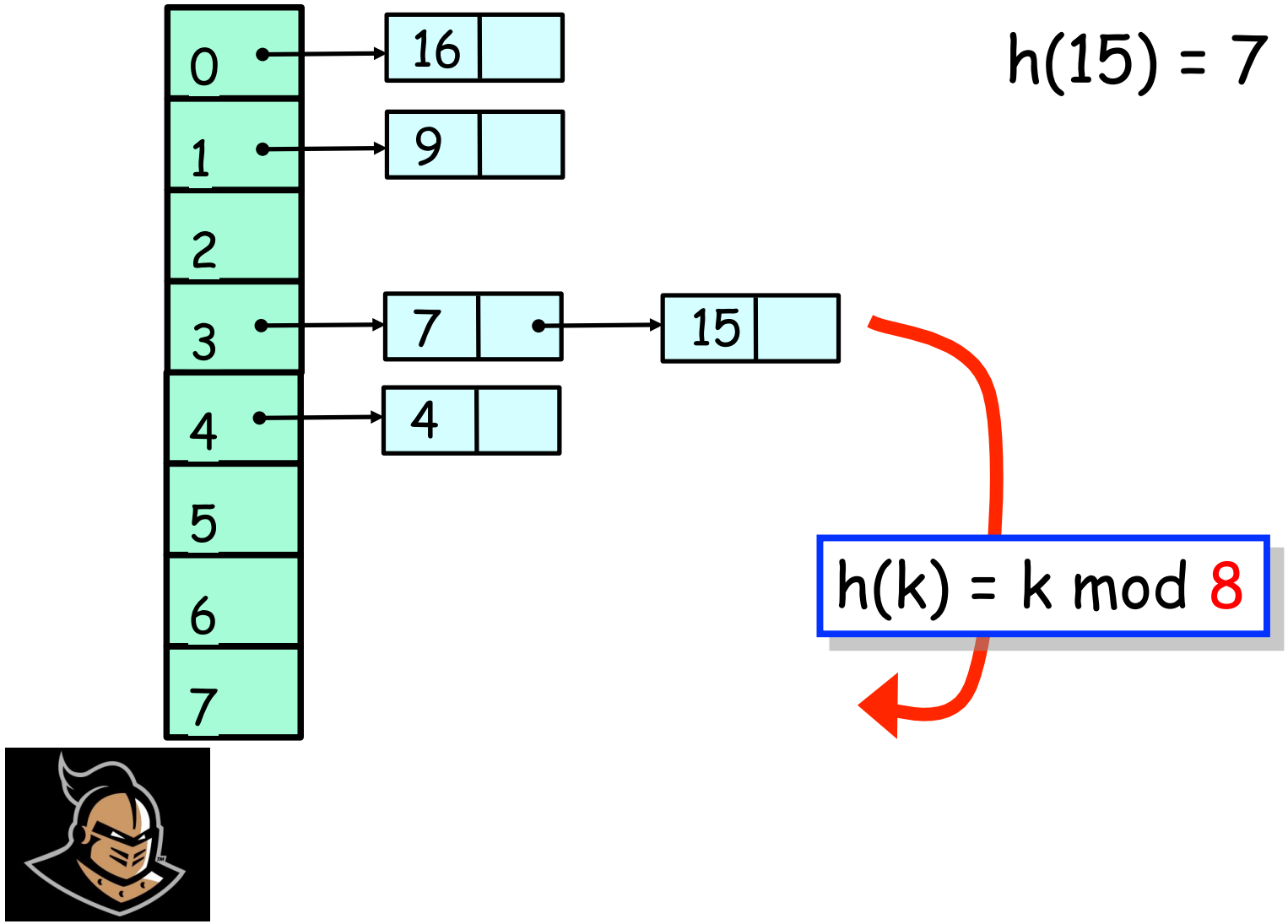
Resizing



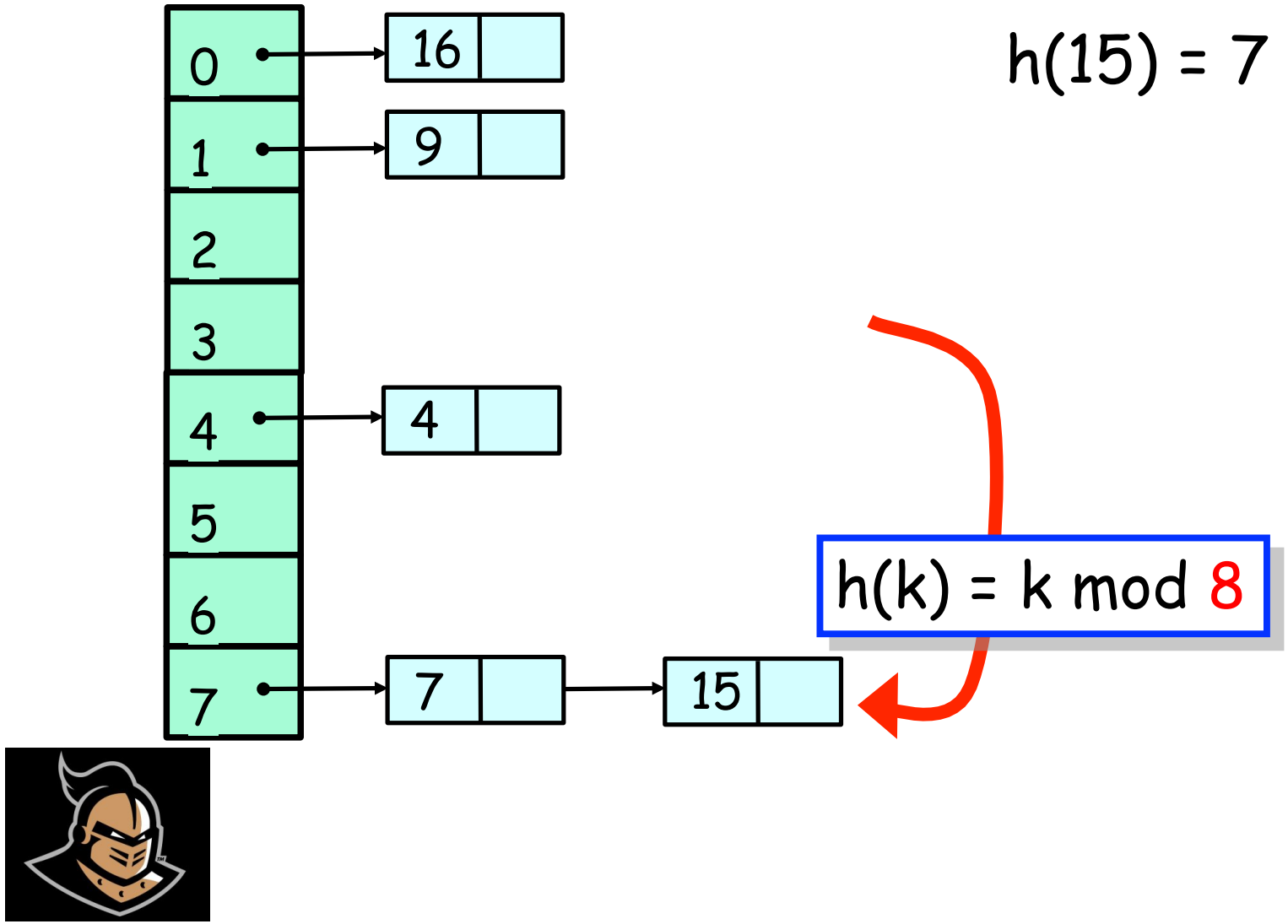
h(4) = 4 mod 8

h(k) = k mod 8

Resizing



Resizing



Simple Hash Set

```
class SimpleHashSet {
  LockFreeList[] table;

  SimpleHashSet(int capacity)
  {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
  }
…
```

Fields

```
class SimpleHashSet {
  LockFreeList[] table;

  SimpleHashSet(int capacity) {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
  }
…
```

Array of lock-free lists

Constructor

```
class SimpleHashSet {
  LockFreeList[] table;

  SimpleHashSet(int capacity) {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
  }
…
```

Initial size

Constructor

```
class SimpleHashSet {
  LockFreeList[] table;

  SimpleHashSet(int capacity) {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
  }
…
```

Allocate memory

Constructor

```
class SimpleHashSet {
  LockFreeList[] table;

  SimpleHashSet(int capacity) {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
  }
…
```

