Concurrent Hashing

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & 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?

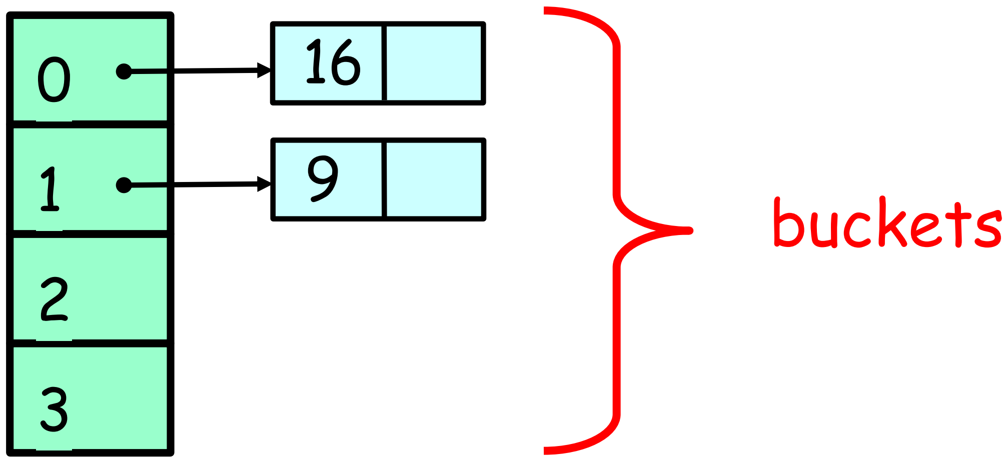
Linear-Time Set Methods

- Problem is
  - **add()**, **remove()**, **contains()**
  - Take time linear in set size
- We want
  - Constant-time methods
  - (at least, on average)

Hashing

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

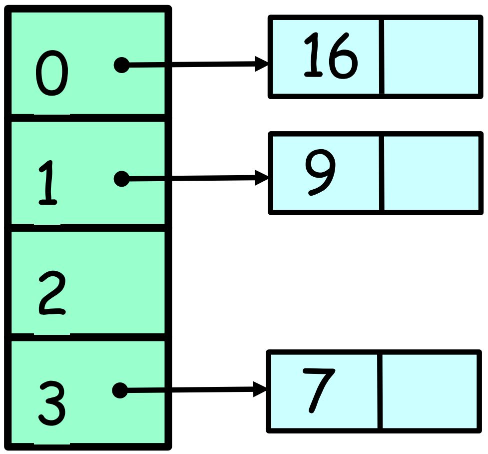
Sequential Hash Map



2 Items

h(k) = k mod 4

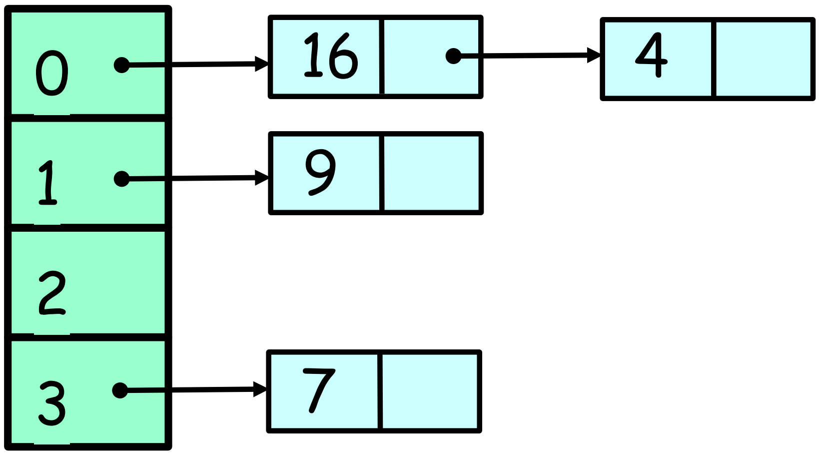
Add an Item



3 Items

h(k) = k mod 4

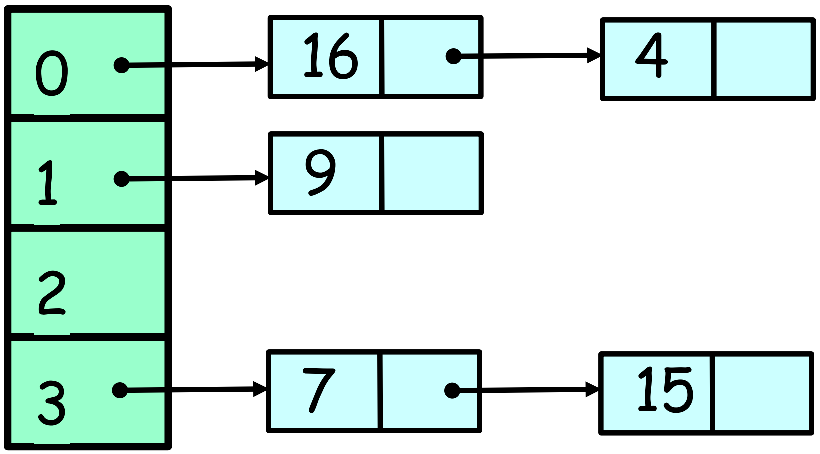
Add Another: Collision



4 Items

h(k) = k mod 4

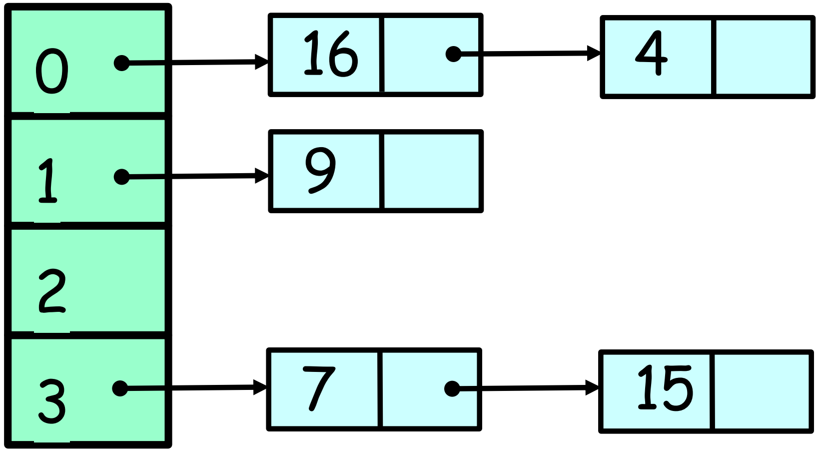
More Collisions



5 Items

h(k) = k mod 4

More Collisions

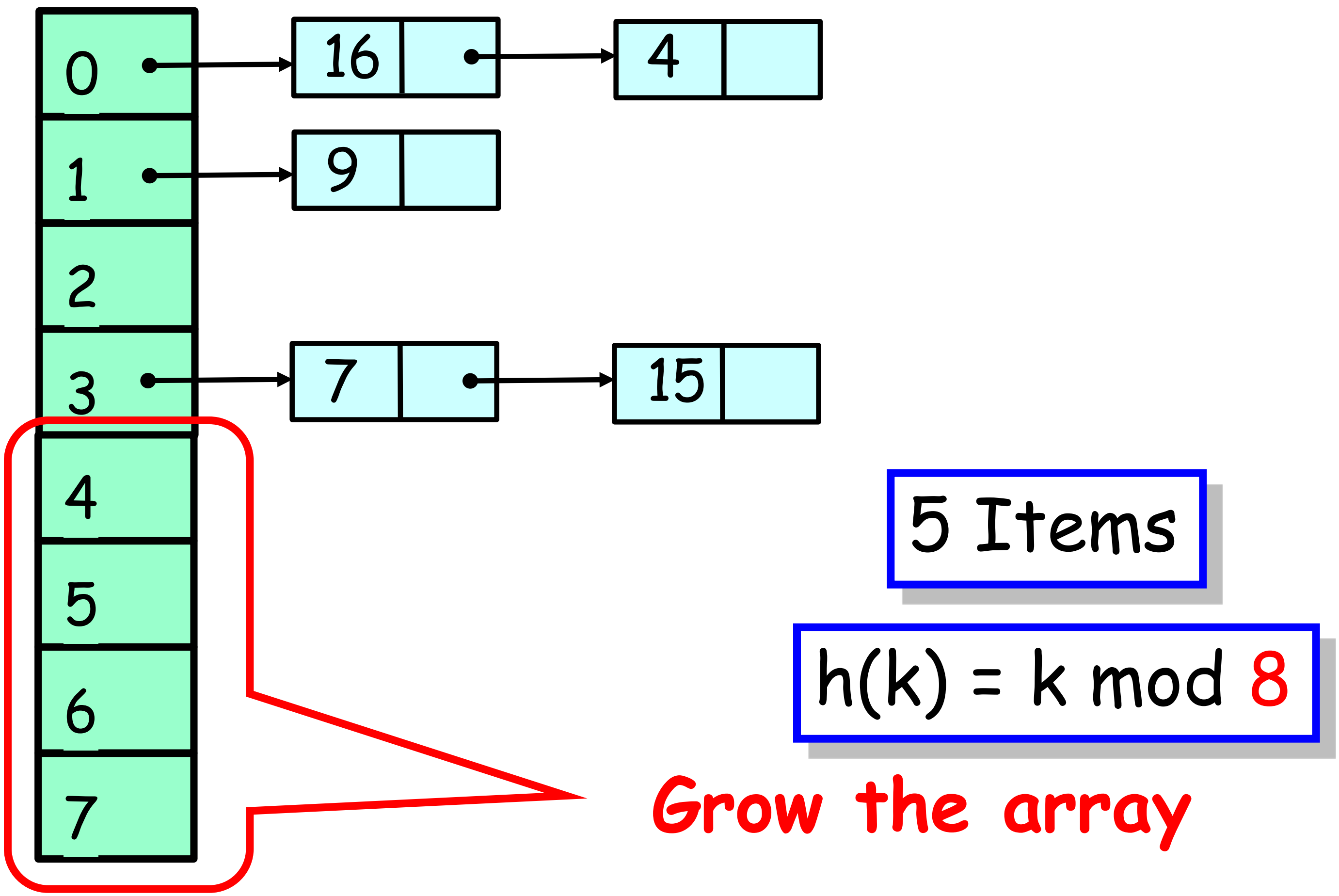


5 Items

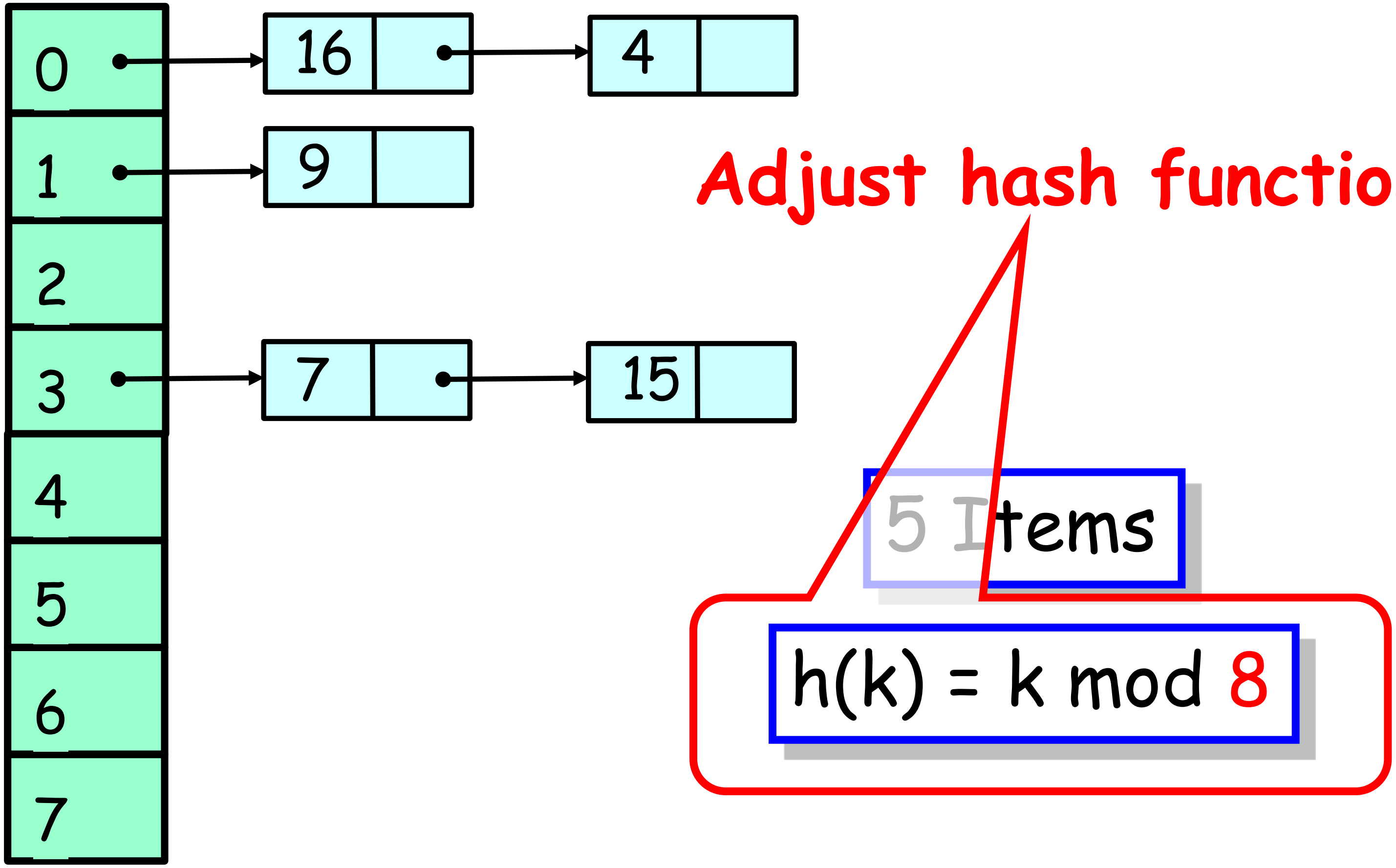
h(k) = k mod 4