Initialization

Add Function

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

Add Function

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

Use object hash code to pick a bucket

Add Function

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

Call bucket's add() function

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 …

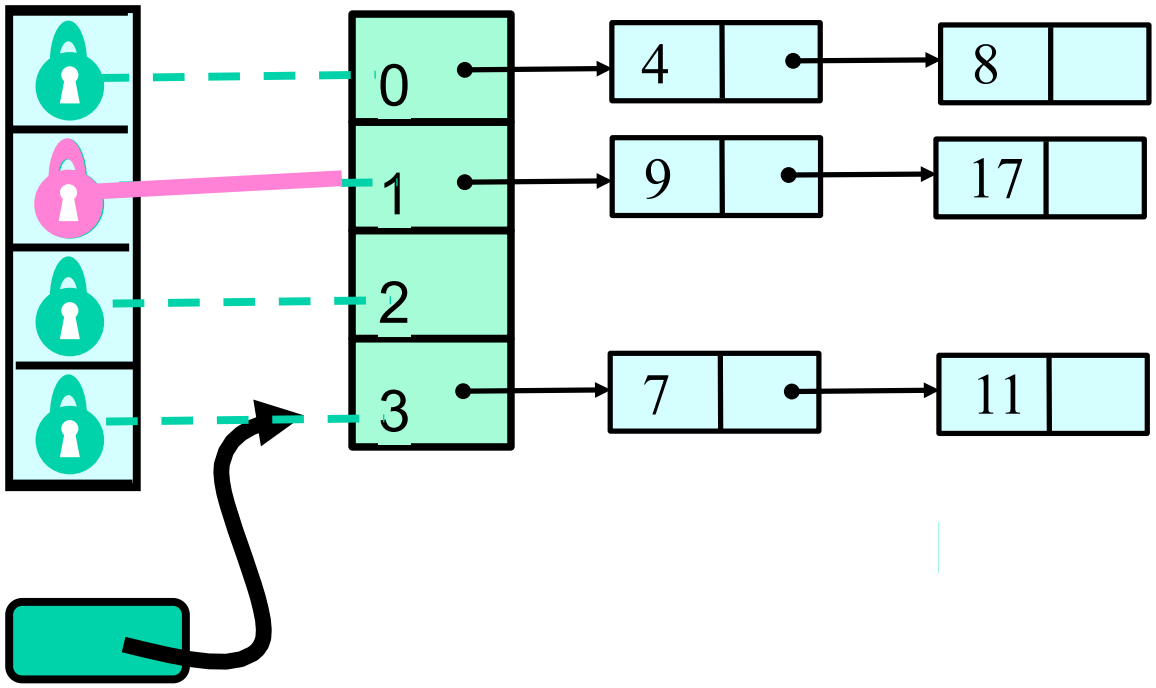
Set Method Mix

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

Coarse-Grained Locking

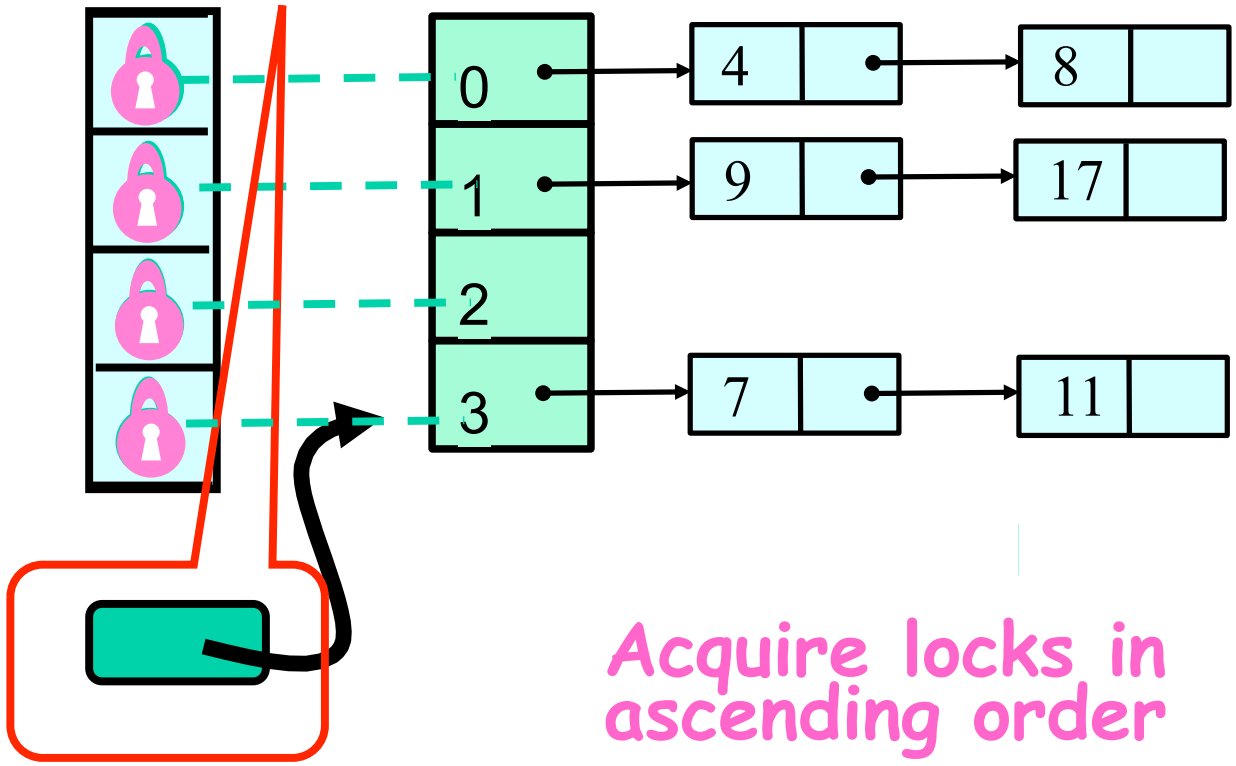
- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck

Fine-Grained Locking



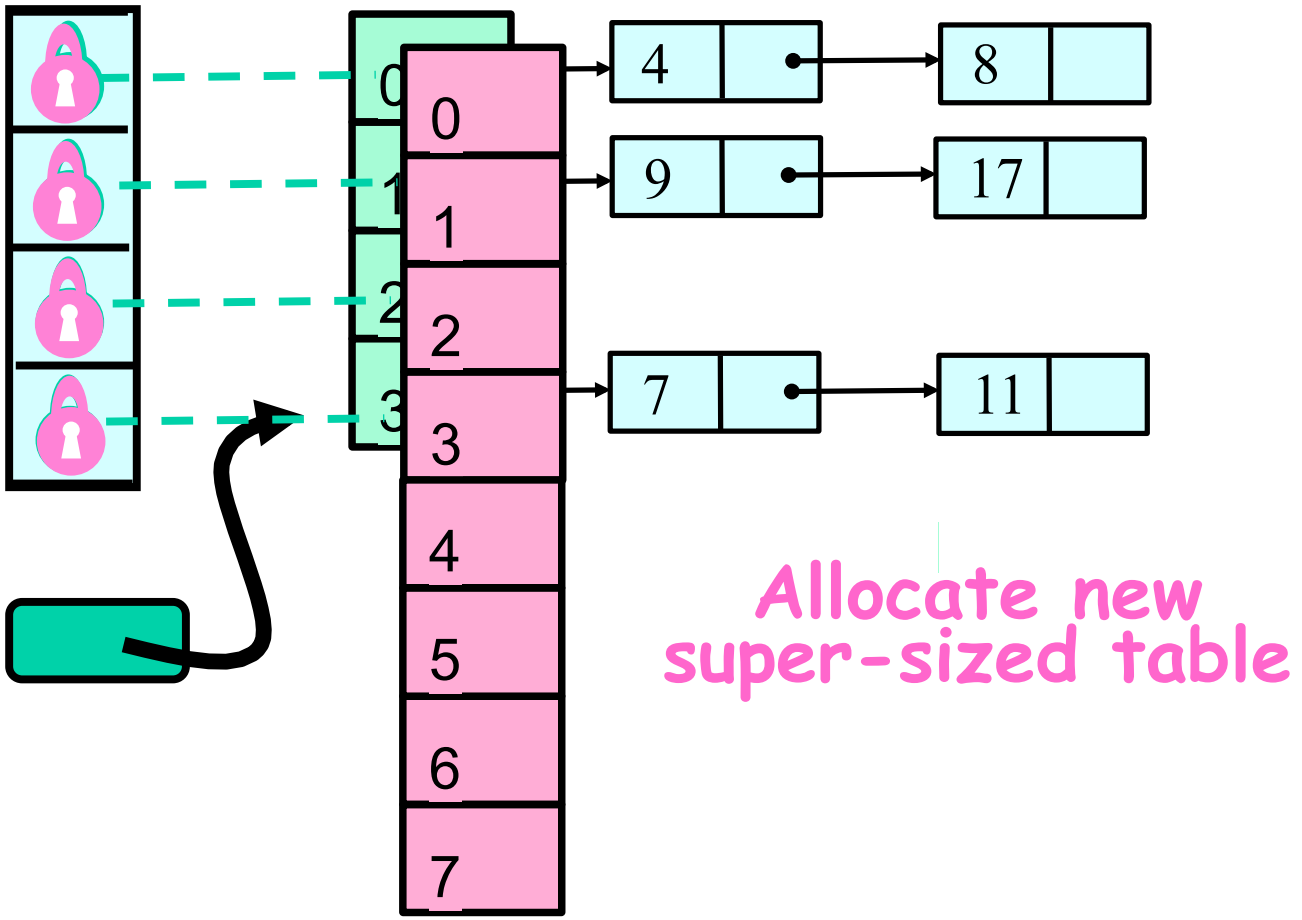
Each lock associated with one bucket

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

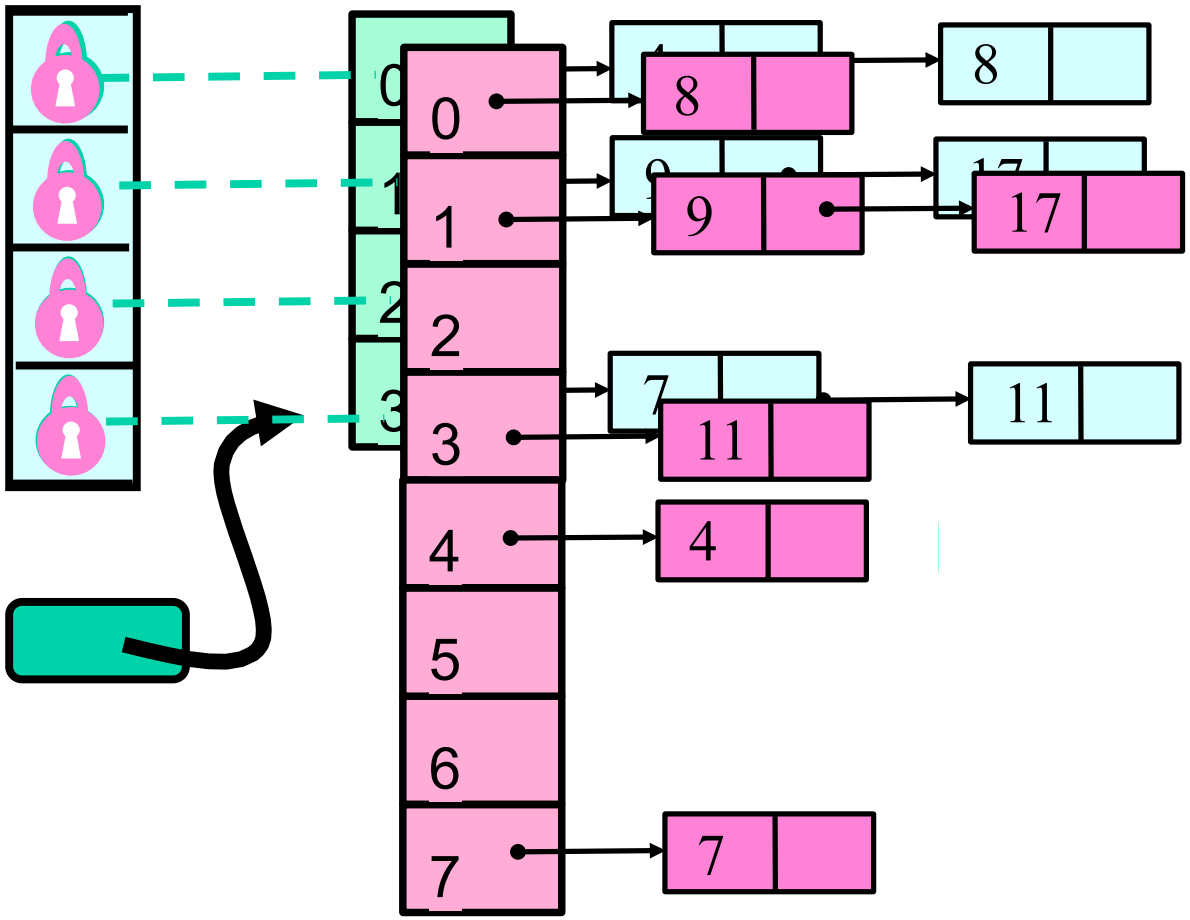


Acquire locks in ascending order

Resize This

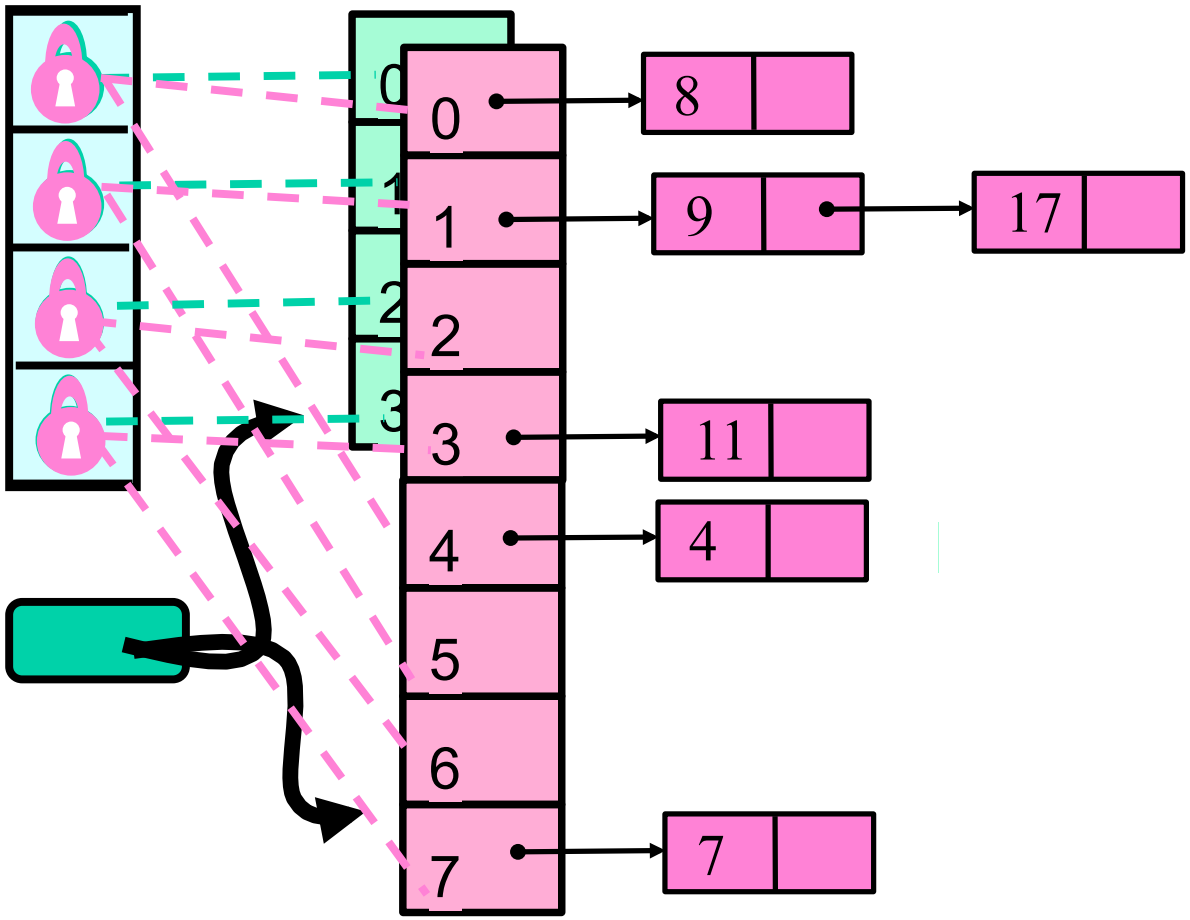


Resize This



Striped Locks: each lock now associated with two buckets

Resize This



Insight

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

FIFO R/W Lock

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

Optimistic Synchronization

- If the contains() function
  - Scans without locking
- If it finds the key
  - OK to return true
- What if it doesn't find the key?

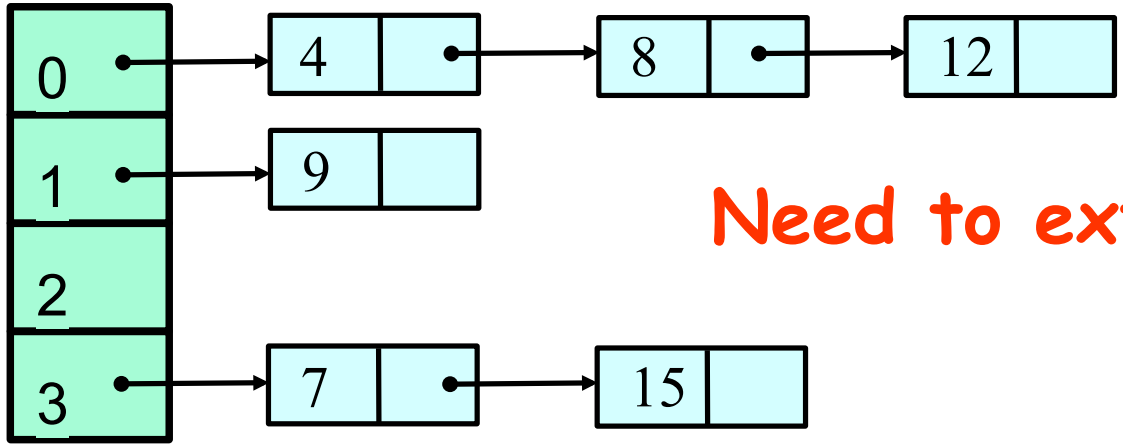
Optimistic Synchronization

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

Stop The World Resizing

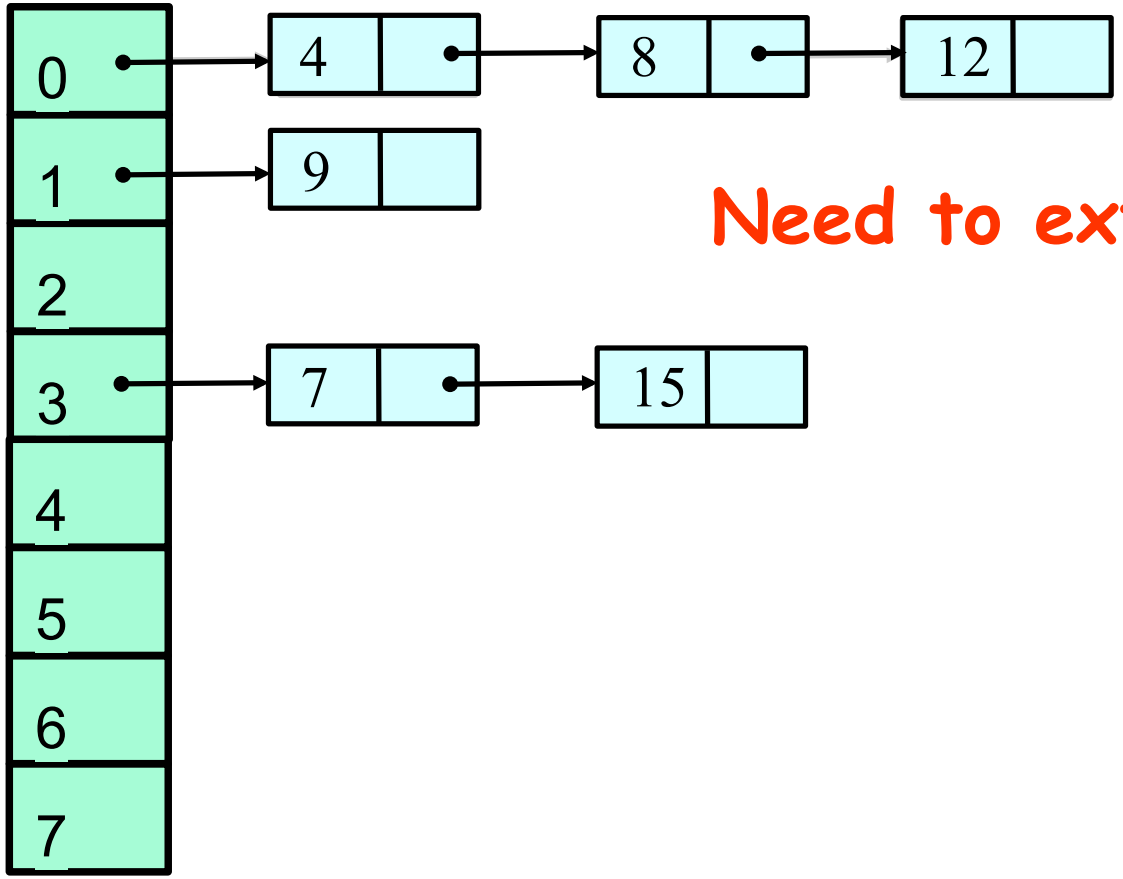
- The resizing we have seen up till now stops all concurrent operations
- Can we design a resize operation that will be incremental
- Need to avoid locking the table
- A lock-free table with incremental resizing