Problem: buckets getting too long

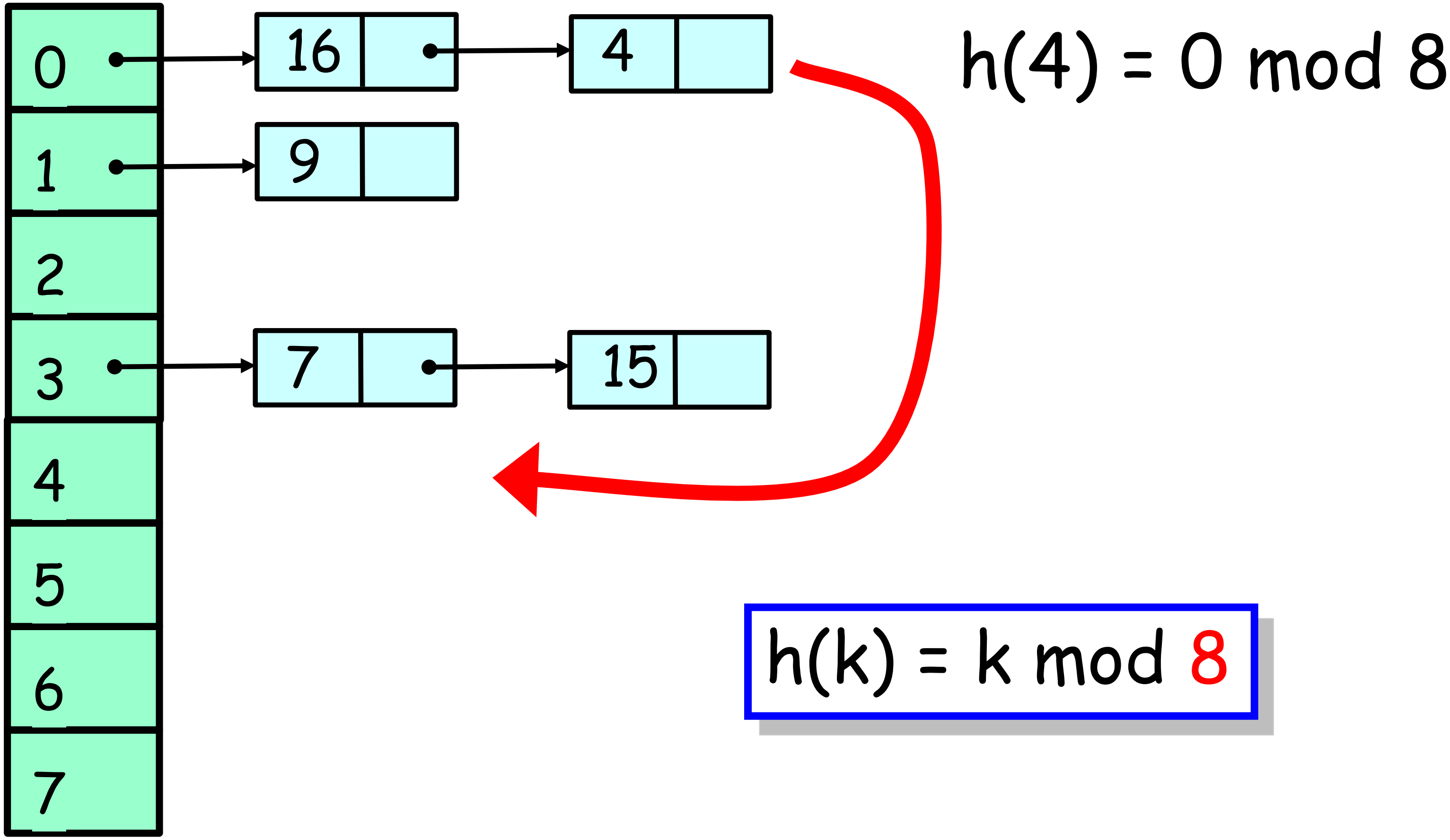
Resizing



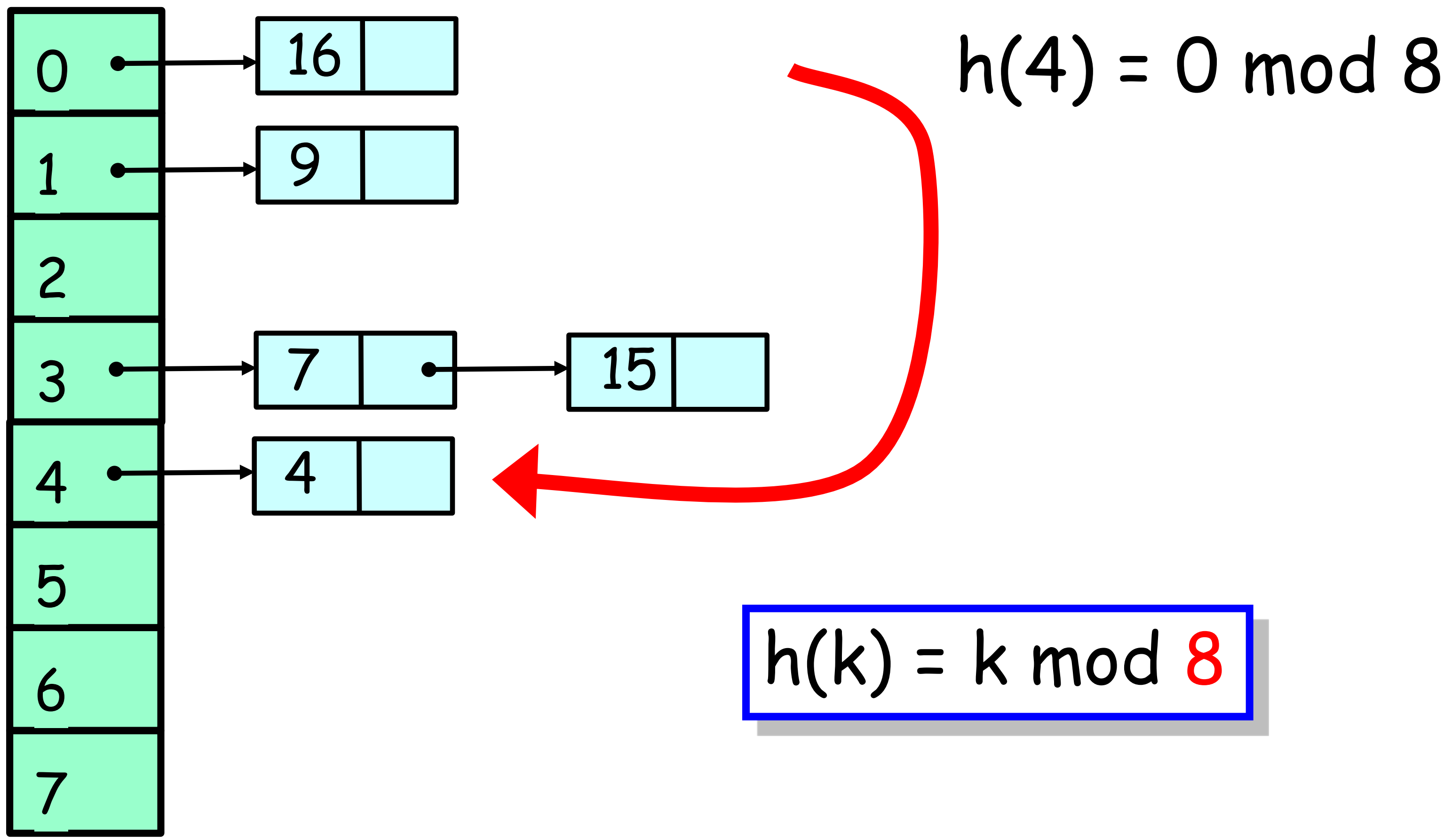
Resizing



Resizing

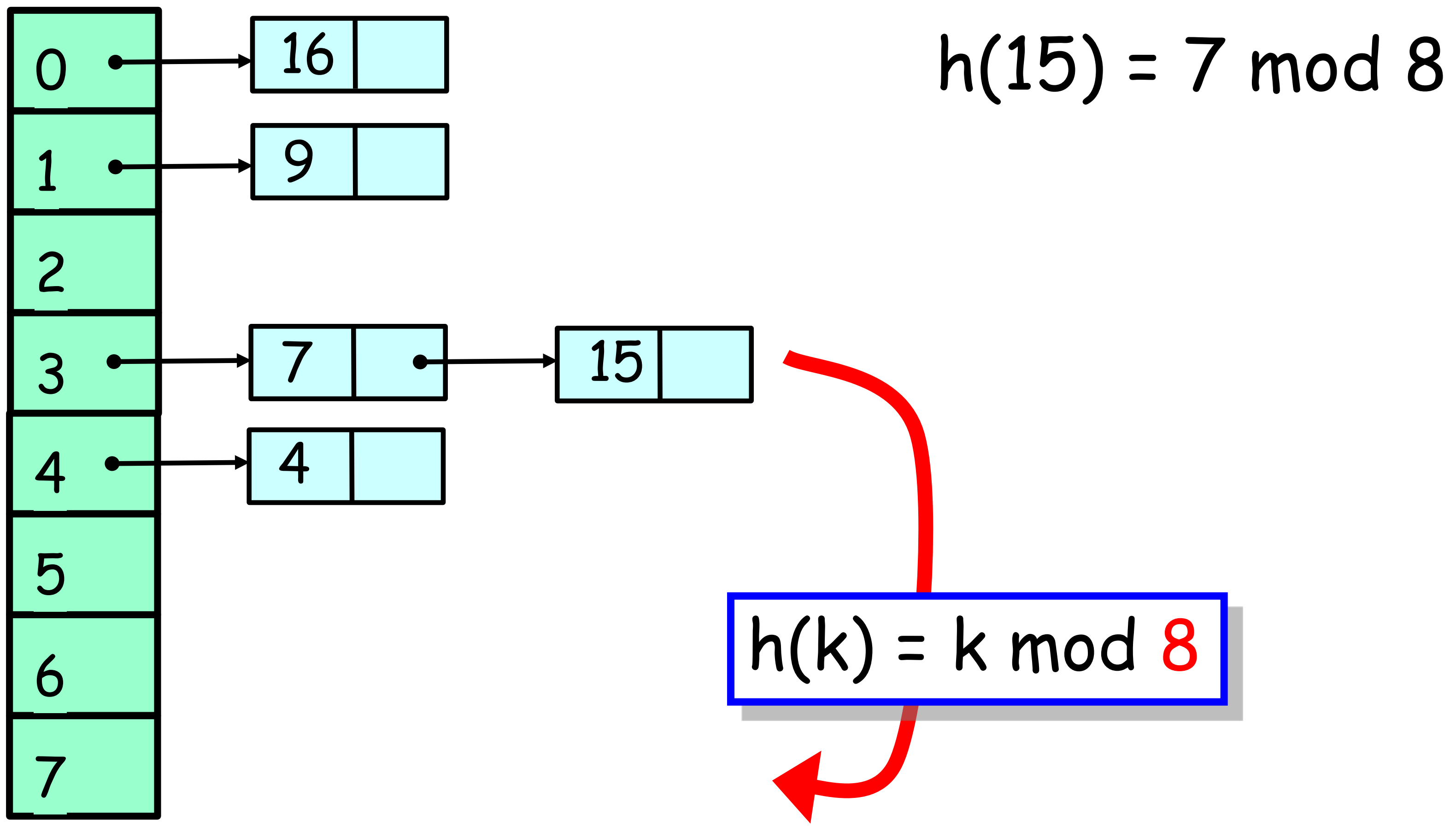


Resizing

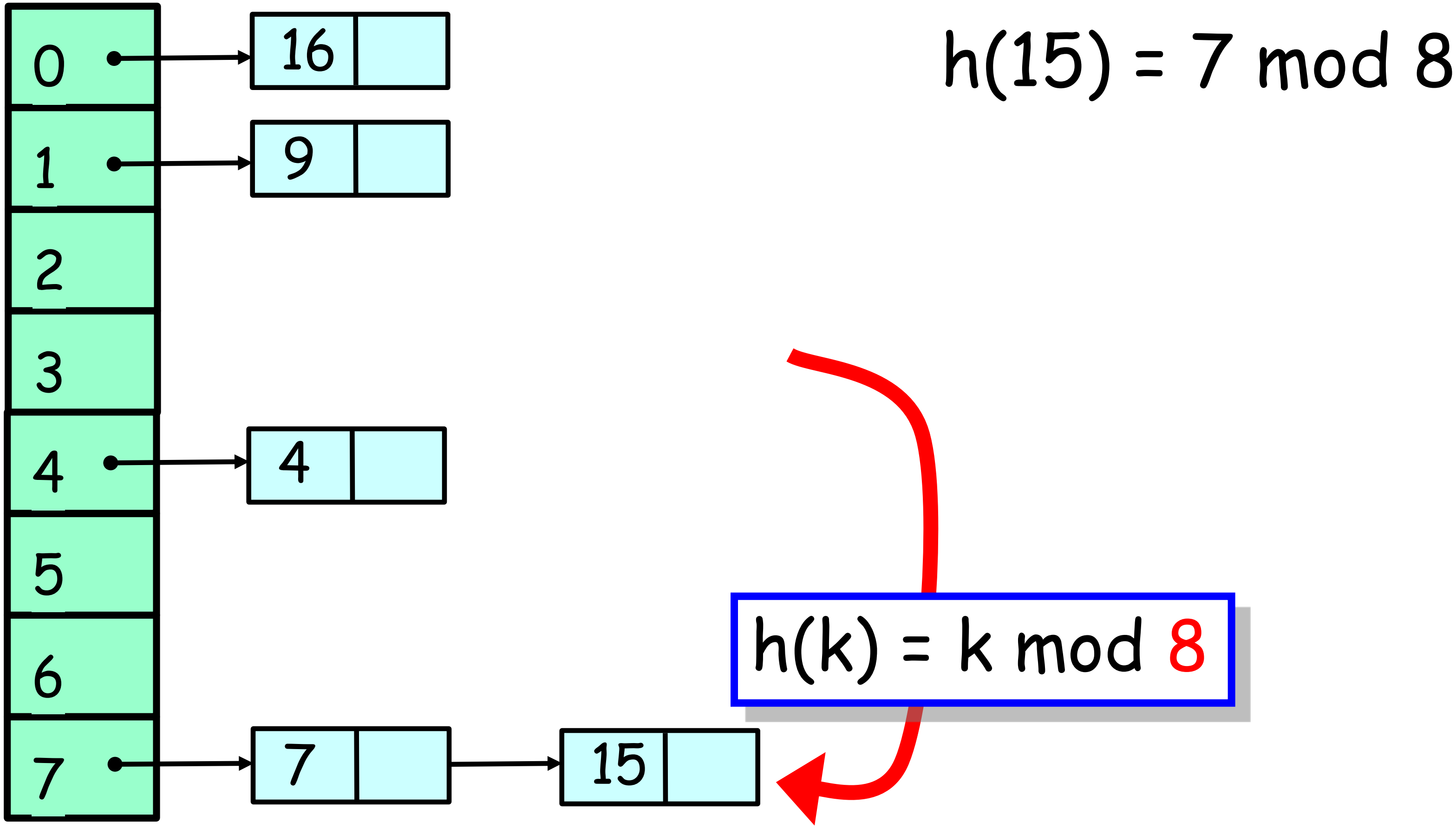


$h(k) = k \bmod 8$

Resizing



Resizing



Hash Sets

- Implement a Set object
  - Collection of items, no duplicates
  - **add()**, **remove()**, **contains()** methods
  - You know the drill …

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 much

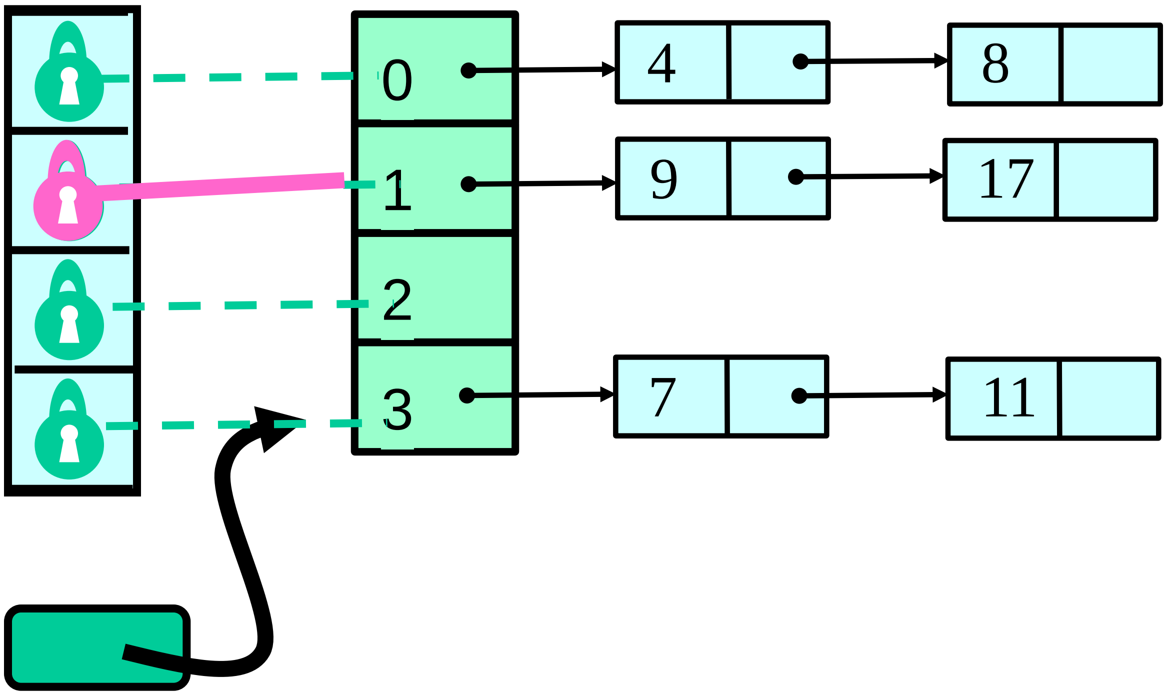
When to Resize?

- 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

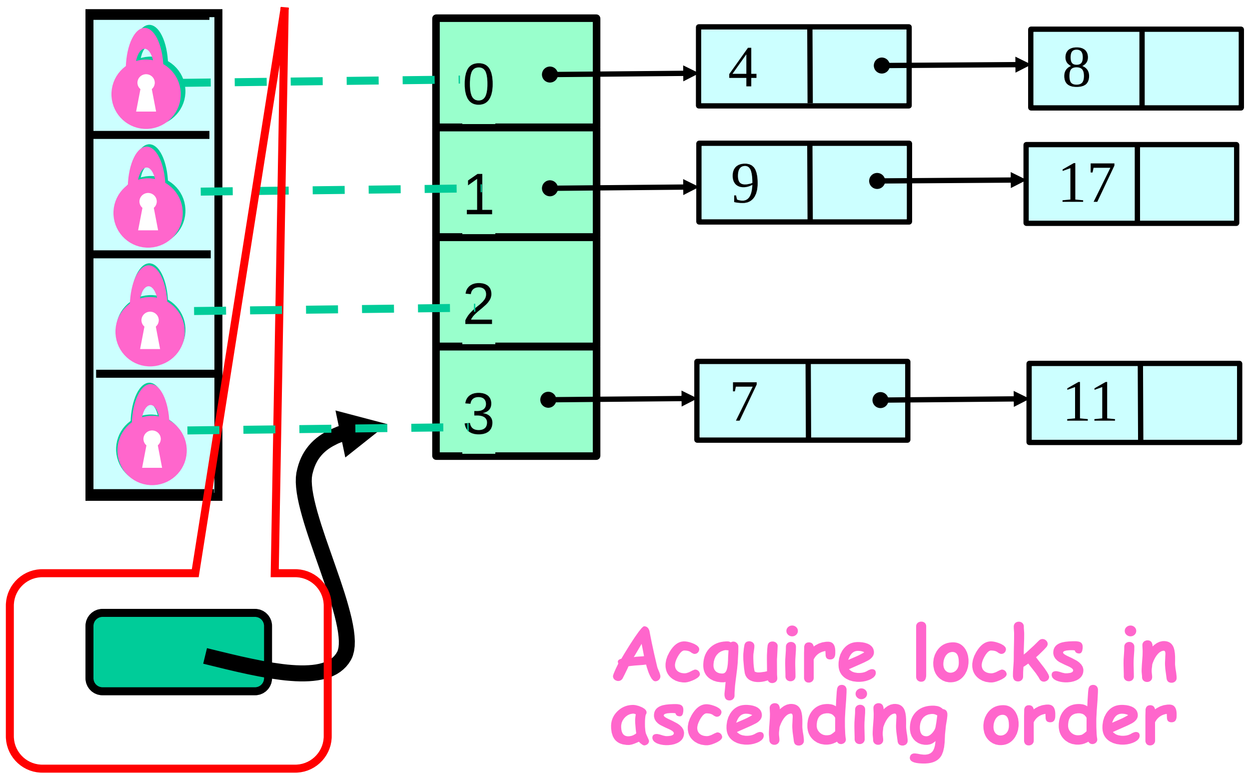
- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck

Fine-grained Locking

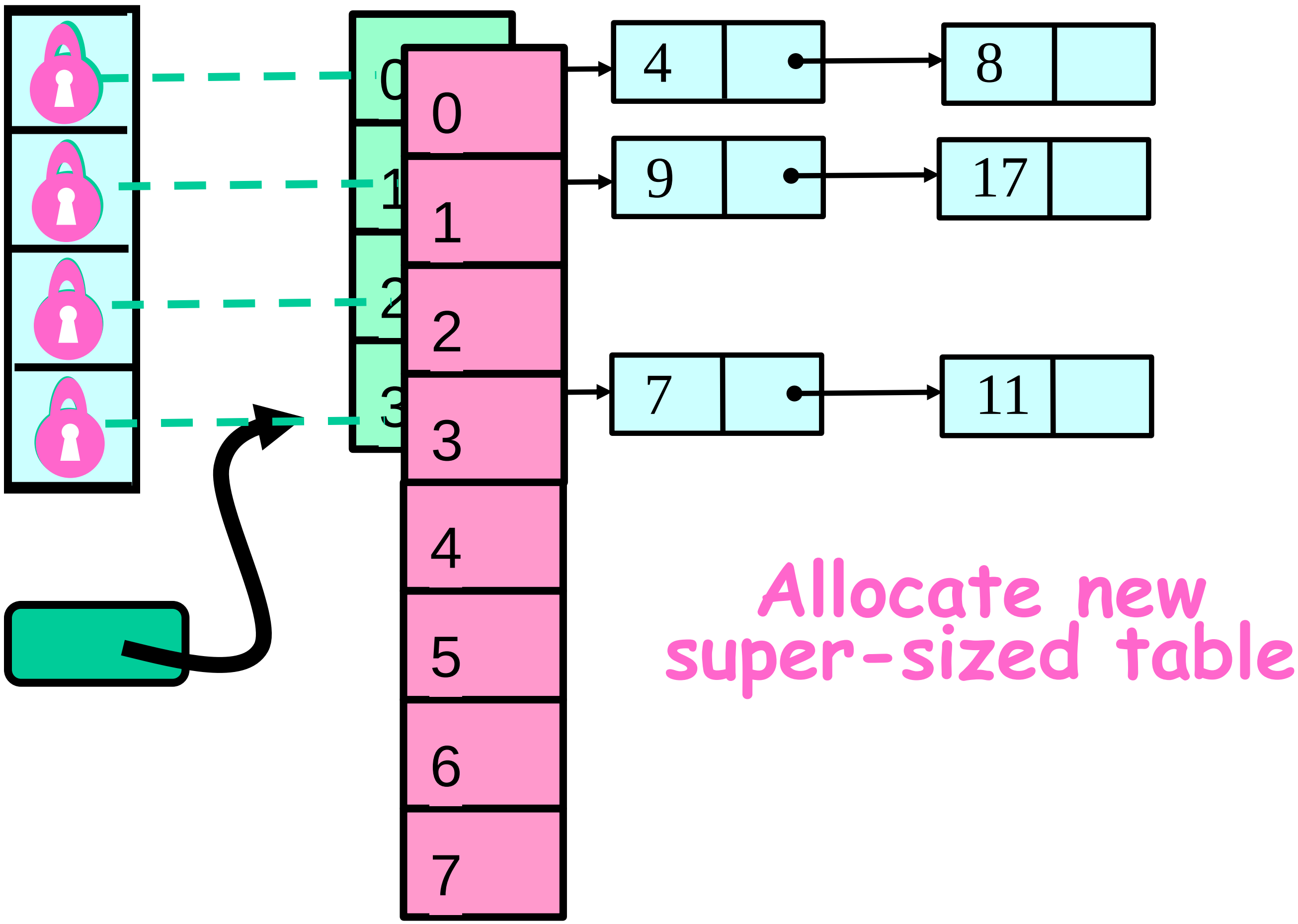


Each lock associated with one bucket

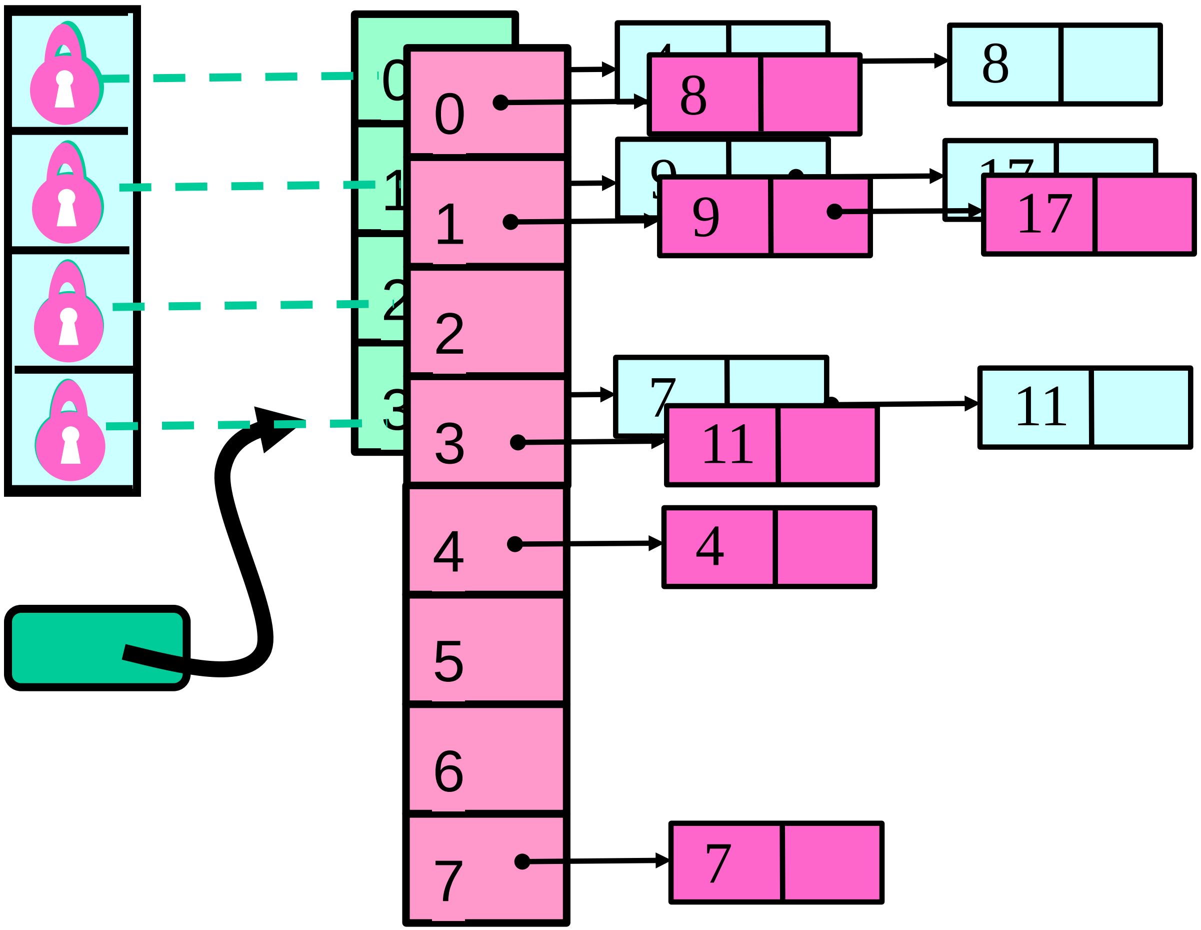
Make sure table reference didn't change between resize decision and lock acquisition



Resize This

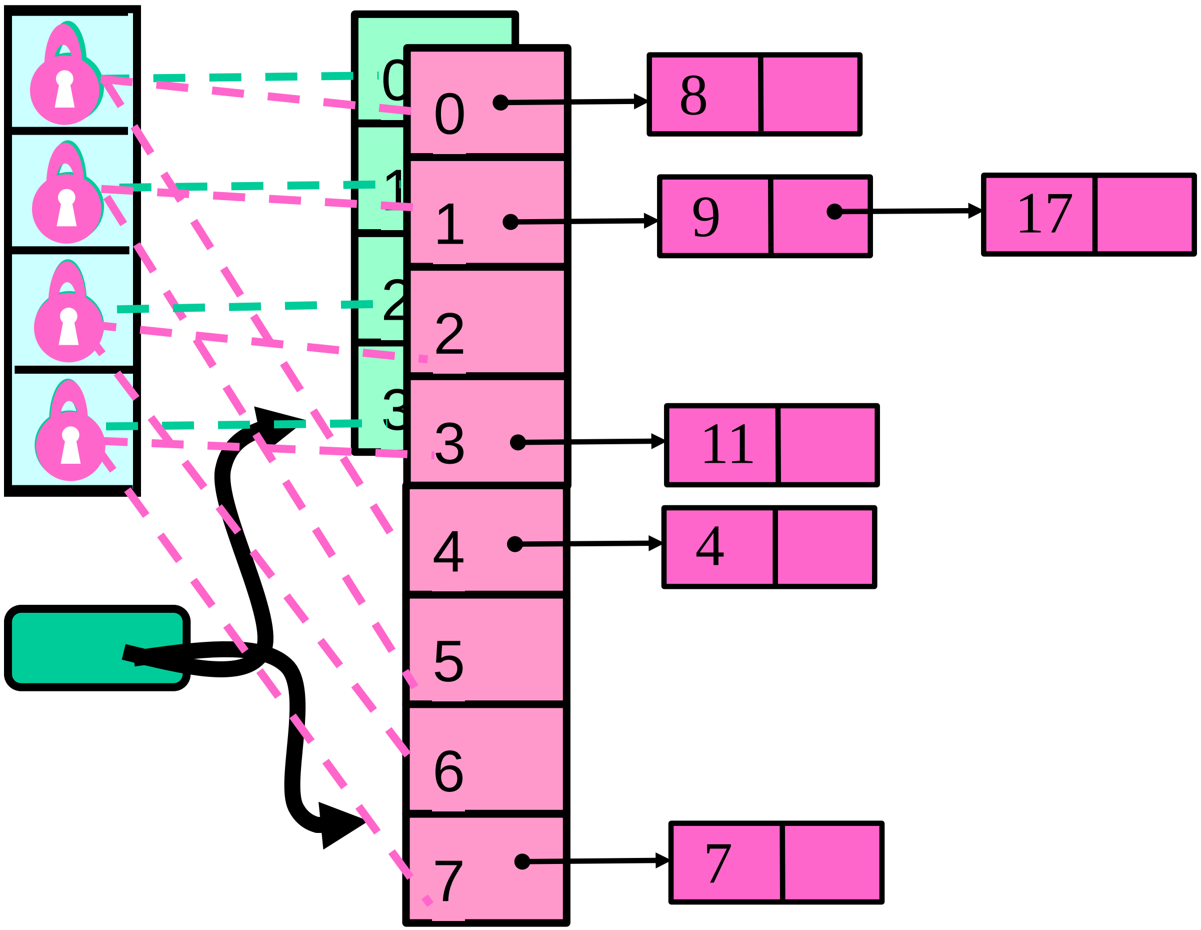


Resize This



Striped Locks: each lock now associated with two buckets

Resize This



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

Initially same number of locks and buckets

The add() method

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

Fine-Grained Locking

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

Which lock?

The add() method

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

Acquire the lock

Fine-Grained Locking

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

Which bucket?