Lock-Free Resizing Problem



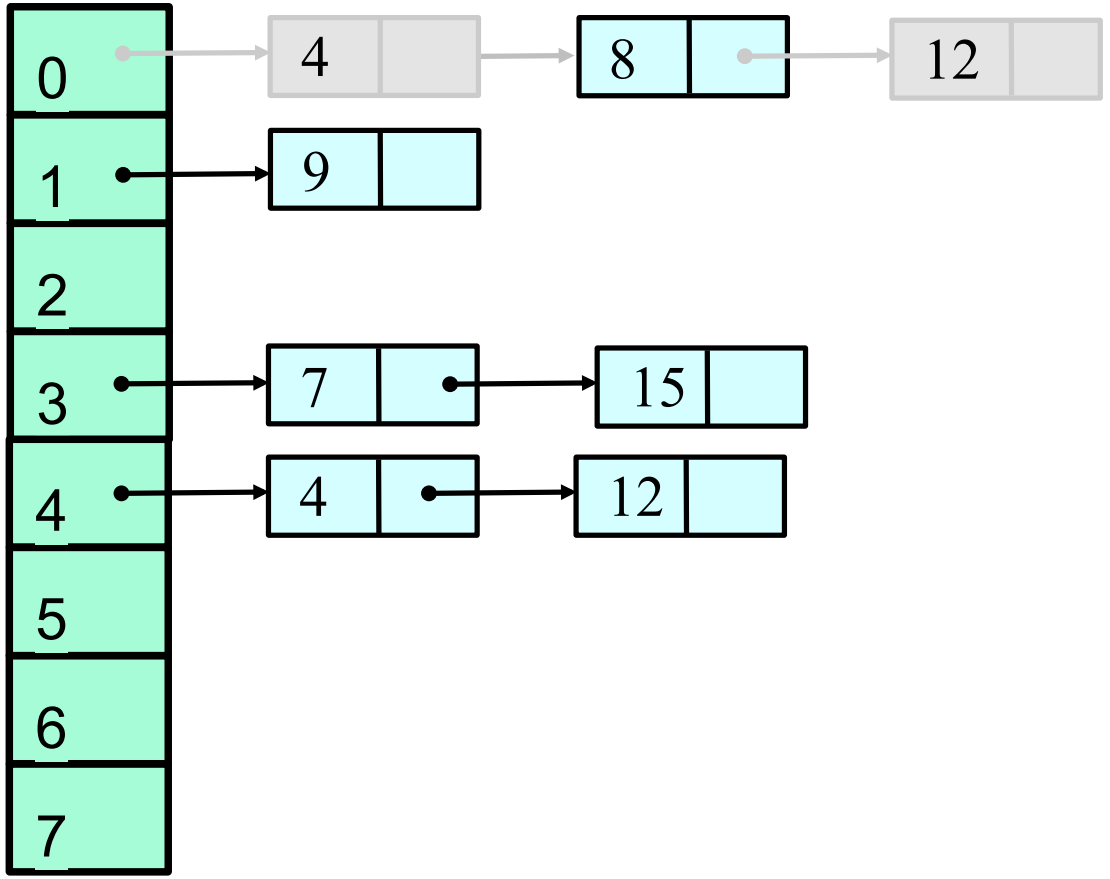
Need to extend table

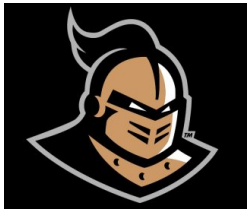
Lock-Free Resizing Problem



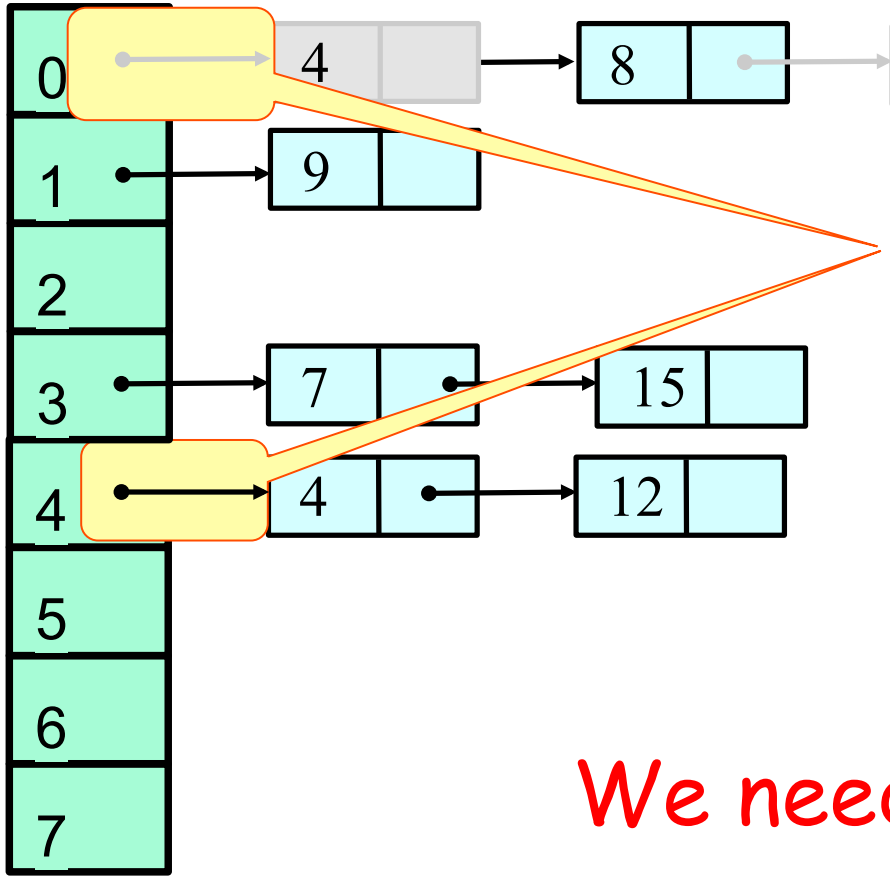
Need to extend table

Lock-Free Resizing Problem





Lock-Free Resizing Problem

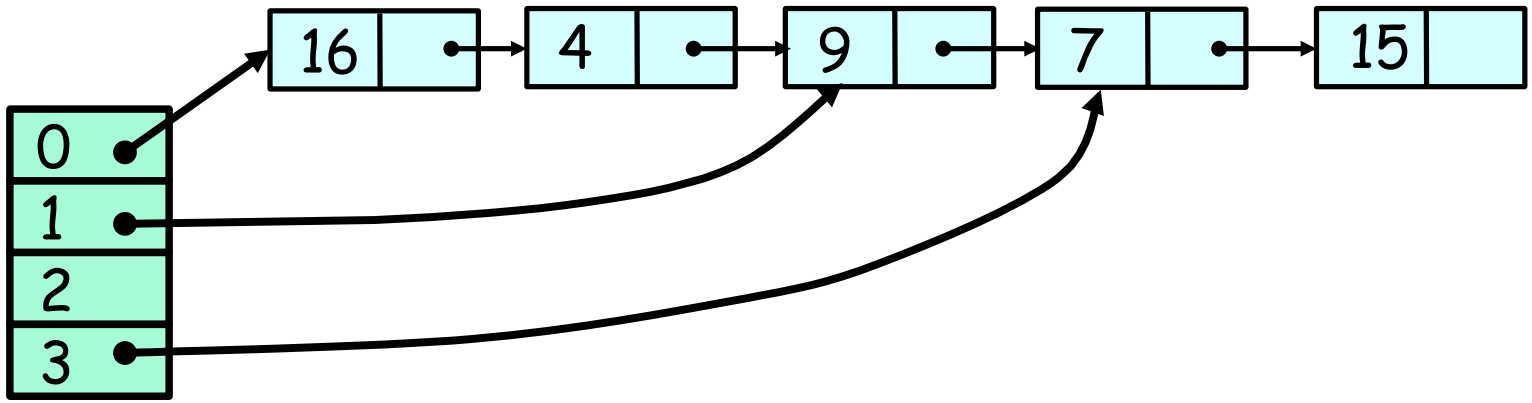


to remove and then add even a single item single location CAS not enough

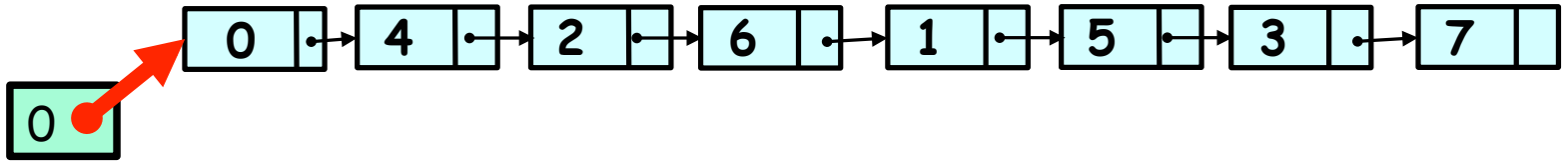
We need a new idea…

Don't move the items

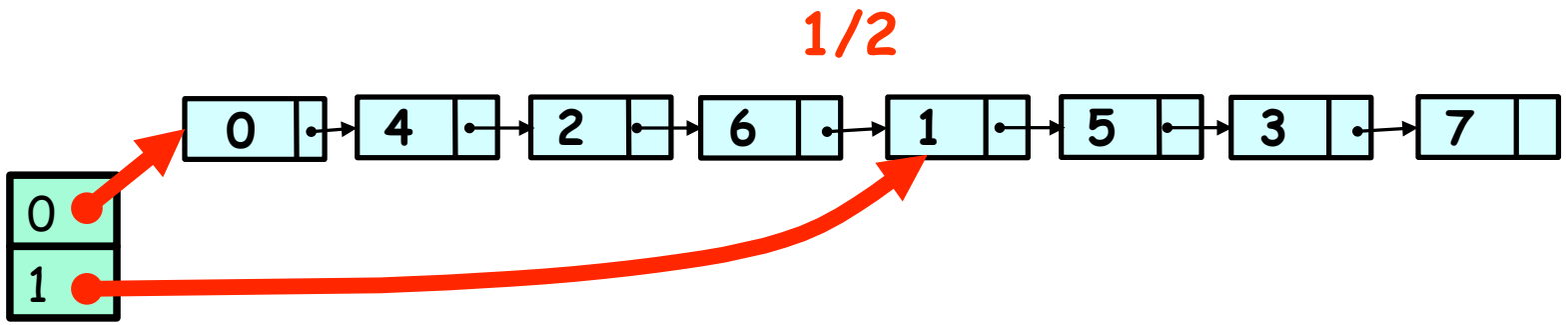
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