The add() method

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

Call that bucket's add() method

Fine-Grained Locking

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

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

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
     sequentialResize();
  }}}
```

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,
                    MRSWLock[] 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 …

Insight

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

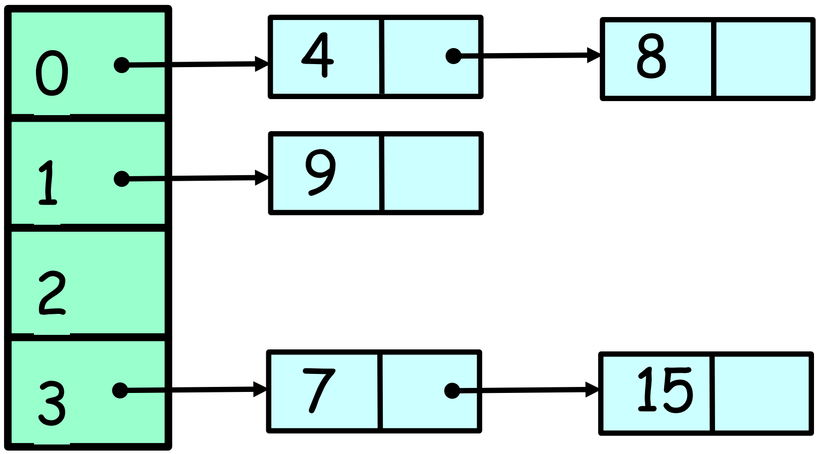
Optimistic Synchronization

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

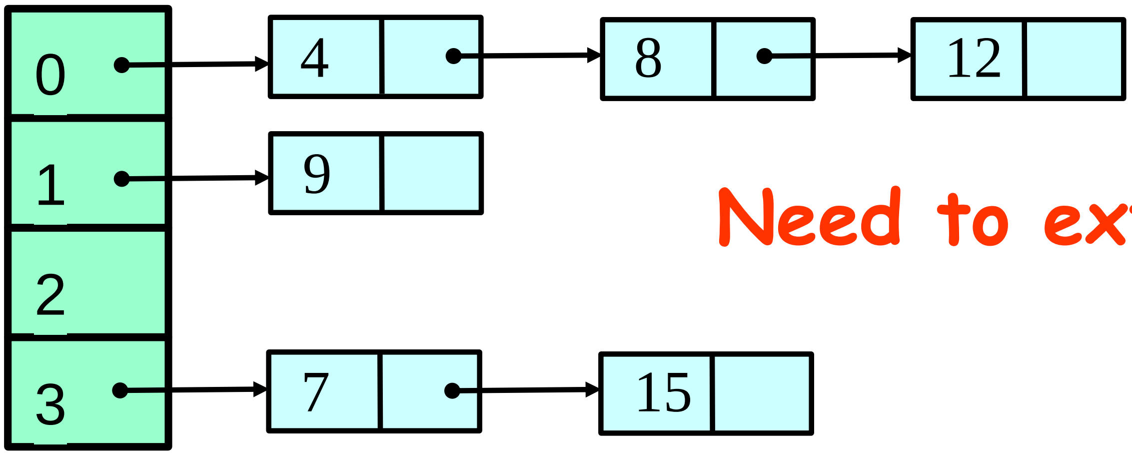
Stop The World Resizing

- 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

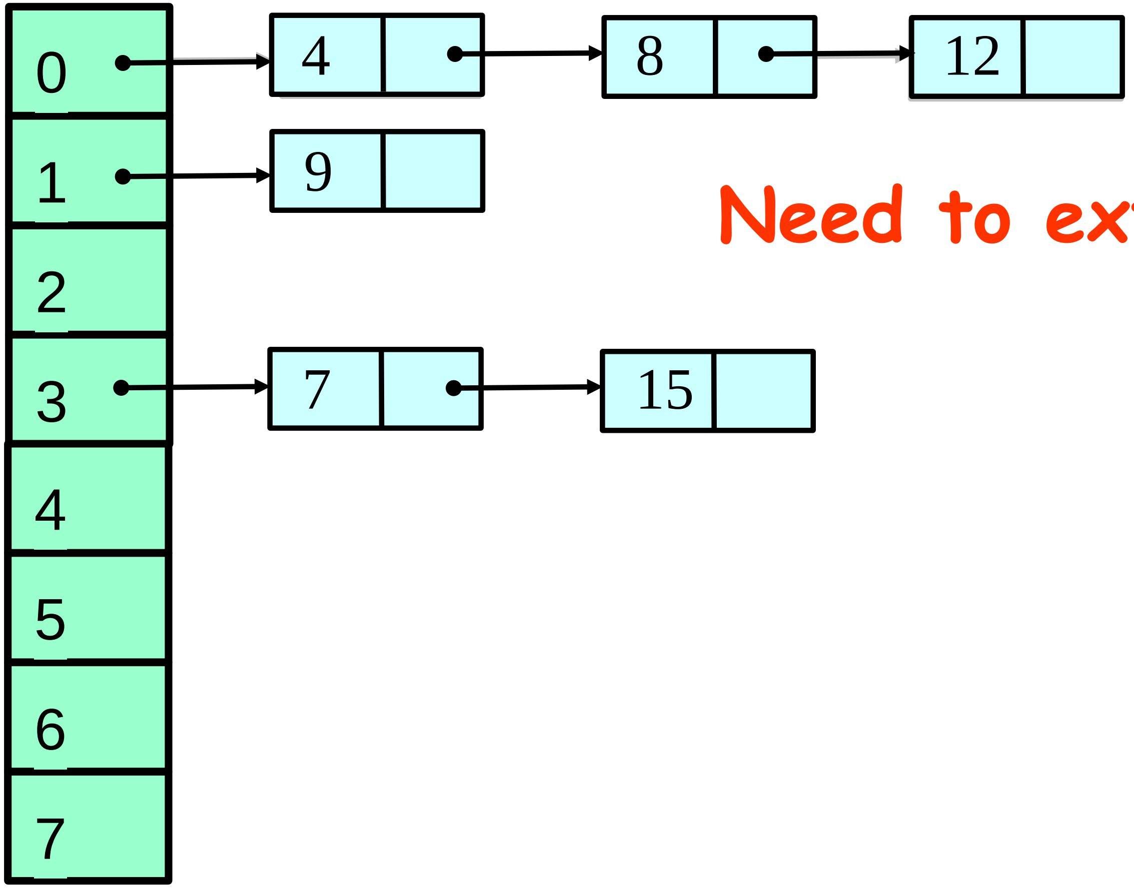


Lock-Free Resizing Problem



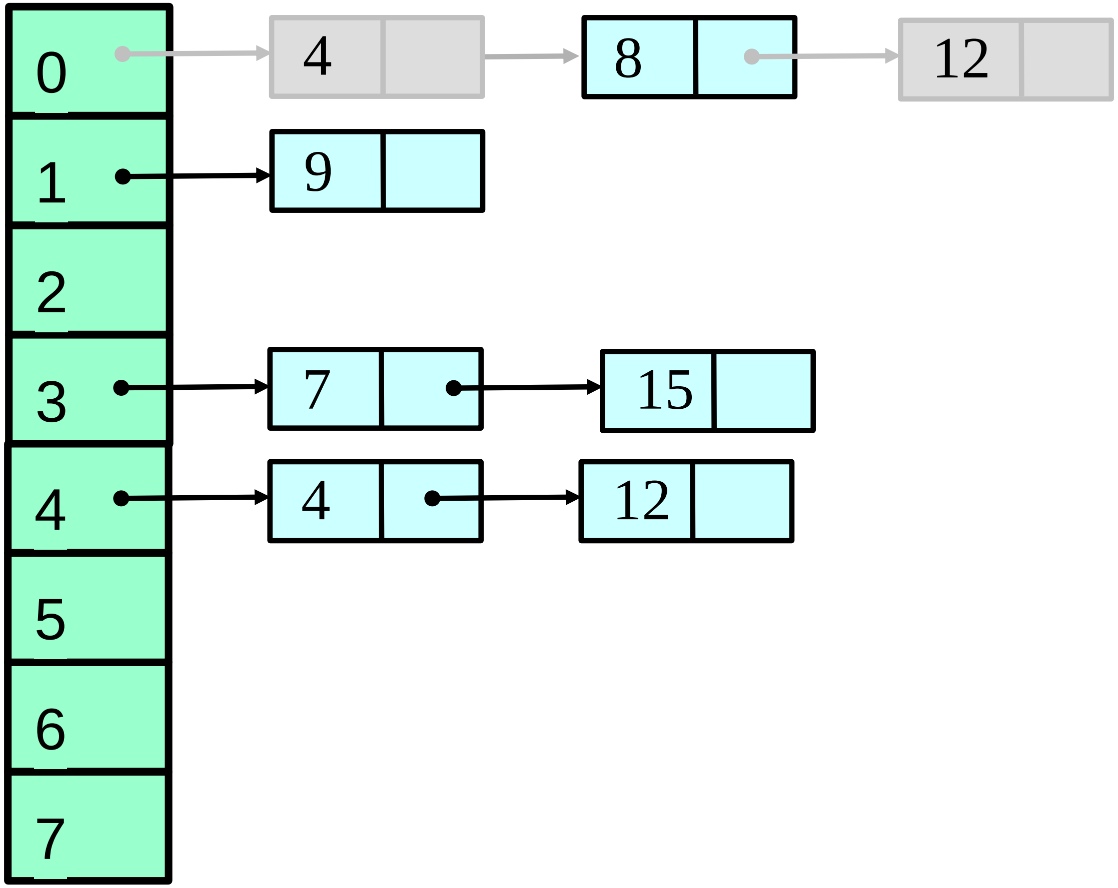
Need to extend table

Lock-Free Resizing Problem

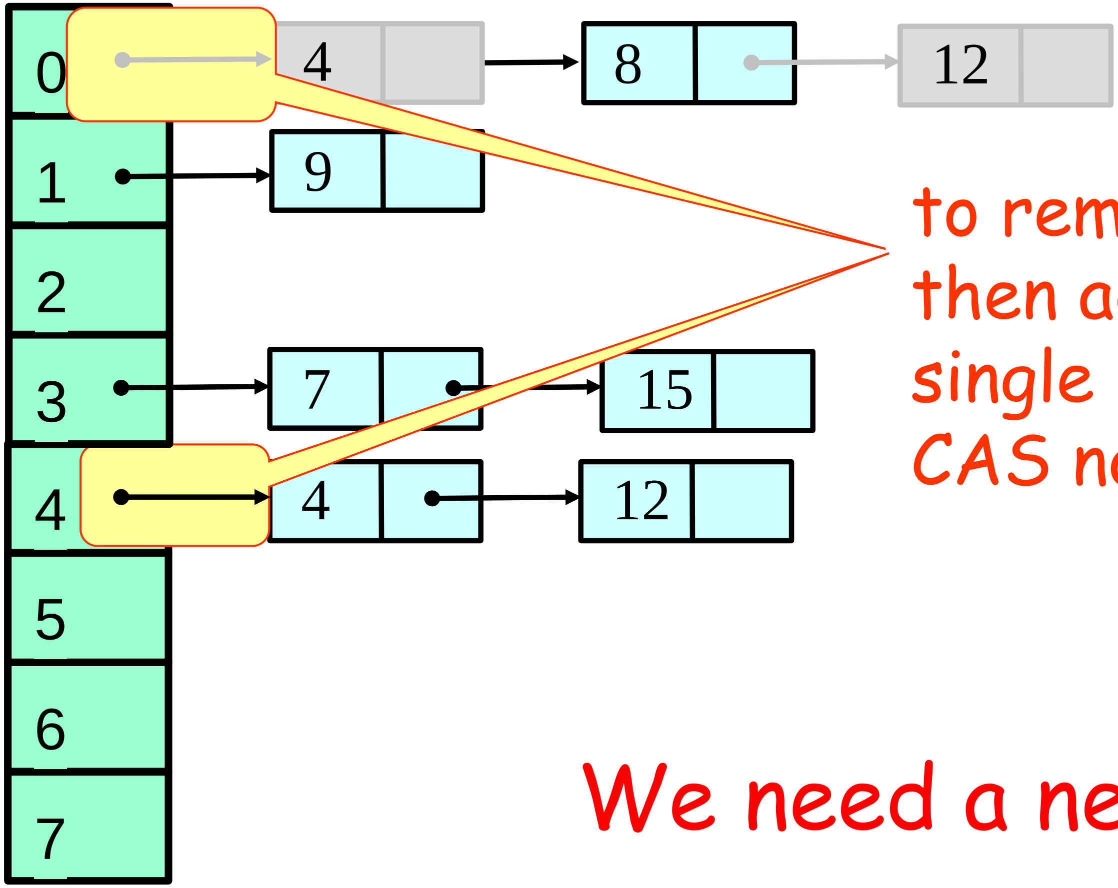


Need to extend table

Lock-Free Resizing Problem



Lock-Free Resizing Problem

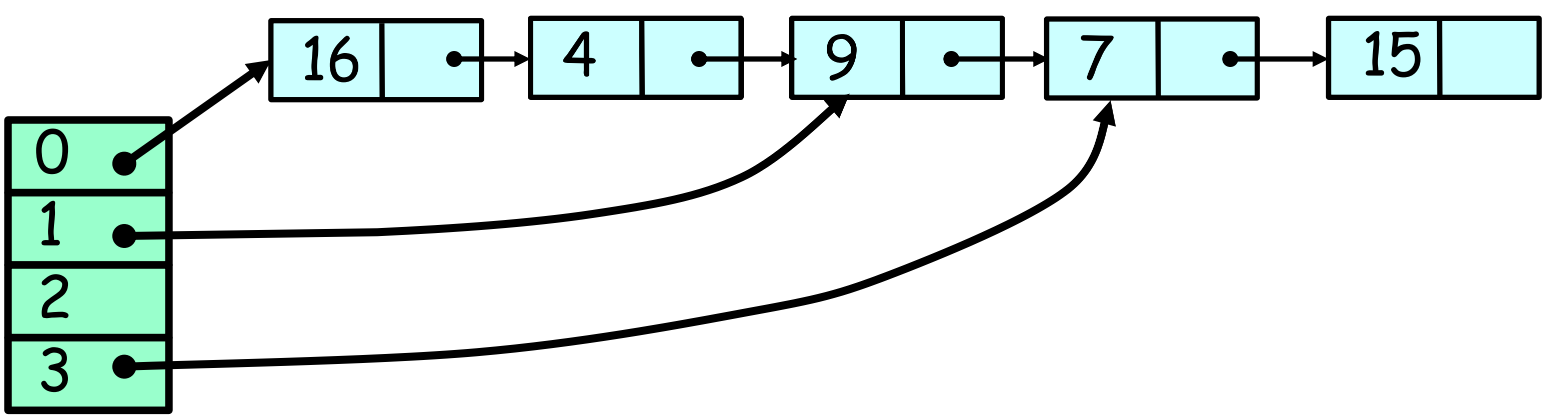


to remove and then add even a single item one CAS not enough

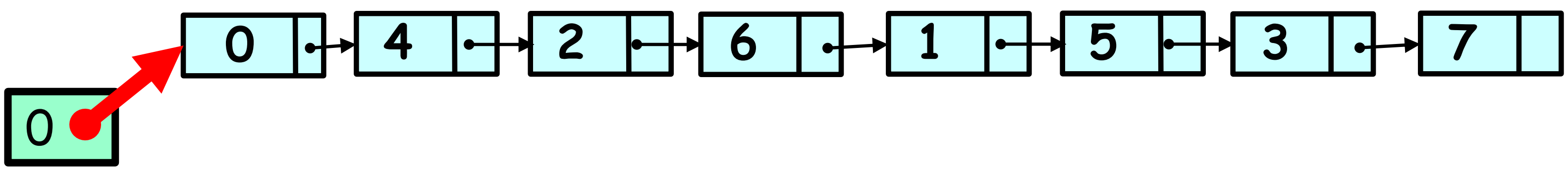
We need a new idea…

Don't move the items

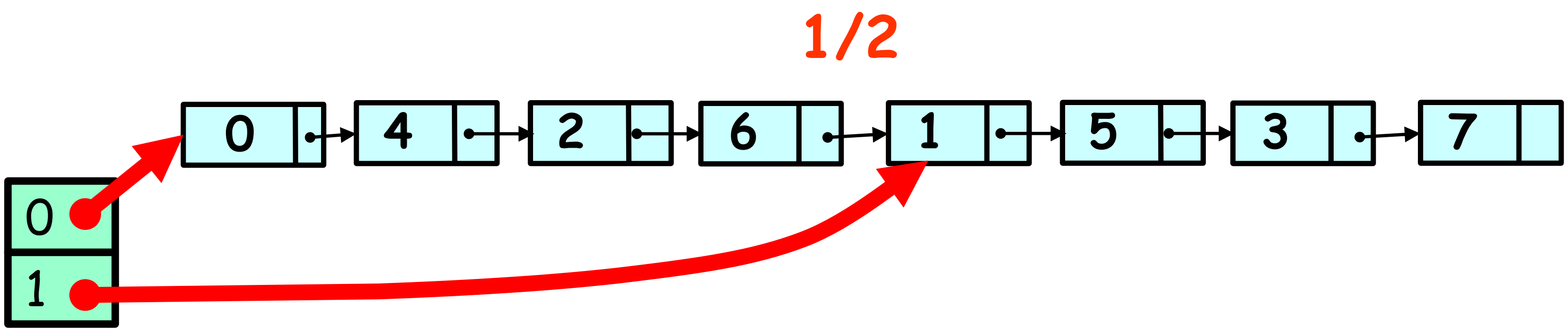
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



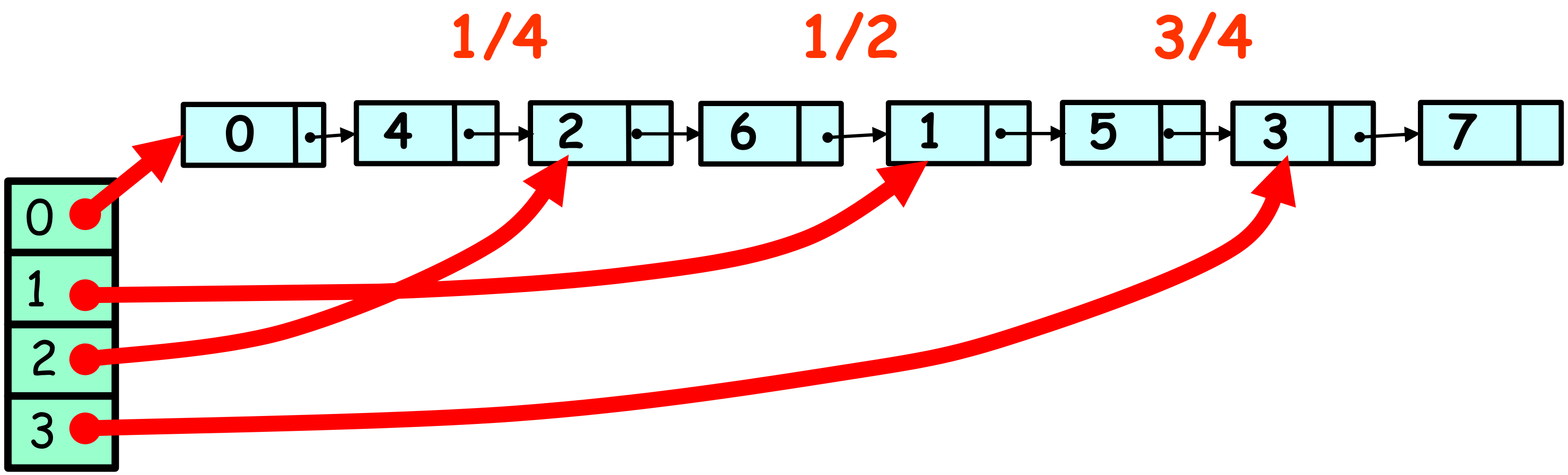
Recursive Split Ordering



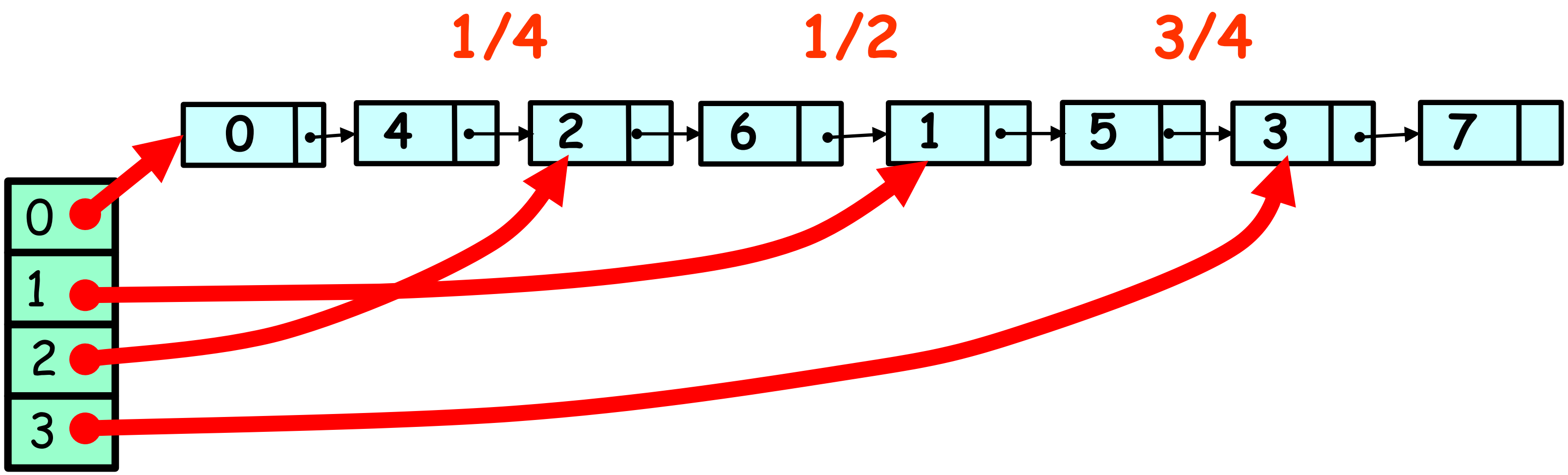
Recursive Split Ordering



Recursive Split Ordering

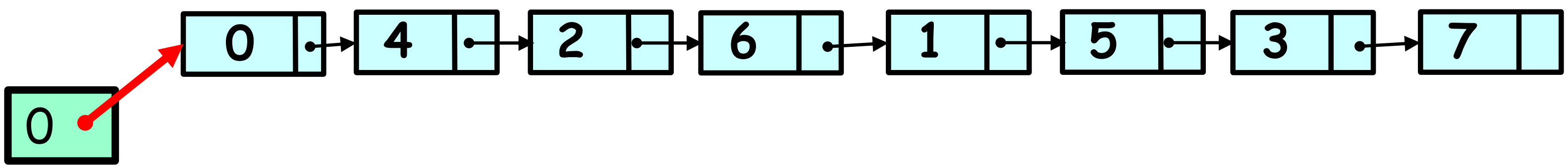


Recursive Split Ordering

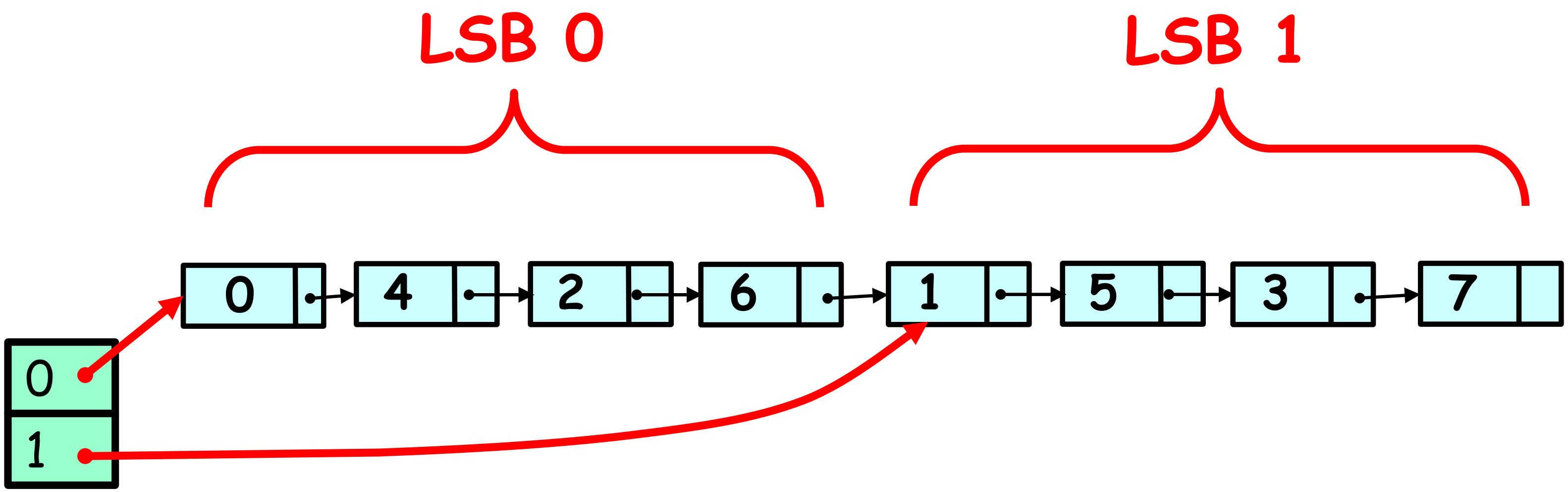


List entries sorted in order that allows recursive splitting. How?

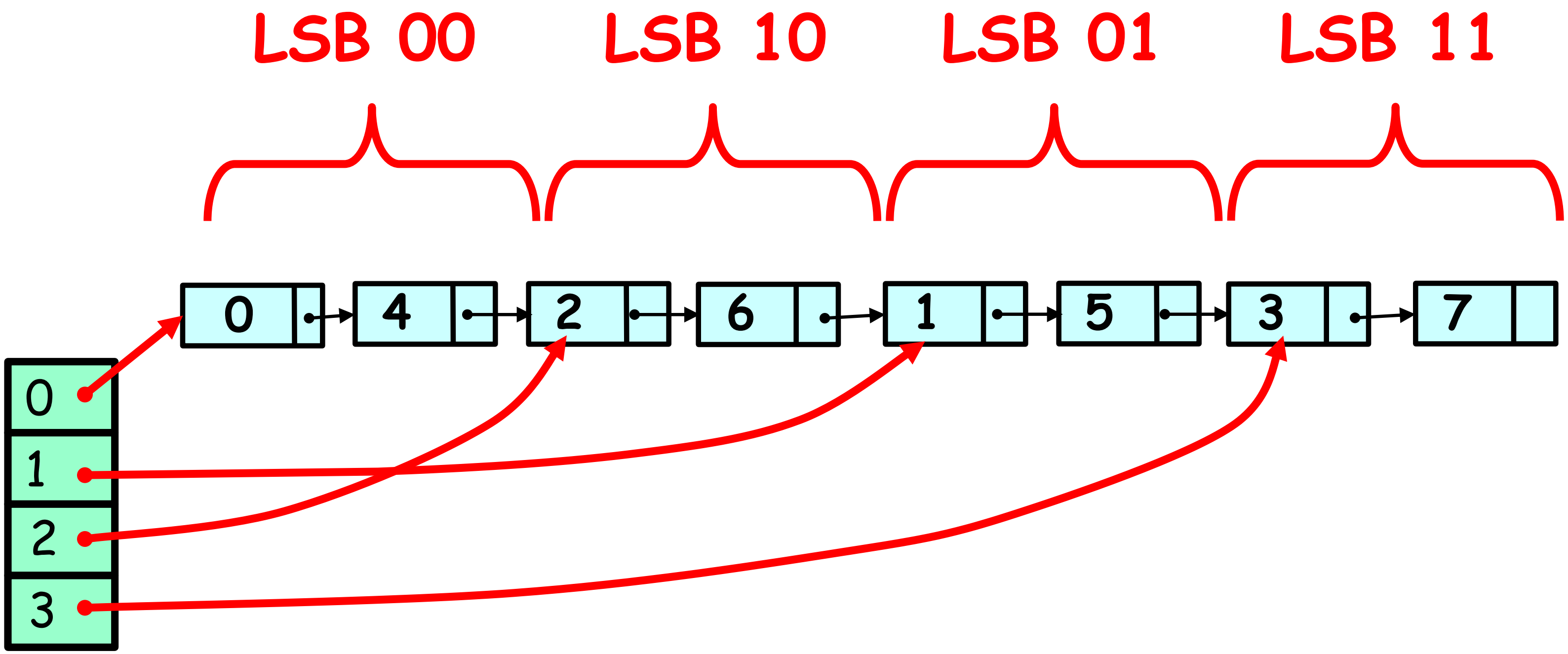
Recursive Split Ordering



Recursive Split Ordering



Recursive Split Ordering



Split-Order

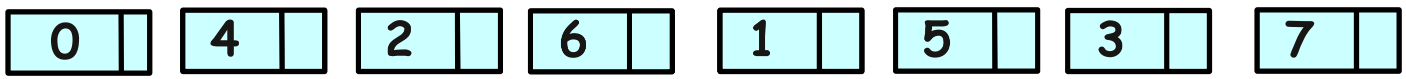
- If the table size is 2^i,
  - Bucket b contains keys k
    - $k = b \pmod{2^i}$
  - bucket index is key's i LSBs
  - (least significant bits)

When Table Splits

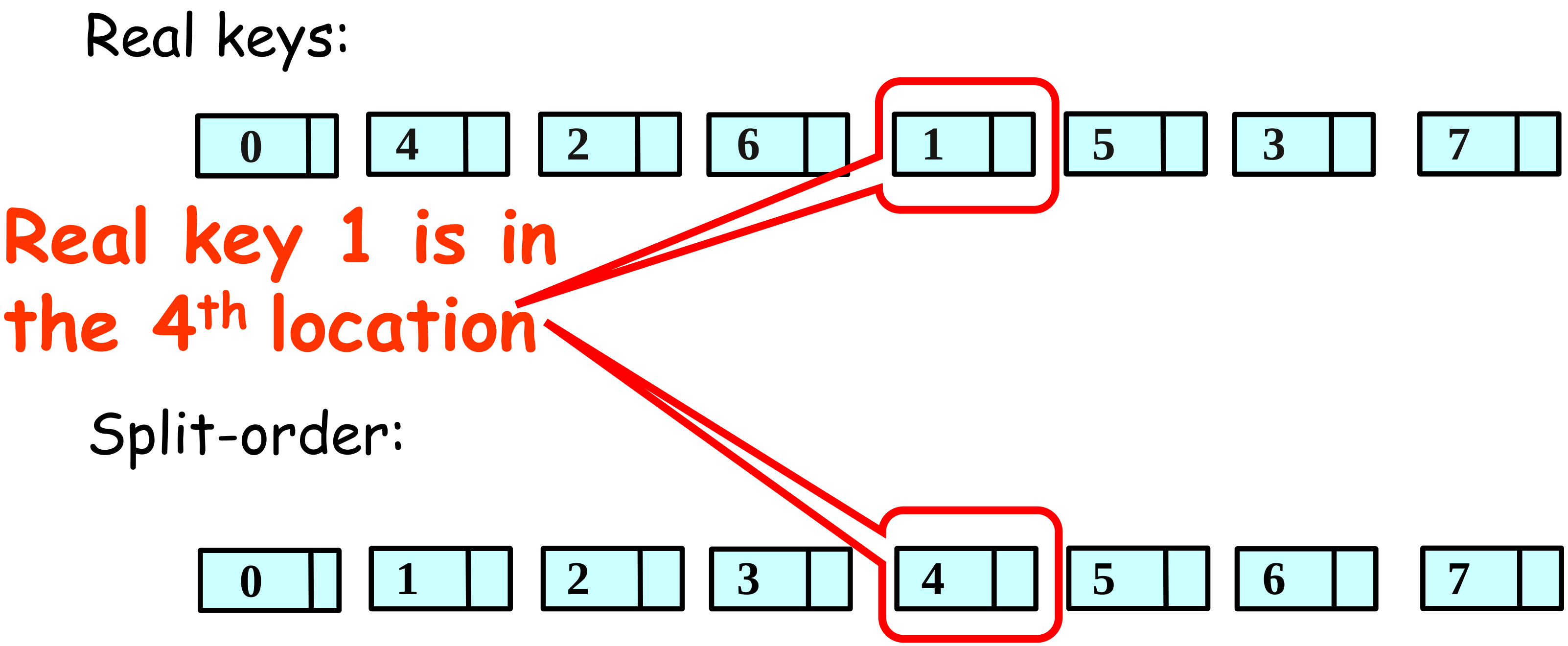
- Some keys stay
  - $b = k \bmod (2^{i+1})$
- Some move
  - $b+2^i = k \bmod (2^{i+1})$
- Determined by $(i+1)^{st}$ bit
  - Counting backwards
- Key must be accessible from both
  - Keys that will move must come later

A Bit of Magic

Real keys:

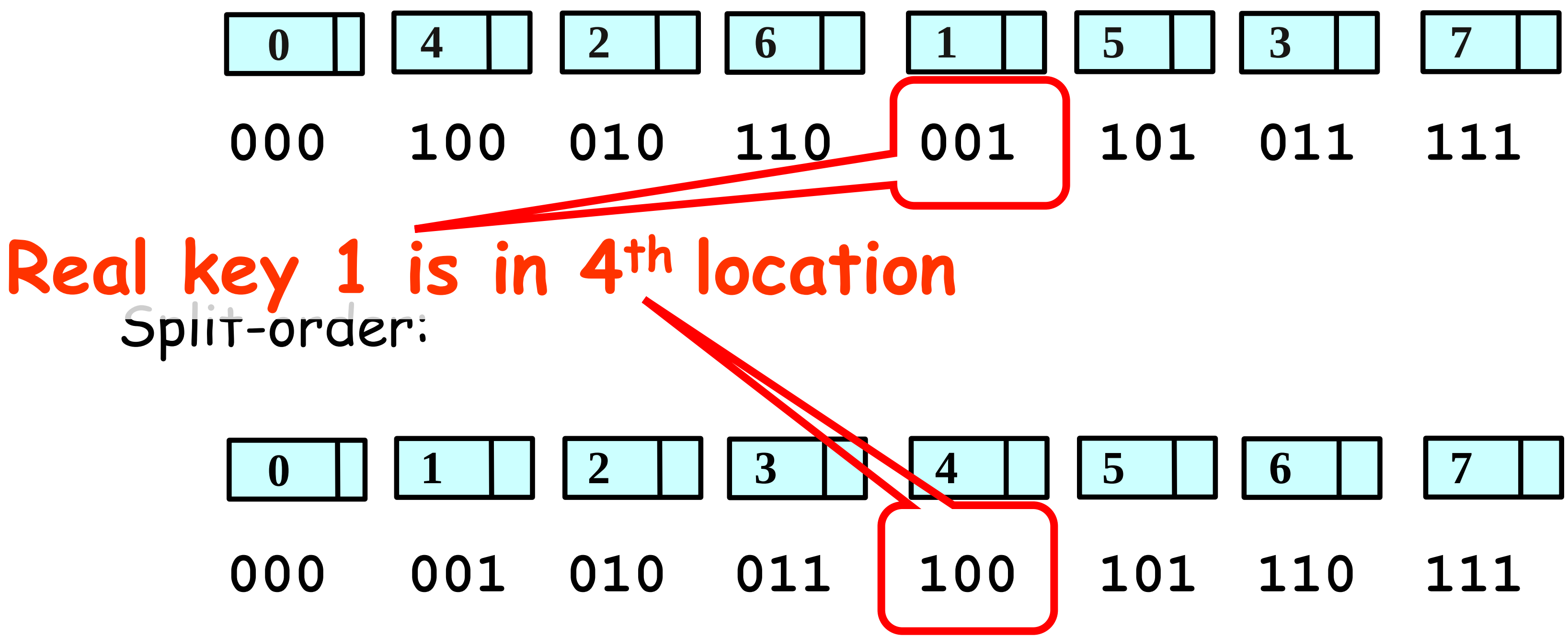


A Bit of Magic



A Bit of Magic

Real keys:



A Bit of Magic

Real keys:

`000 100 010 110 001 101 011 111`

Split-order:

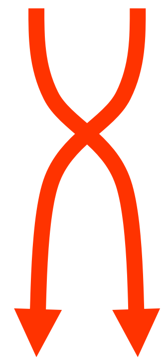
`000 001 010 011 100 101 110 111`

A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

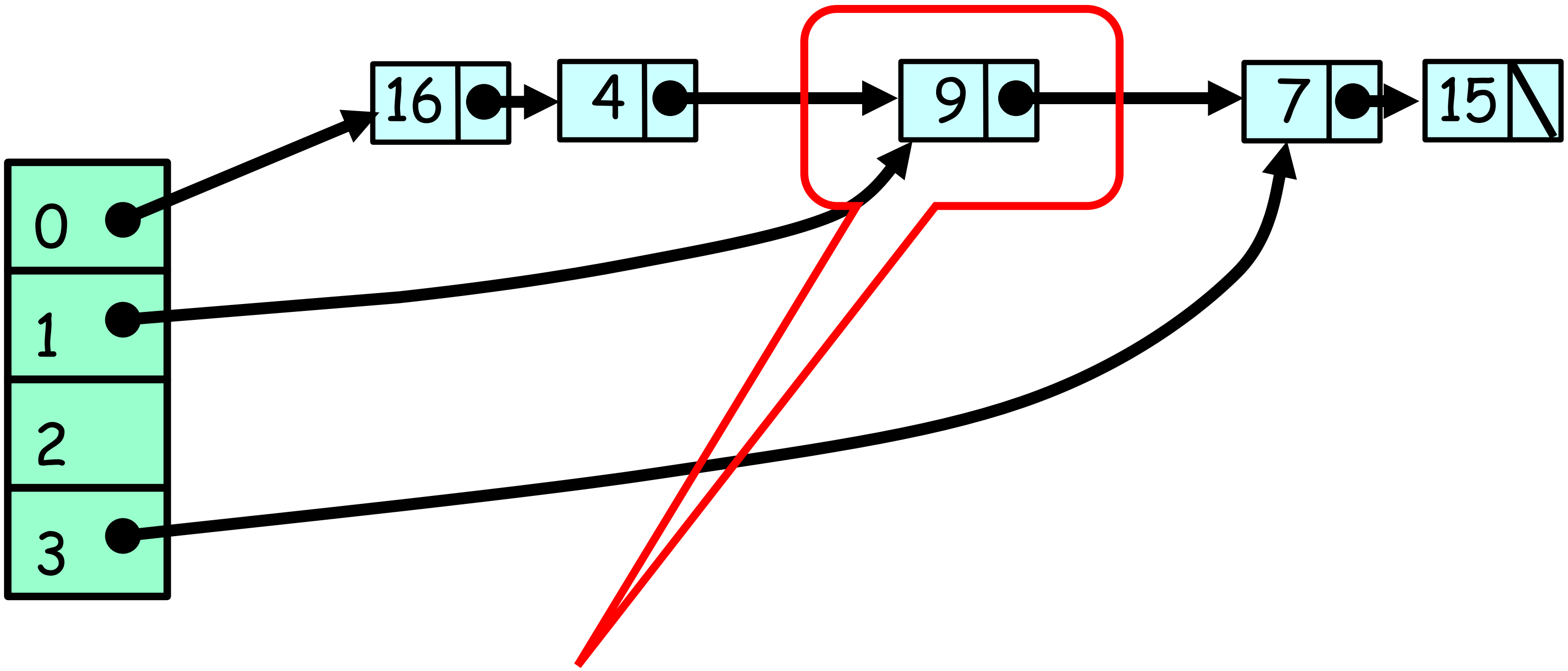
Split-order:



000 001 010 011 100 101 110 111

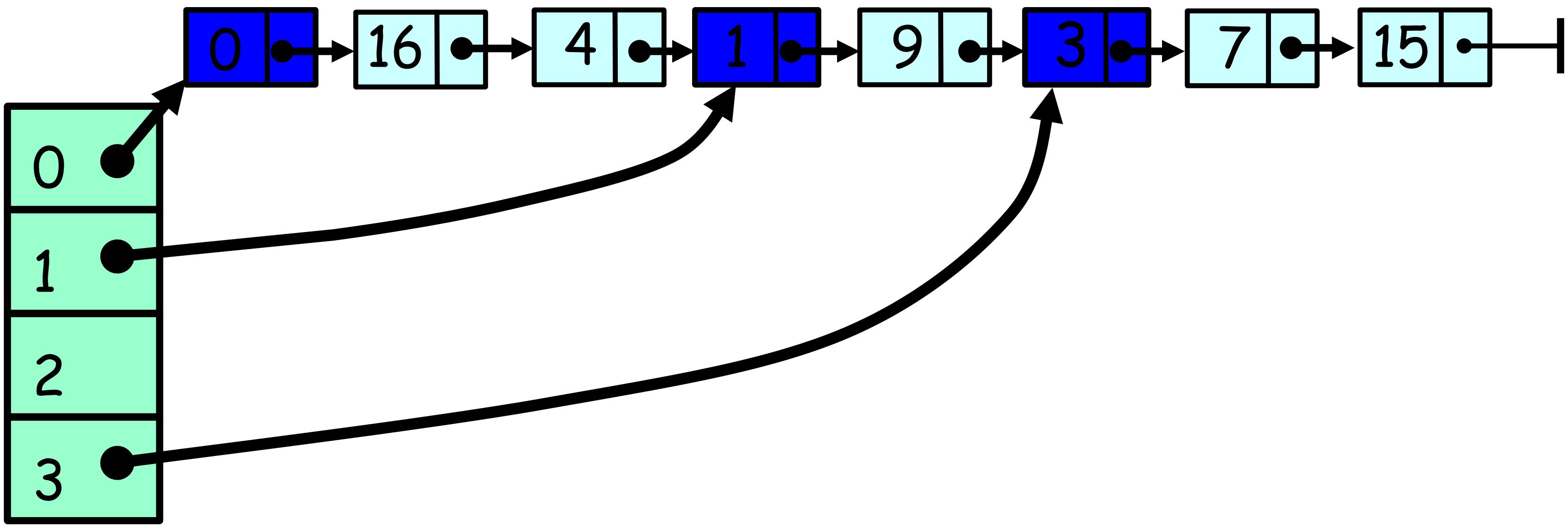
Just reverse the order of the key bits

Sentinel Nodes



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

Sentinel Nodes



Solution: use a Sentinel node for each bucket

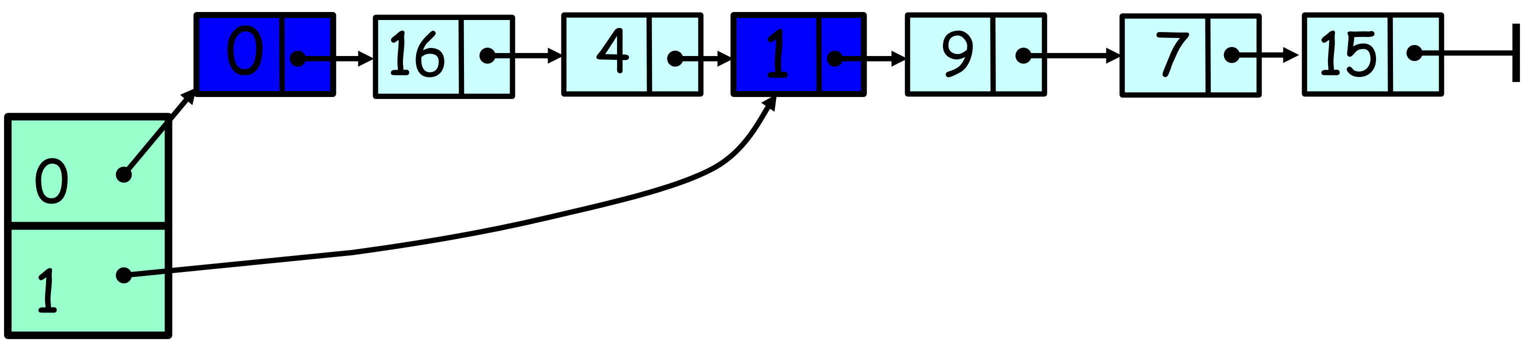
Sentinel vs Regular Keys

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

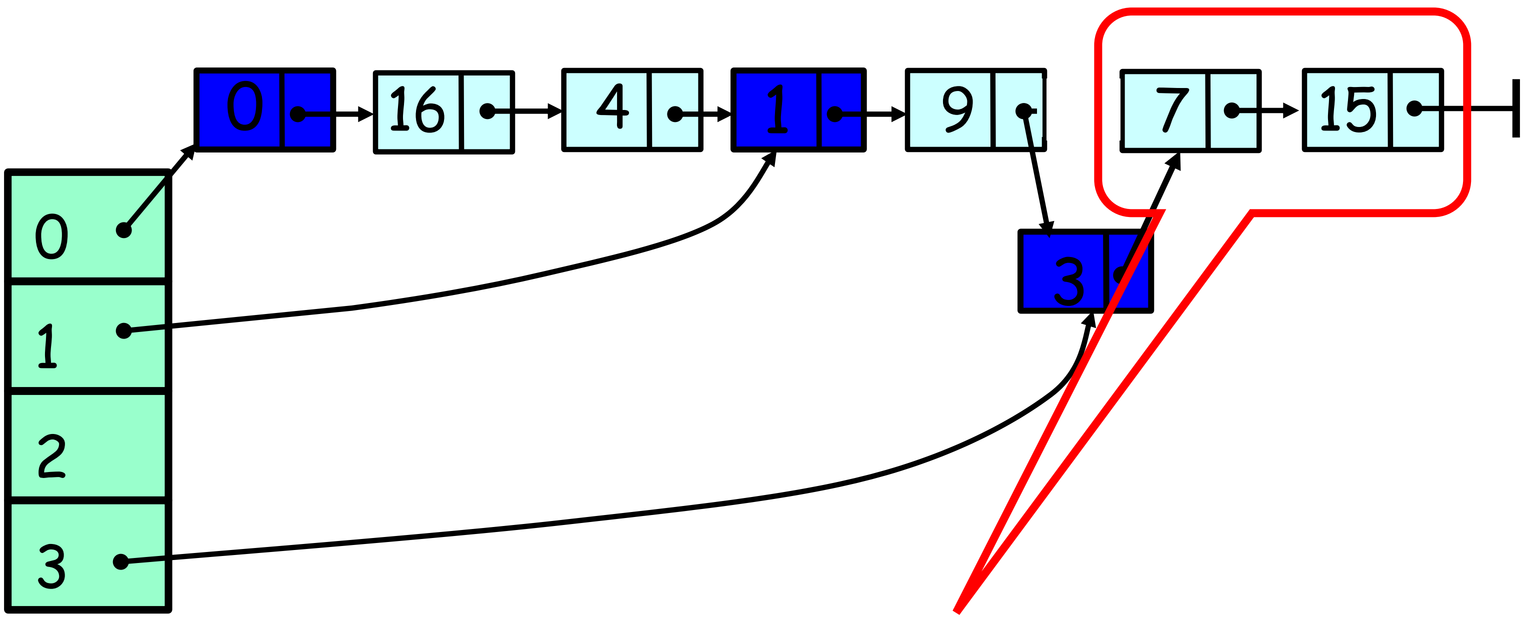
Splitting a Bucket

- We can now split a bucket
- In a lock-free manner
- Using two CAS() calls ...

Initialization of Buckets



Initialization of Buckets

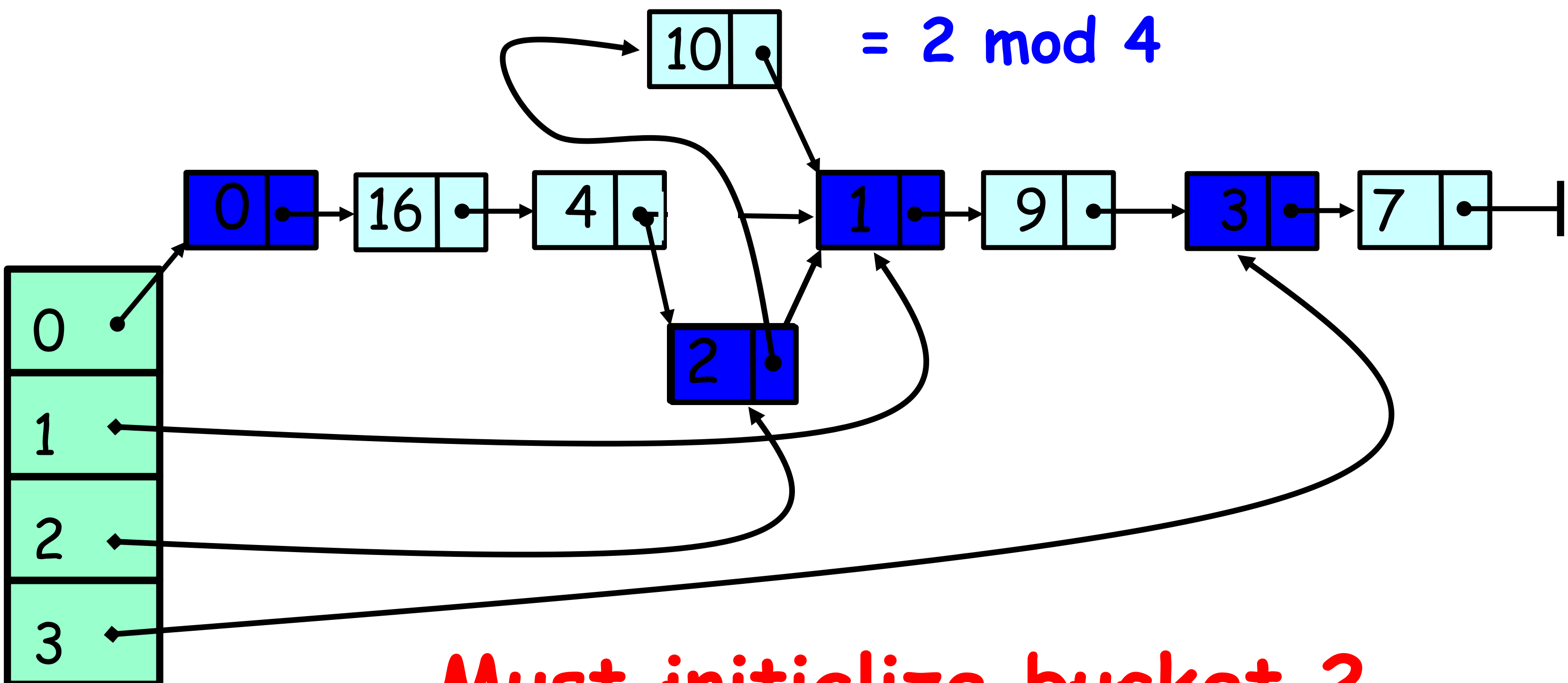


Need to initialize bucket 3 to split bucket 1

3 in list but not connected to bucket yet

Now 3 points to sentinel – bucket has been split

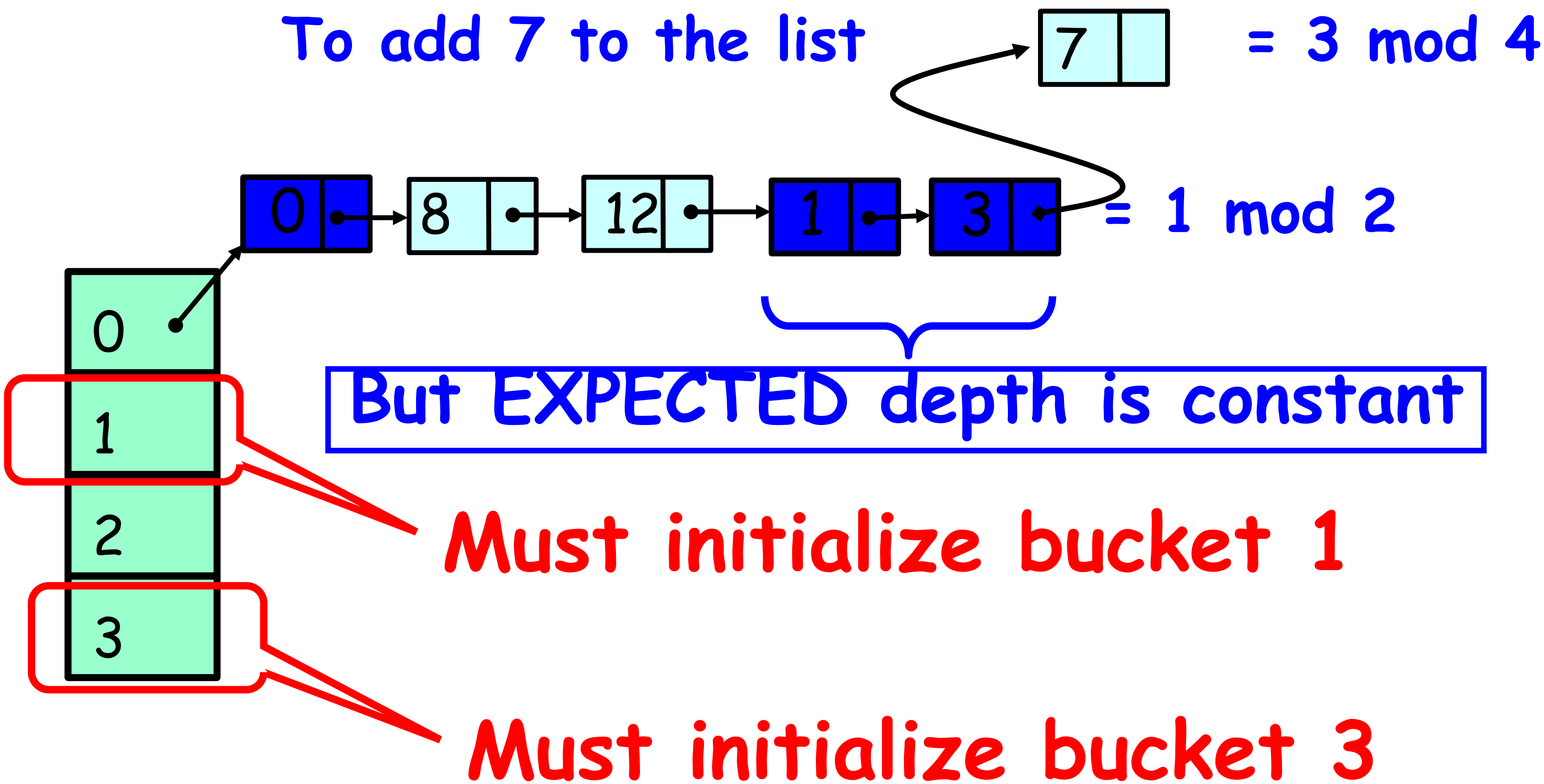
Adding 10



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
   LockFreeList(LockFreeList parent,
                int key) {...};
}
```