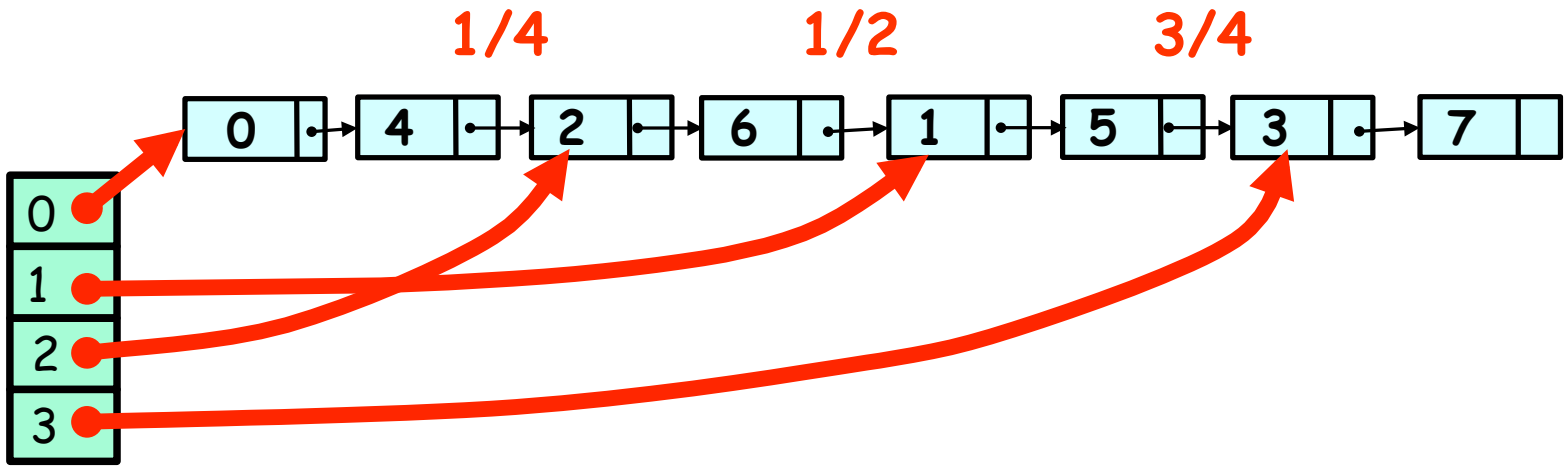
Recursive Split Ordering



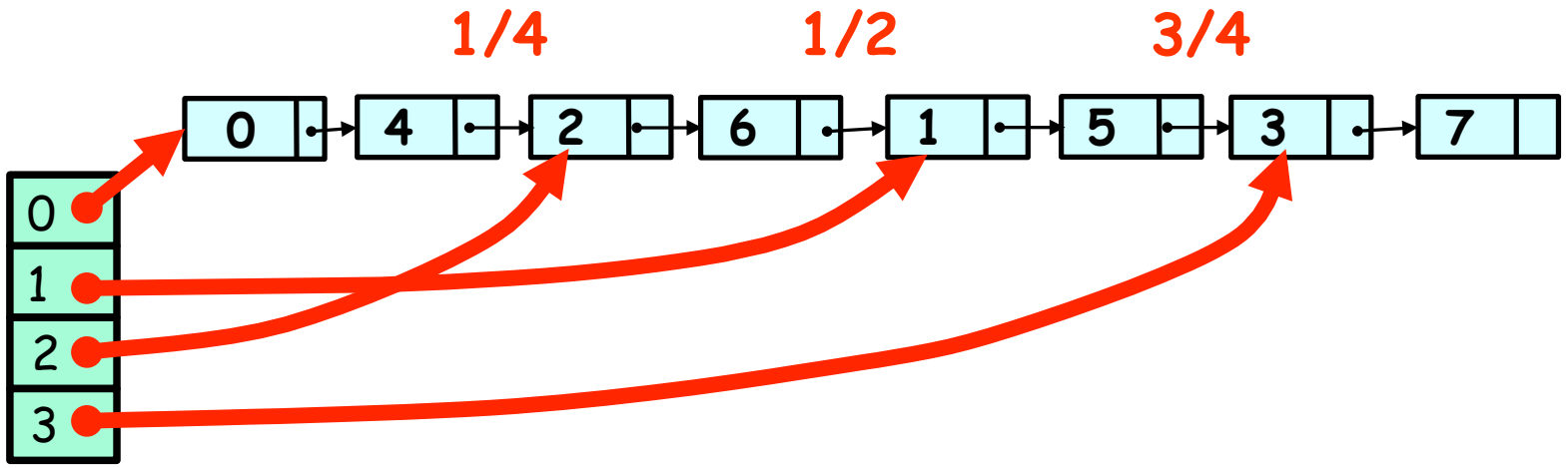
Recursive Split Ordering



Recursive Split Ordering

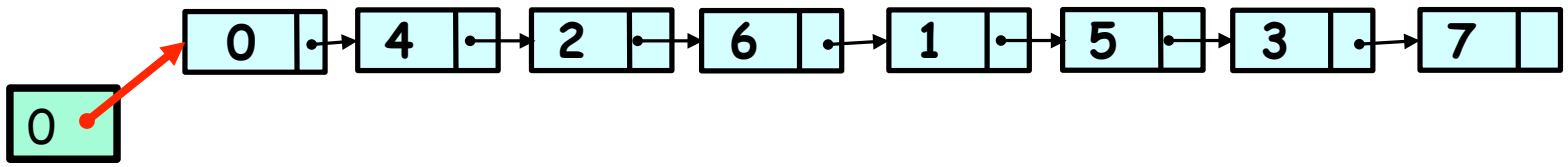


Recursive Split Ordering

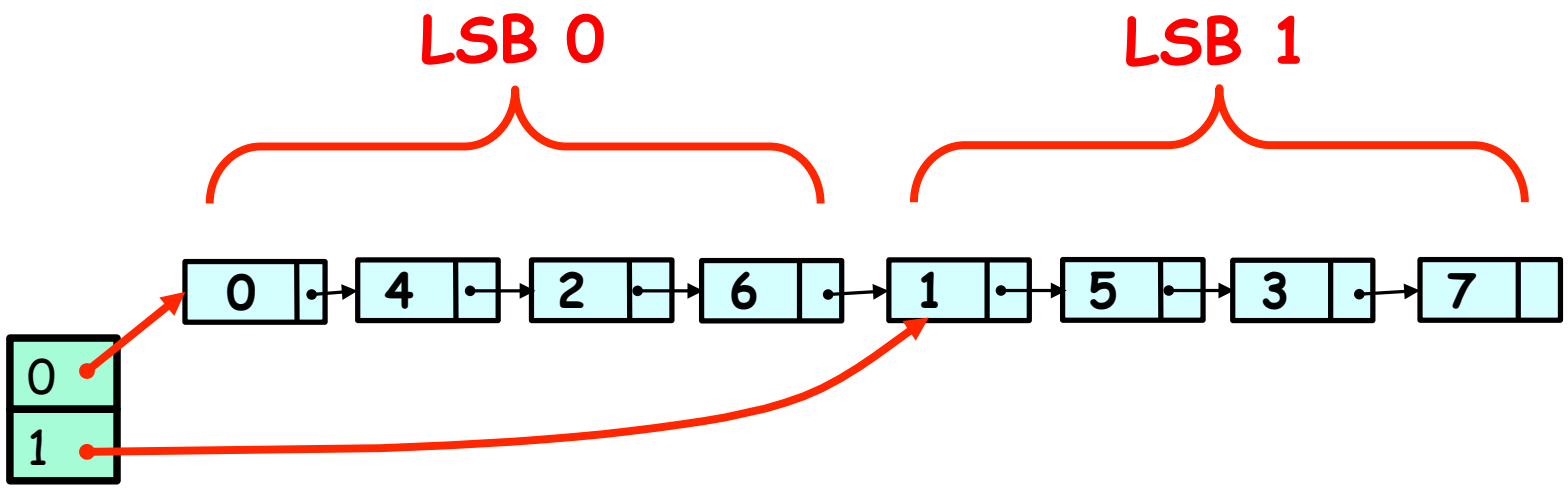


List entries sorted in order that allows recursive splitting. How?

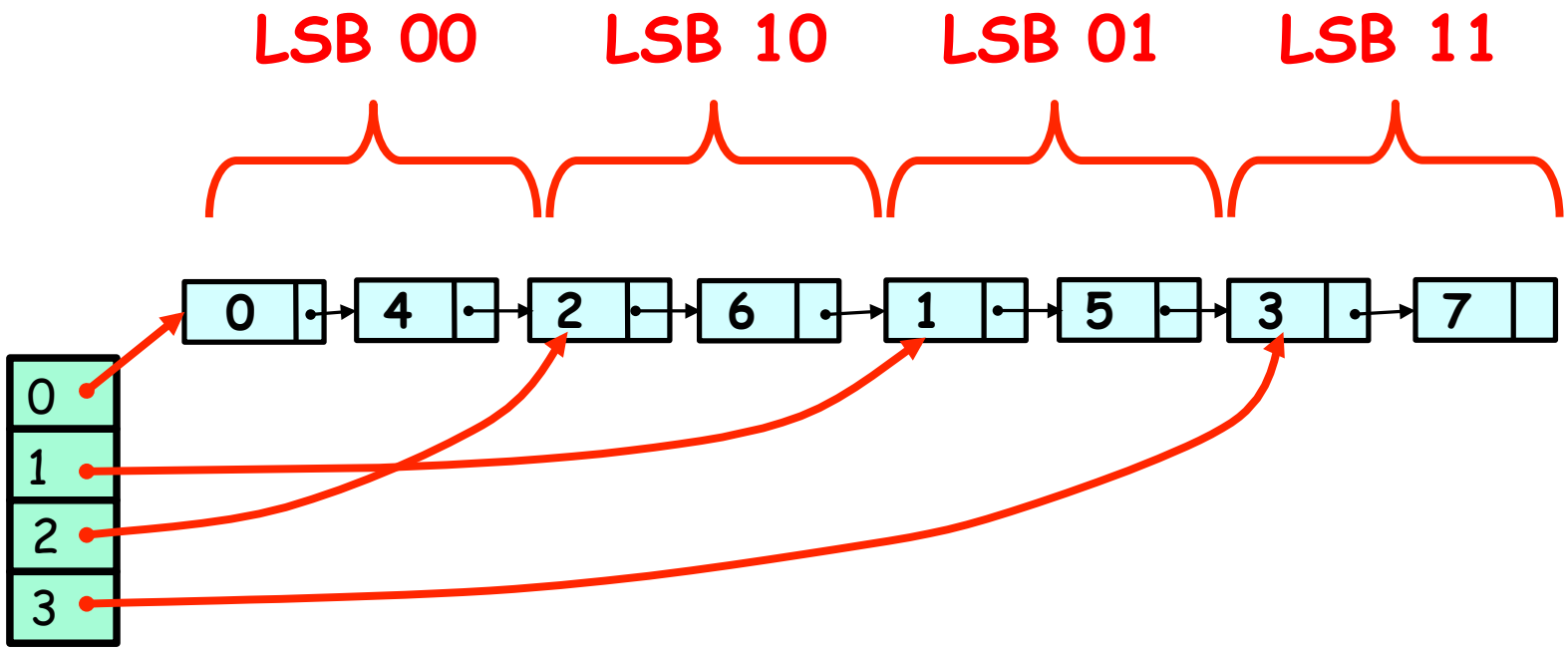
Recursive Split Ordering



Recursive Split Ordering



Recursive Split Ordering



Split-Order

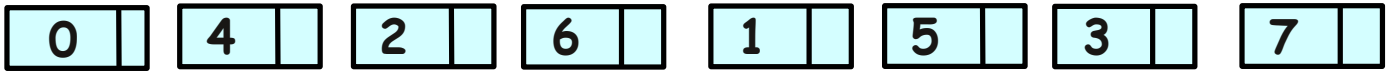
- If the table size is 2^i,
  - Bucket b contains keys k
    - $k = b \pmod{2^i}$
  - bucket index consists of key's i LSBs

When Table Splits

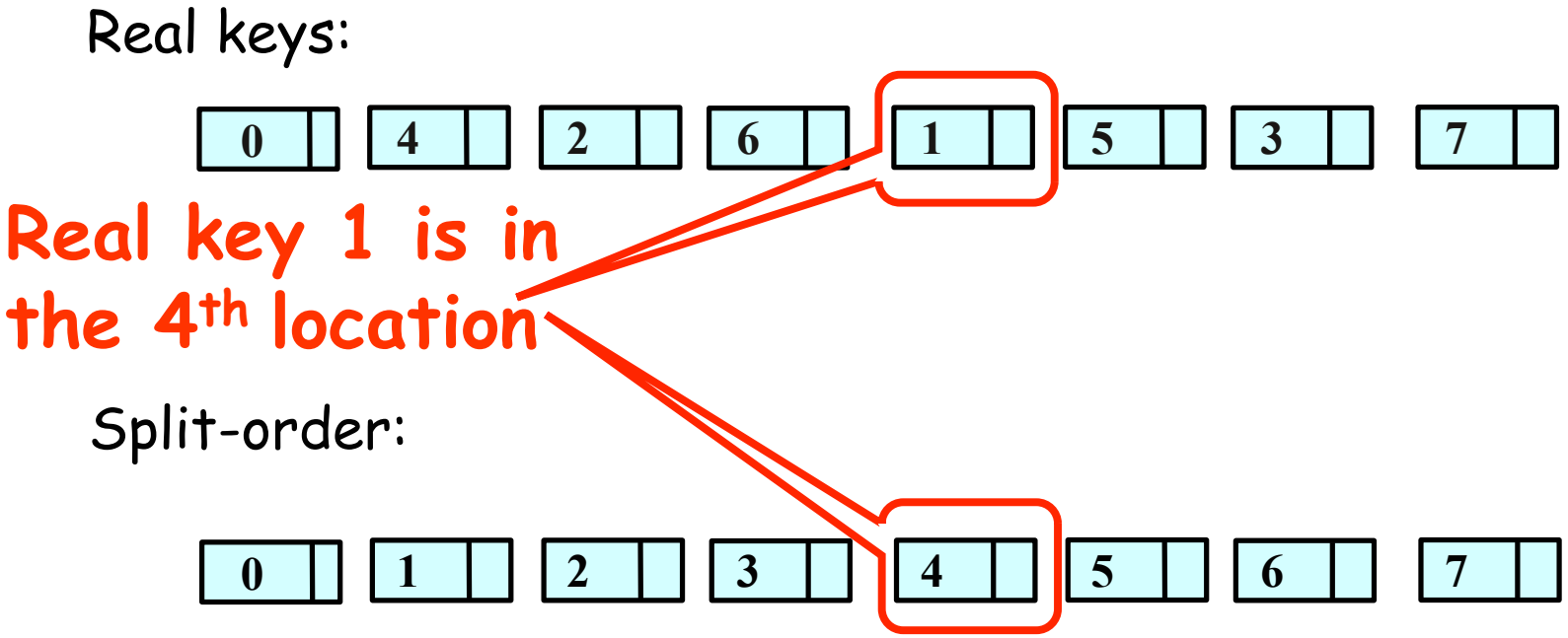
- Some keys stay
  - $b = k \bmod (2^{i+1})$
- Some move
  - $b+2^i = k \bmod (2^{i+1})$
- Determined by $(i+1)^{st}$ bit
  - Counting backwards

A Bit of Magic

Real keys:

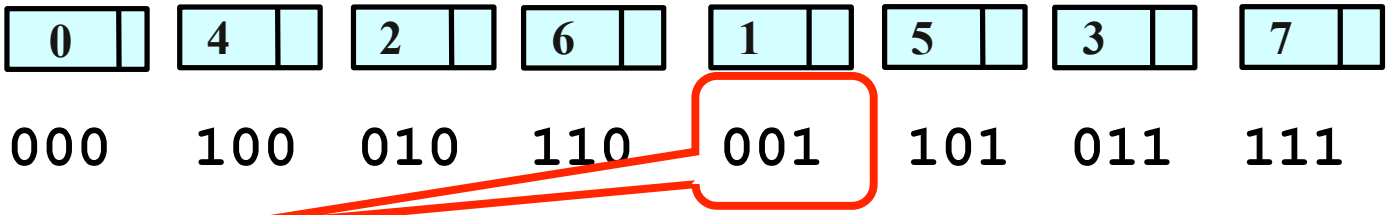


A Bit of Magic



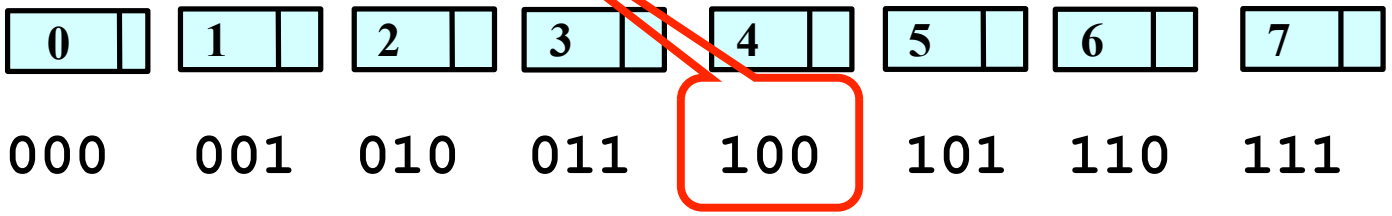
A Bit of Magic

Real keys:



Real key 1 is in 4th location

Split-order:

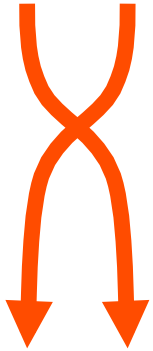


A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

Split-order:

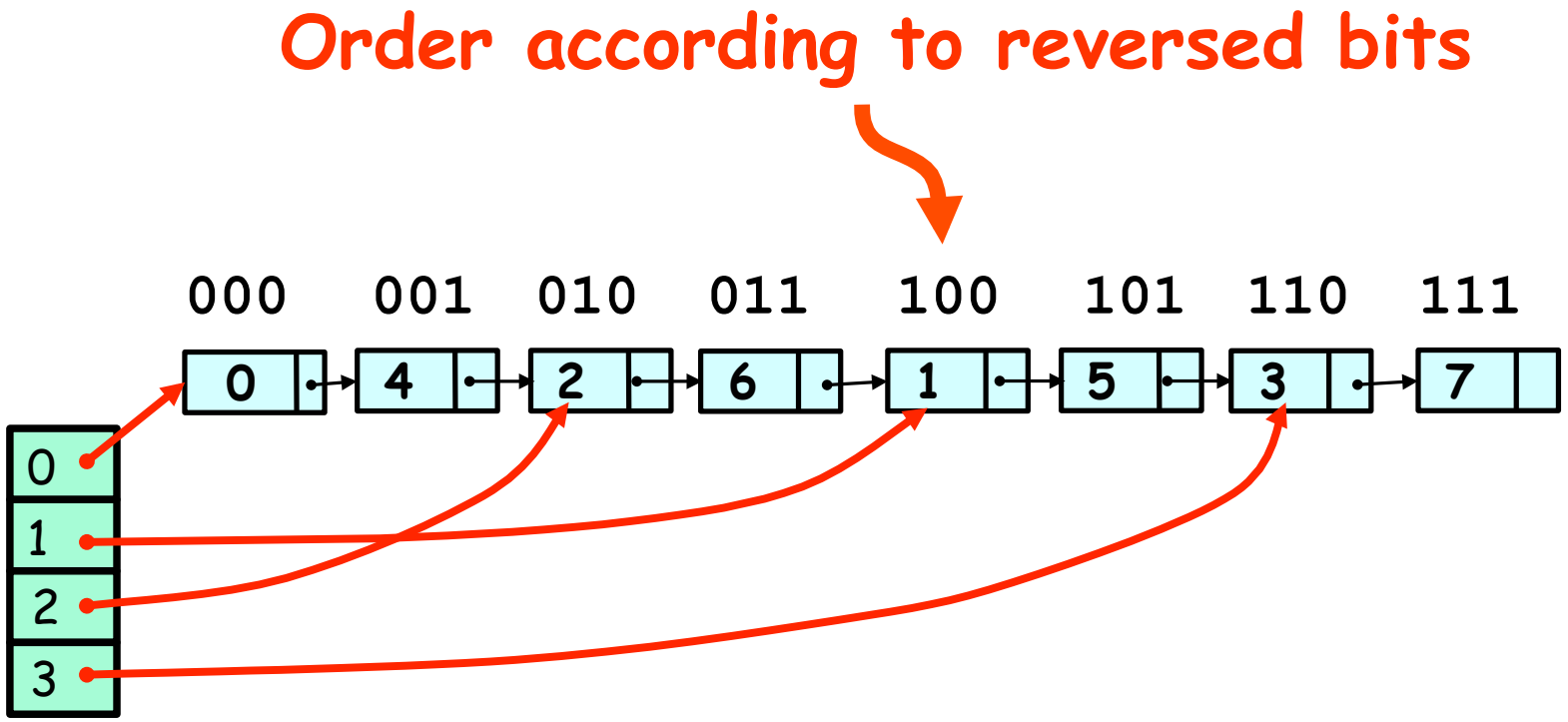


000 001 010 011 100 101 110 111

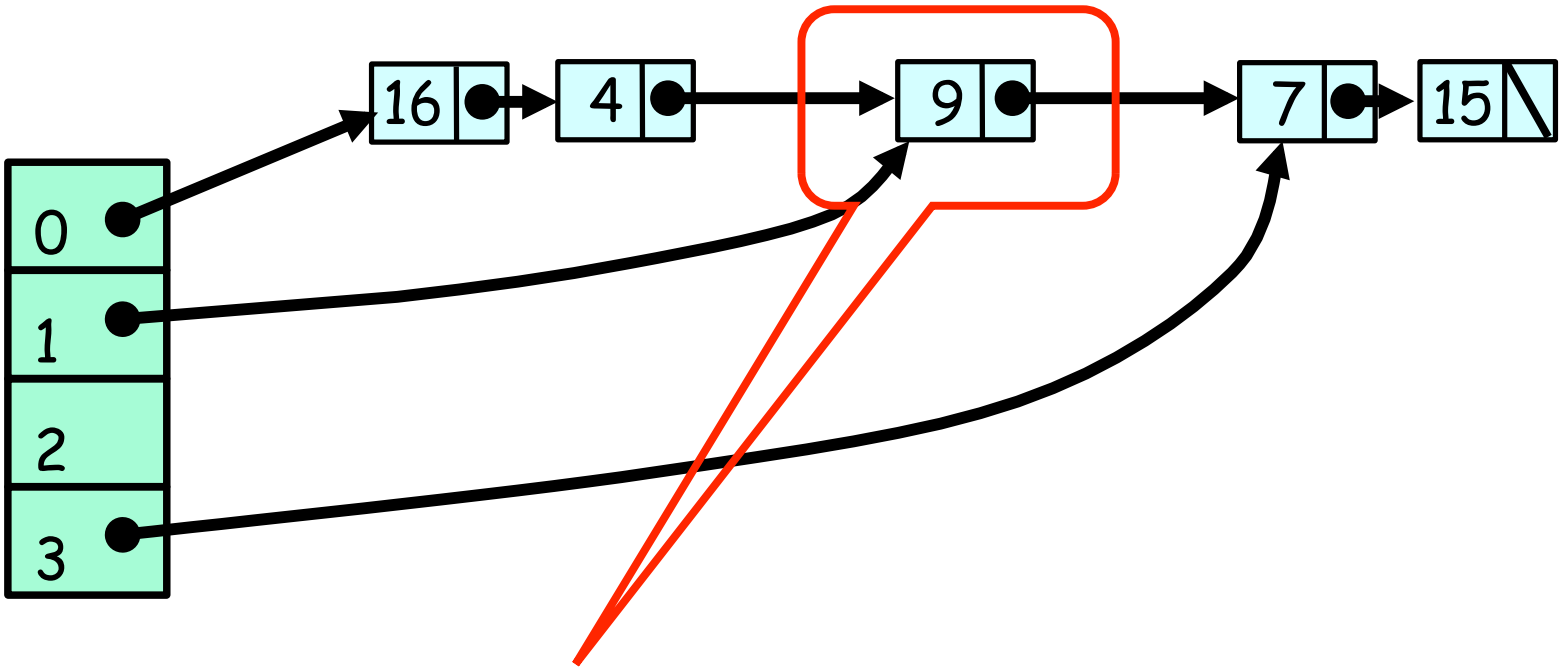
Just reverse the order of the key bits

Split Ordered Hashing

Order according to reversed bits

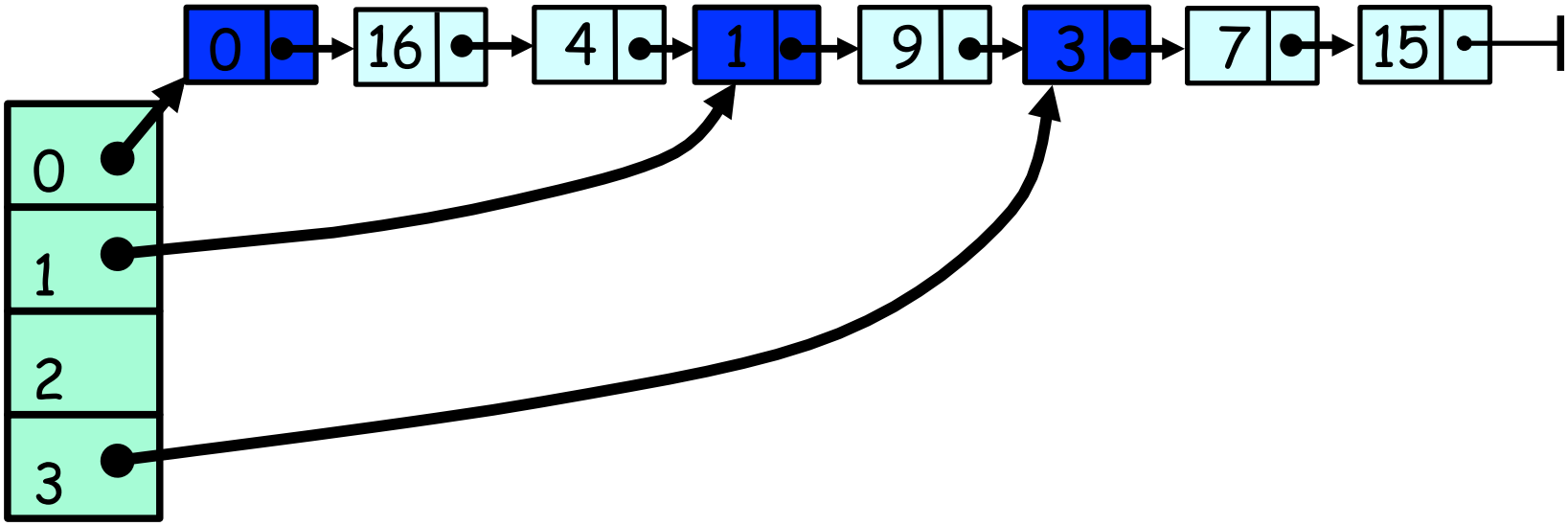


Sentinel Nodes



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

Sentinel Nodes

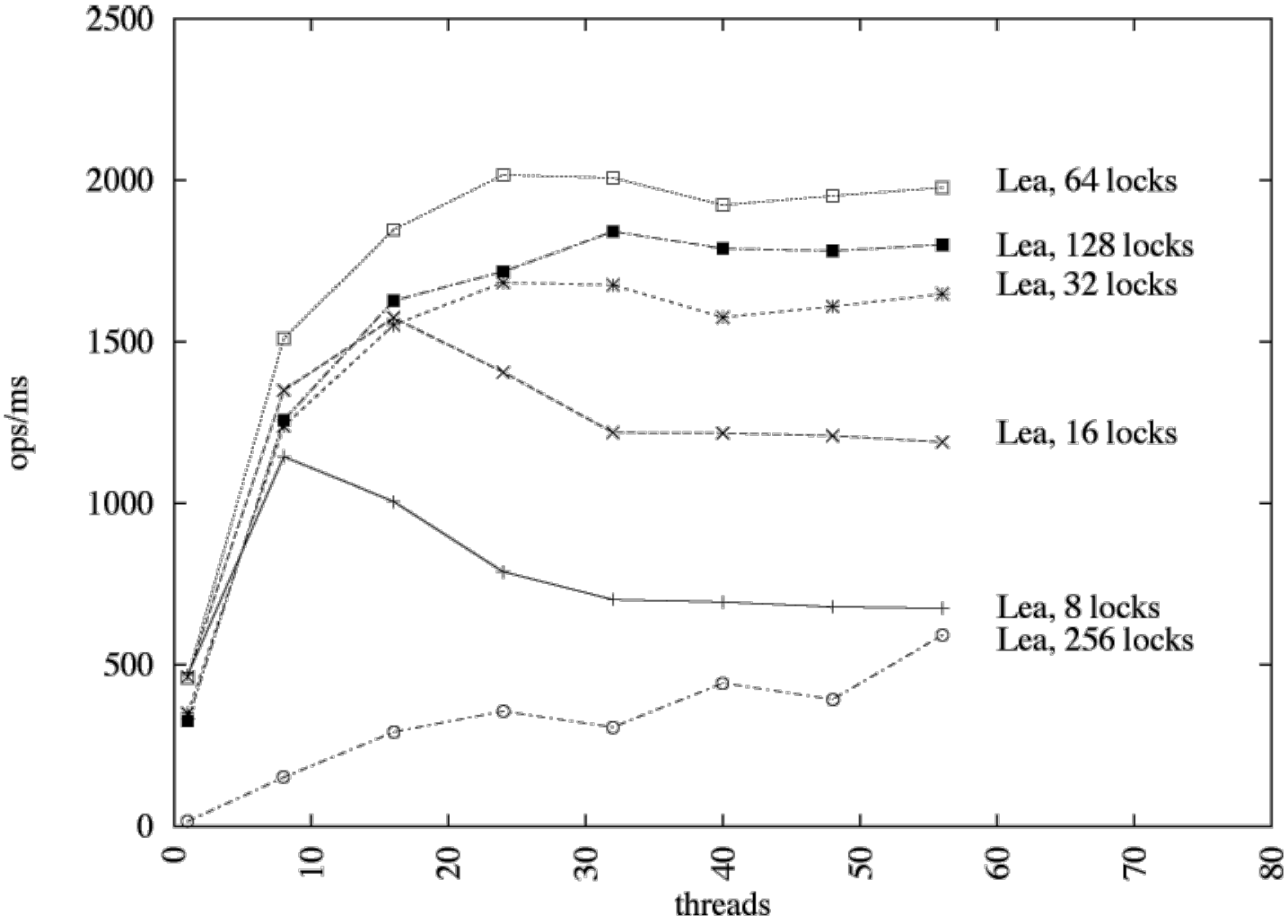


Solution: use a Sentinel node for each bucket

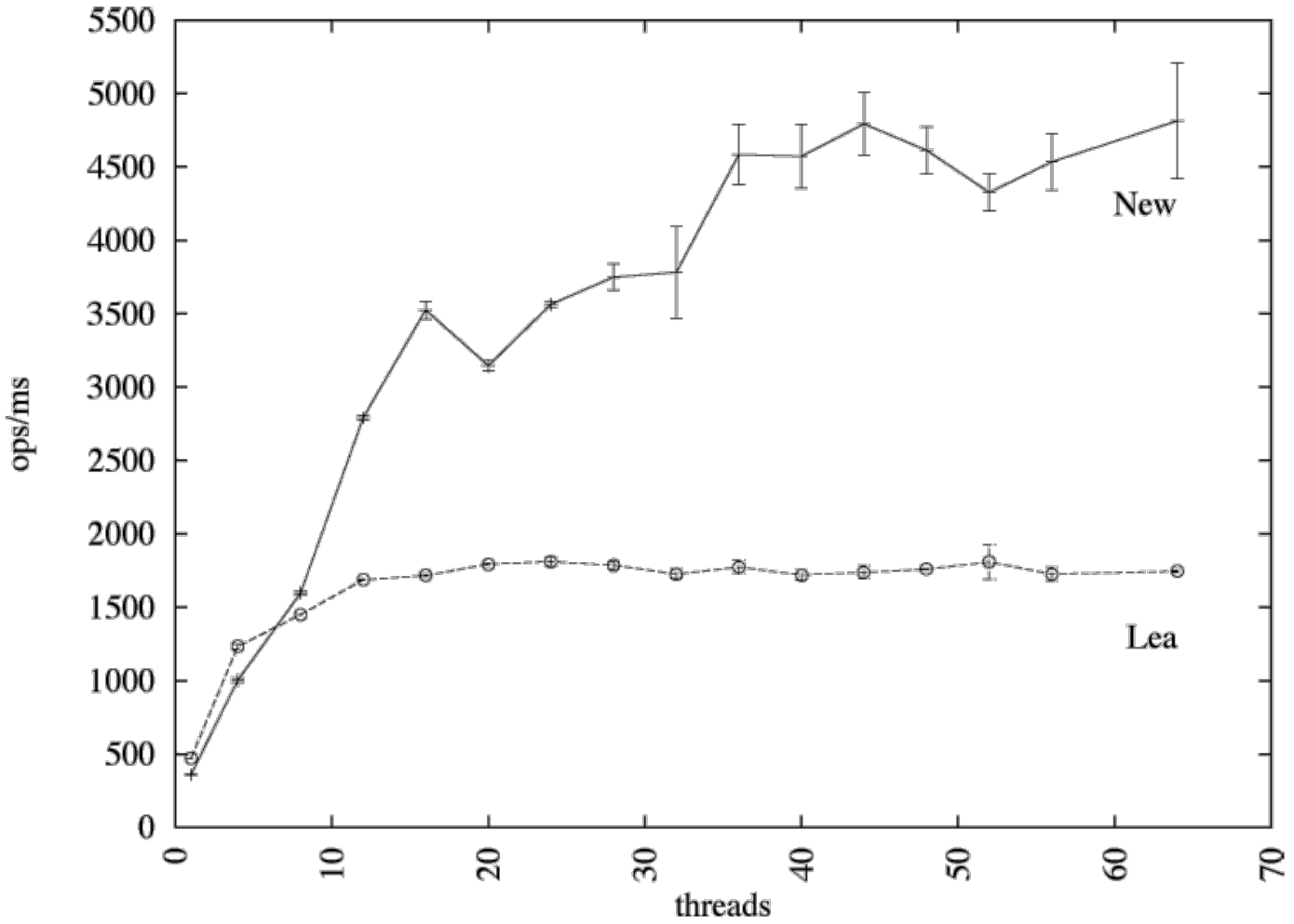
Empirical Evaluation

- On a 30-processor Sun Enterprise 3000
- Lock-Free vs. Fine-Grained (D. Lea) Optimistic
- 10^6 operations: **88%** *contains()*, **10%** *add()*, **2%** *remove()*

Number of Fine-Grain Locks



Lock-free vs Locks



Coarse-Grained Simulation for Exascale Software Evaluation

The Problem: Exascale

- We don't yet completely understand the incredibly complex design space in the exascale regime
  - Application scaling
  - Programming, communication, execution models
  - Data management
  - Fault tolerance at all levels
  - Power, performance, and cost tradeoffs for hardware technologies

- What we will end up with if we don't understand:
  - A machine we can't turn on (because the power: $$$)
  - A machine that turns itself off (because it fails a lot)
  - Extreme diminishing returns on machine investment for useful work (application performance)

Sandia
National
Laboratories

SST/macro

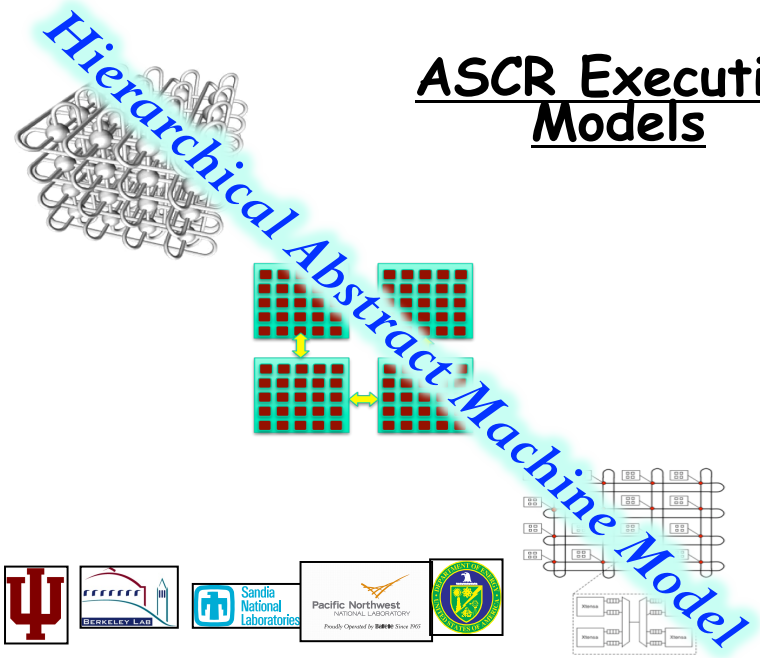
http://sst.sandia.gov/about_sstmacro.html

- A coarse-grained communication and synchronization simulator
  - Network (structure, features, parameters)
  - Libraries (MPI, etc)
  - Application (skeletons)
- Built to scale
  - Lightweight threads (> 1M)
  - No computation in skeleton

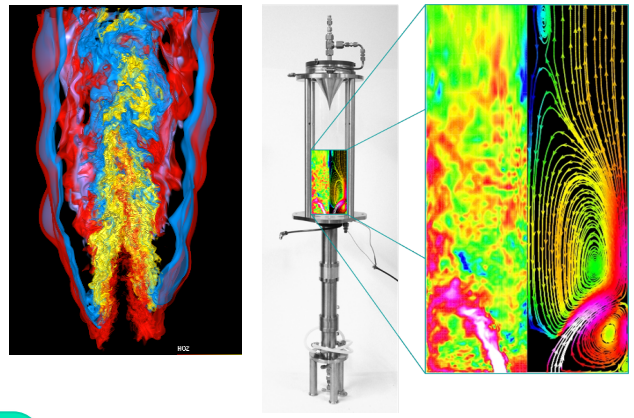
Sandia National Laboratories

SST/Macro Projects

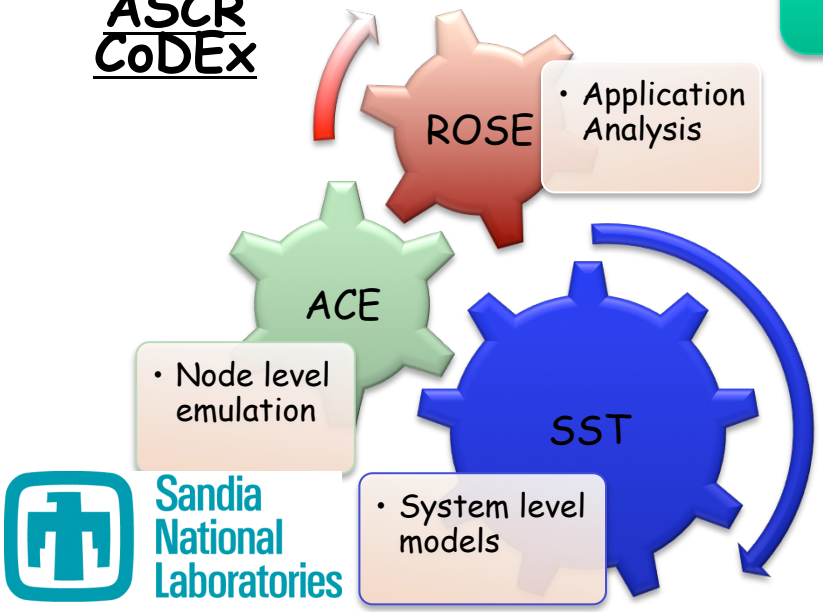
ASCR Execution Models



ASCR ExaCT: Combusion Codesign



ASCR CoDEx



SST/ Macro

- SST/Macro is a key tool in a number of exascale/co-design efforts.
- It brings key capabilities that work synergistically with other tools to form a complete picture