Change: add takes key argument

Lock-Free List

Inserts sentinel with key if not already present …

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

Lock-Free List

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

```
                          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();
    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(1);
    setSize = new AtomicInteger(0);
  }
```

In practice, want something that grows dynamically

Fields

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

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

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(1);
    setSize = new AtomicInteger(0);
  }
```

Track set size so we know when to resize

Fields

Initially use 1 bucket and size is zero

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

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

Add() 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(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;
}
```

Time to resize?

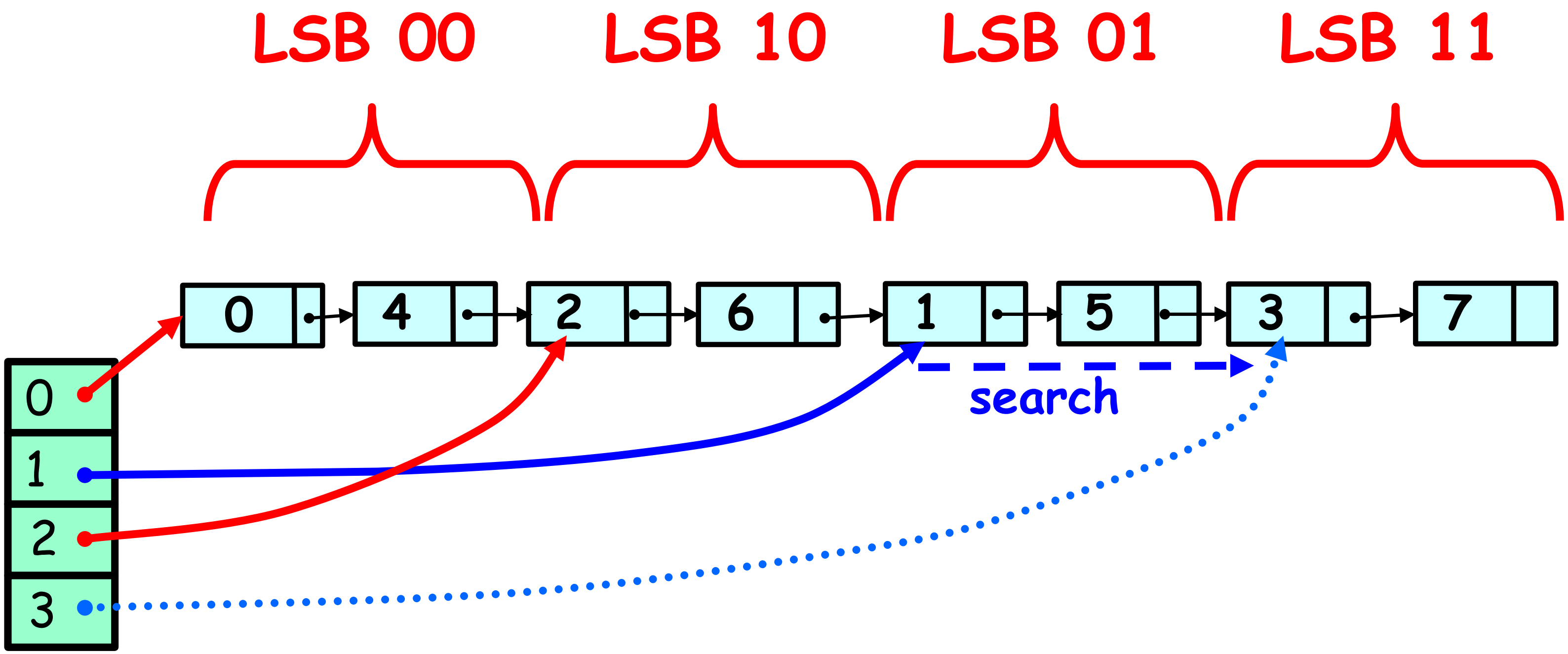
Resize

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

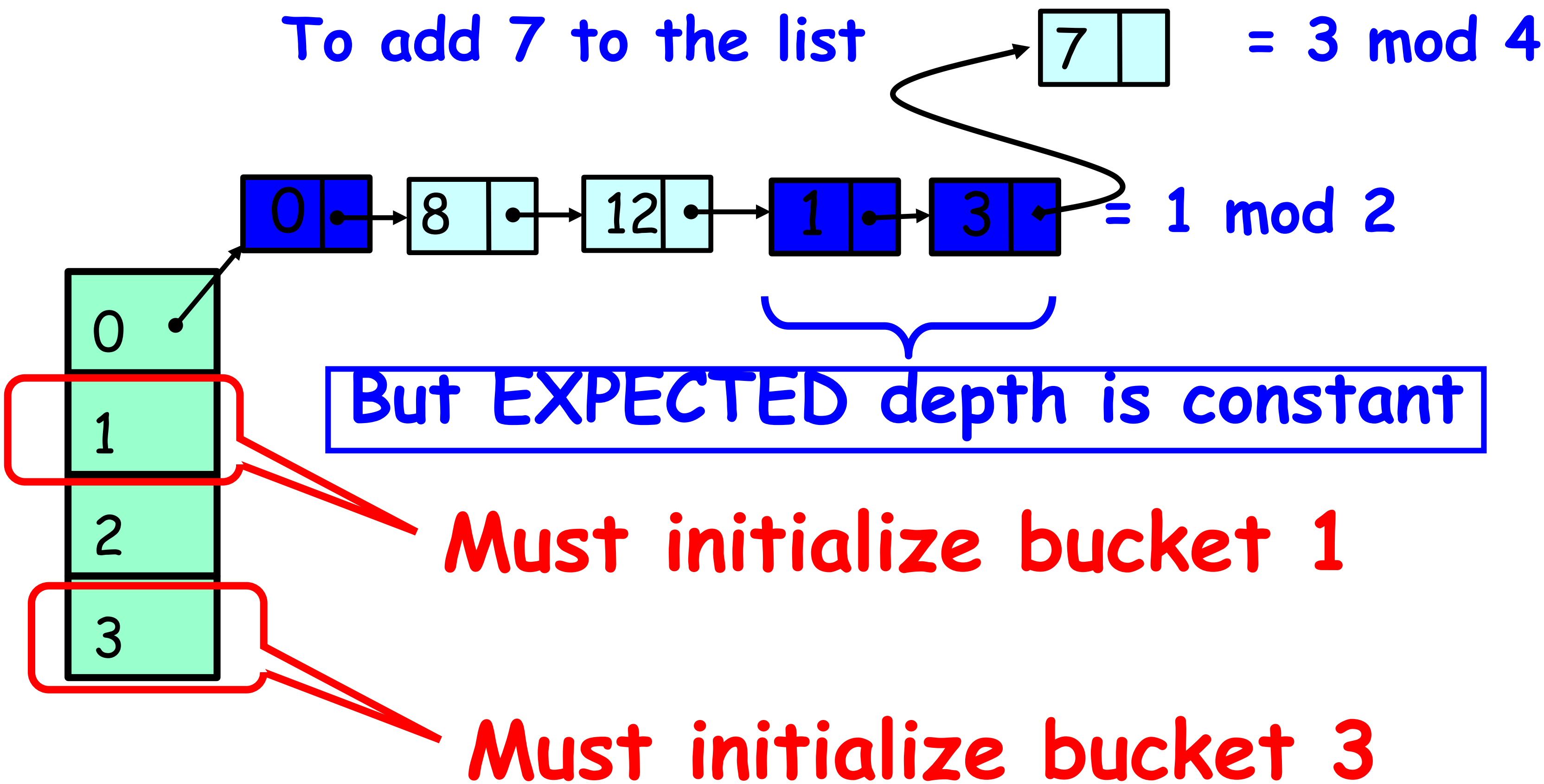
Initialize Buckets

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

Parent Provides Short Cut



Recall: Recursive Initialization



Initialize Bucket

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

Initialize Bucket

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

Find parent, recursively initialize if needed

Initialize Bucket

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

Prepare key for new sentinel

Initialize Bucket

```
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

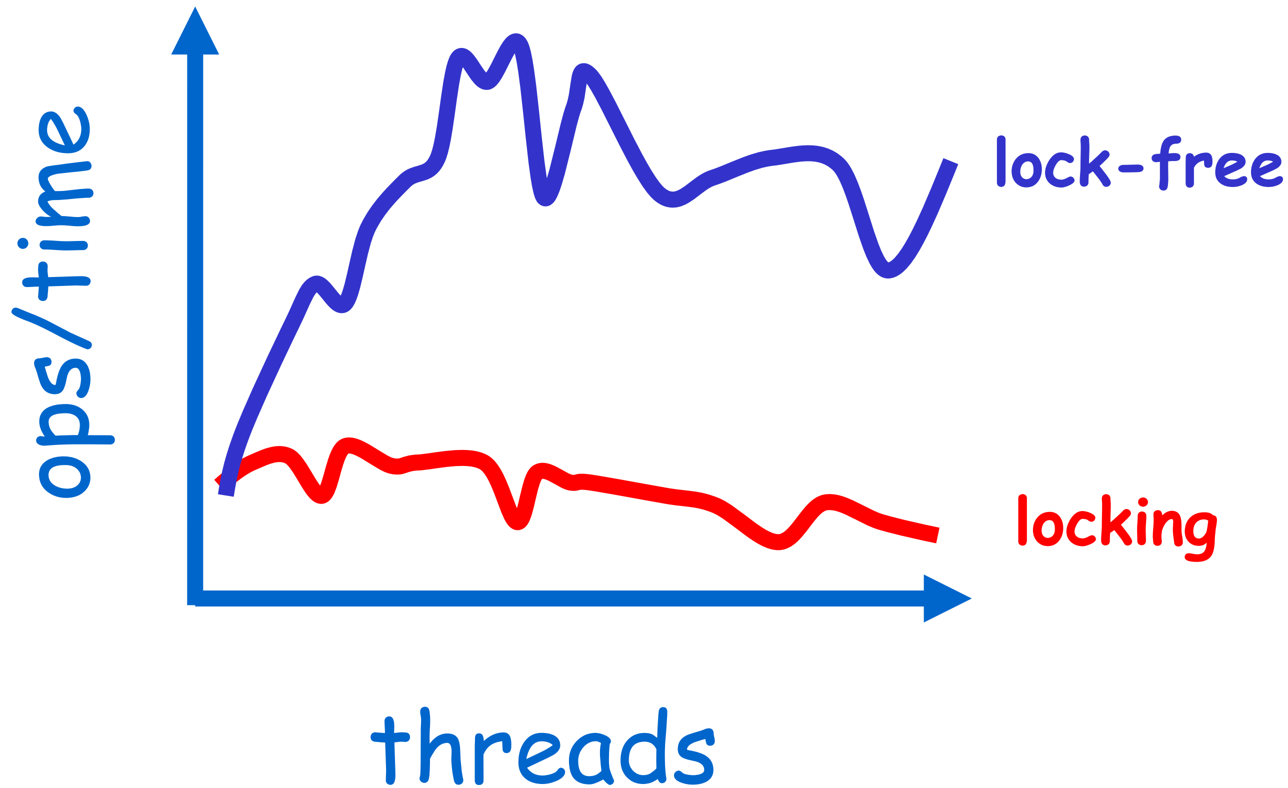
Correctness

- 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

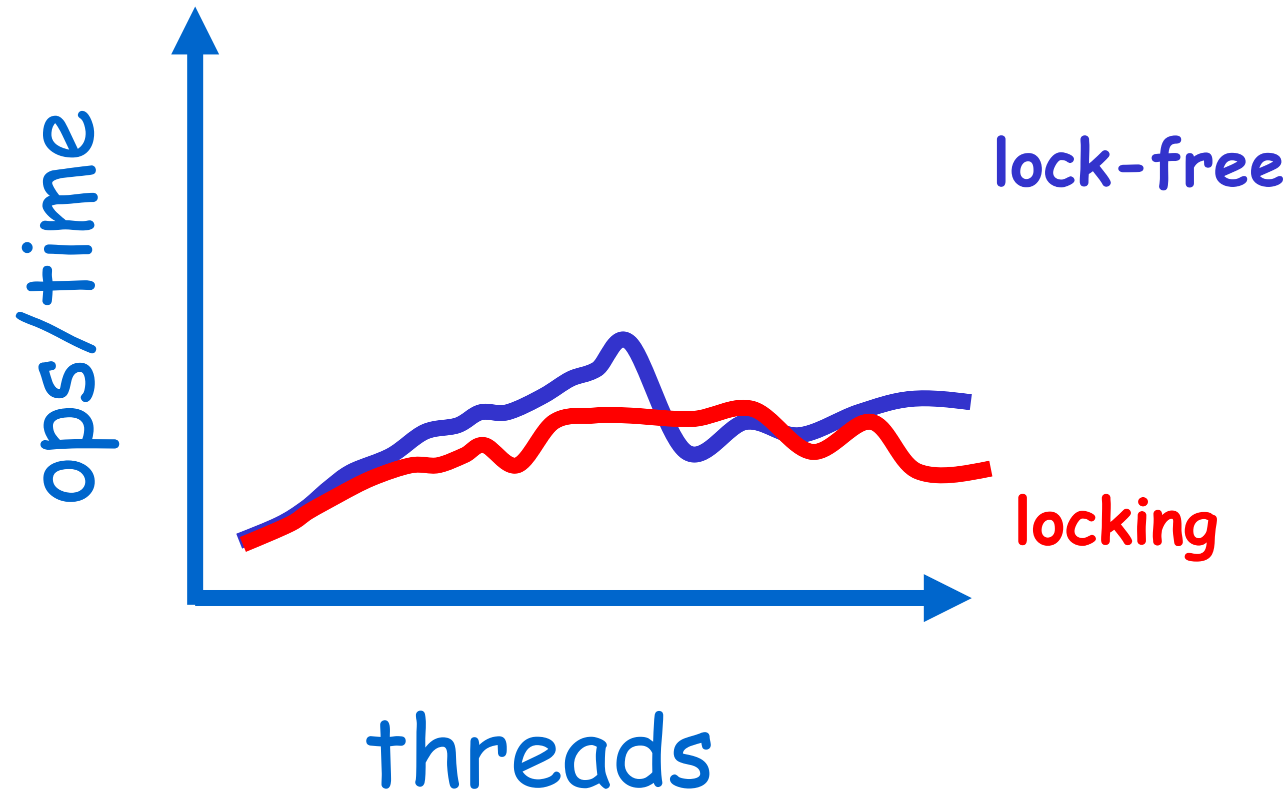
- On a 30-processor Sun Enterprise 3000
- Lock-Free vs. fine-grained (Lea) optimistic
- In a non-multiprogrammed environment
- 10^6 operations: **88%** *contains()*, **10%** *add()*, **2%** *remove()*

Work = 0



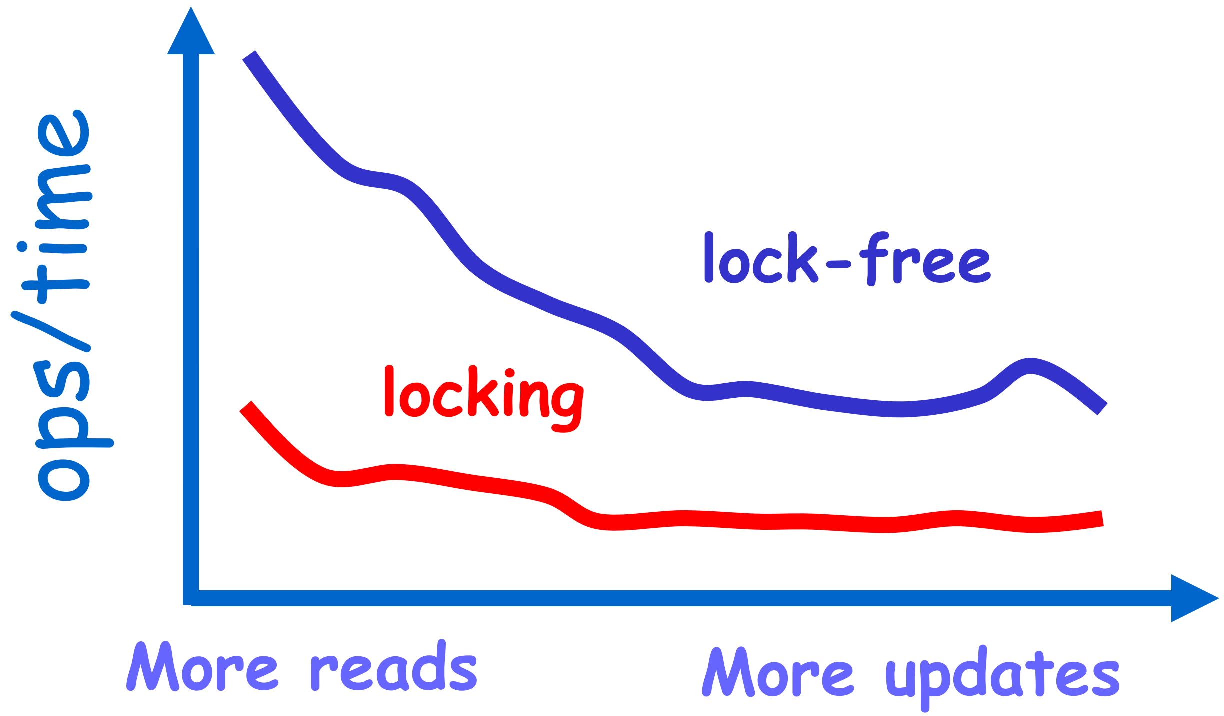
Adapted from Shalev & Shavit 2003

Work = 500



Adapted from Shalev & Shavit 2003

Varying The Mix



64 threads

Adapted from Shalev & Shavit 2003

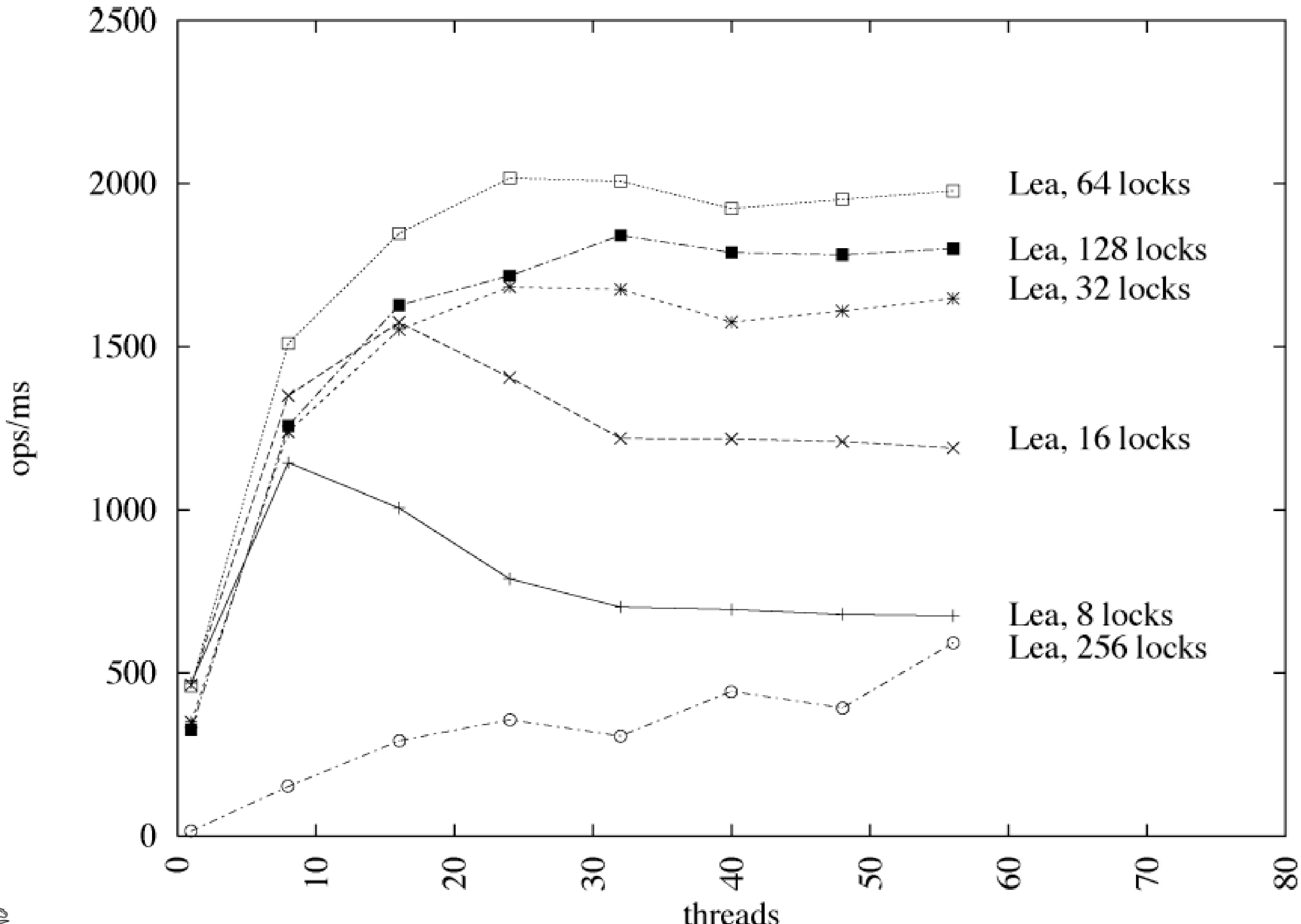
Summary

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

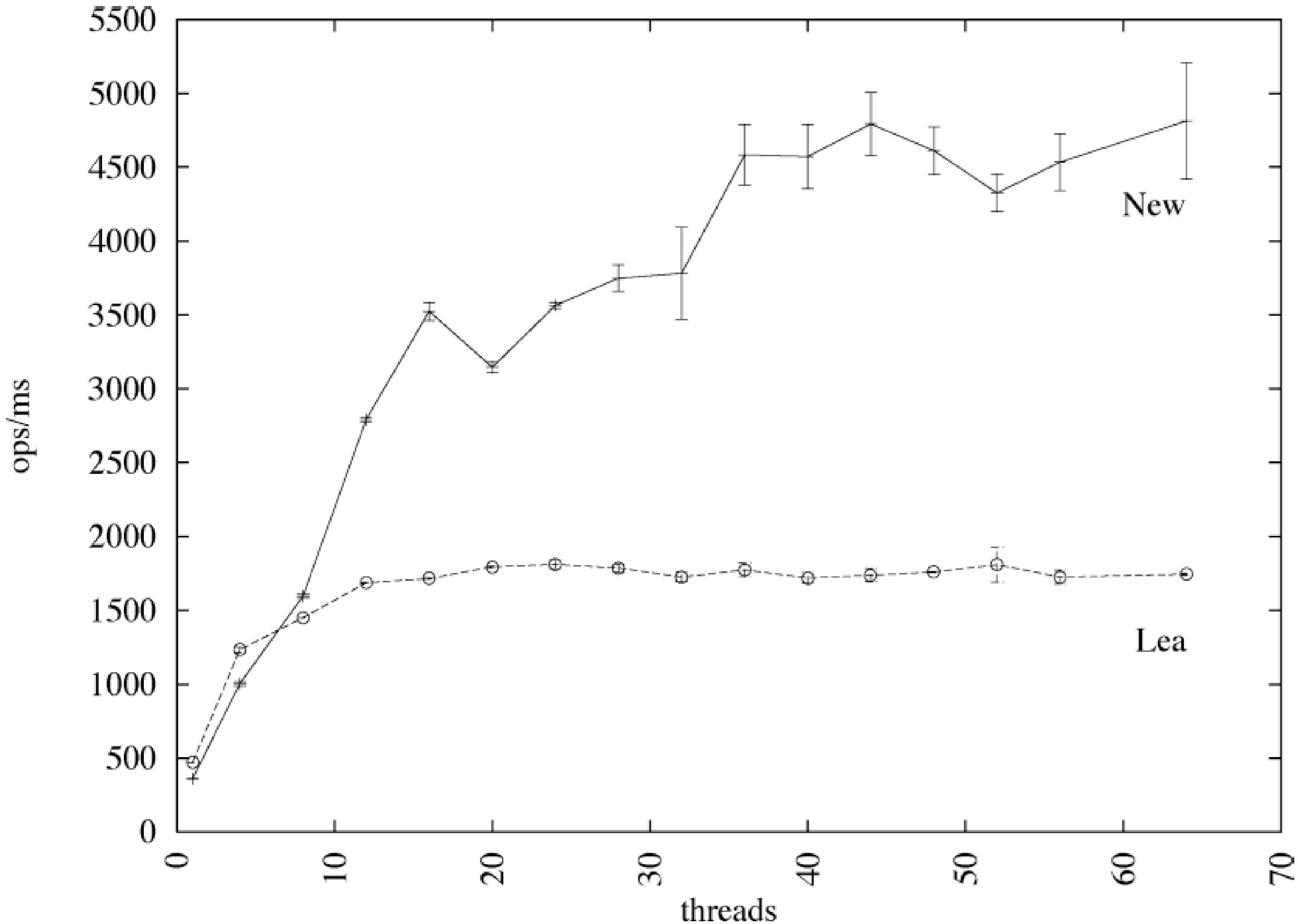
Additional Performance

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

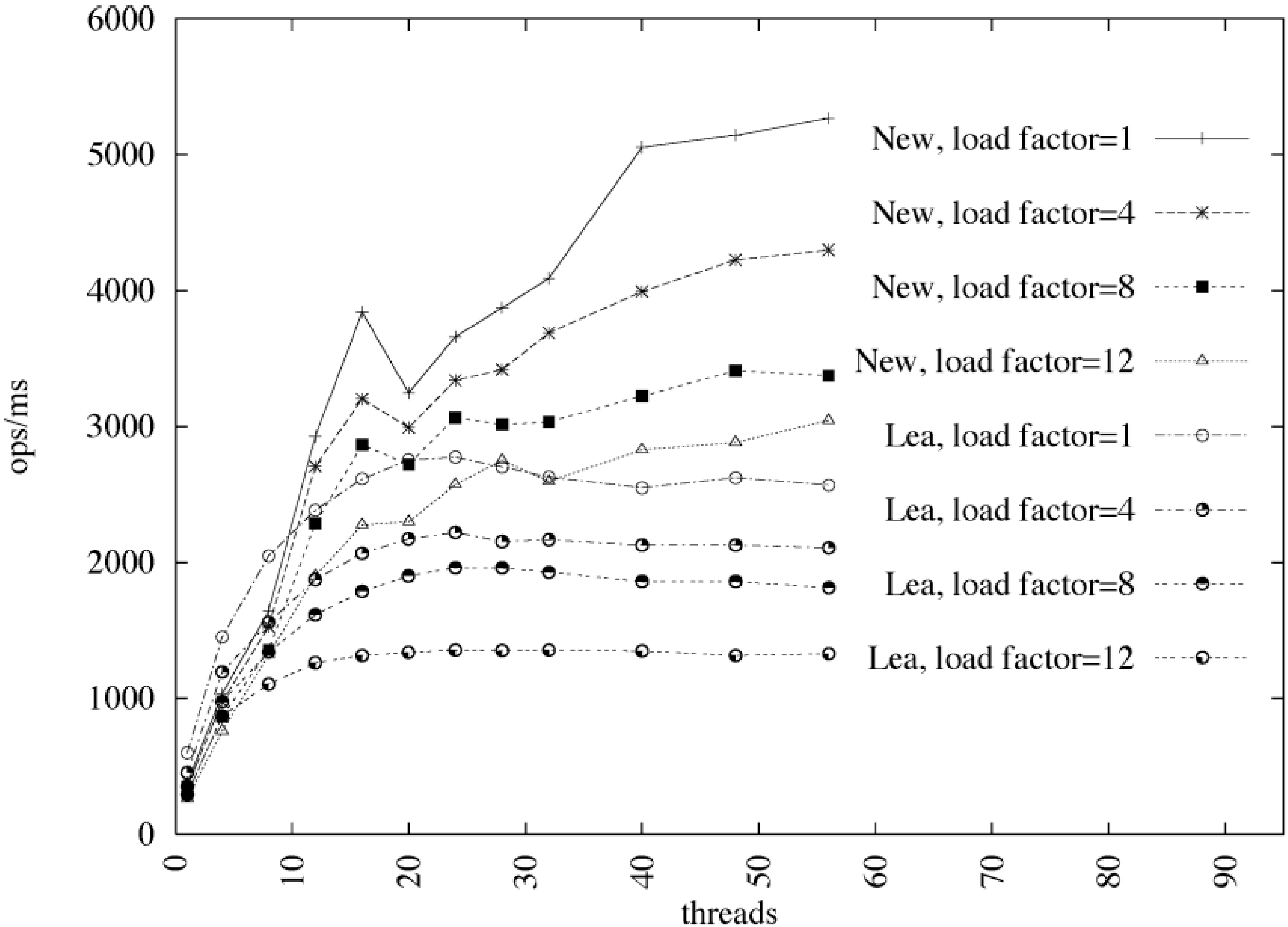
Number of Fine-Grain Locks



Lock-free vs Locks

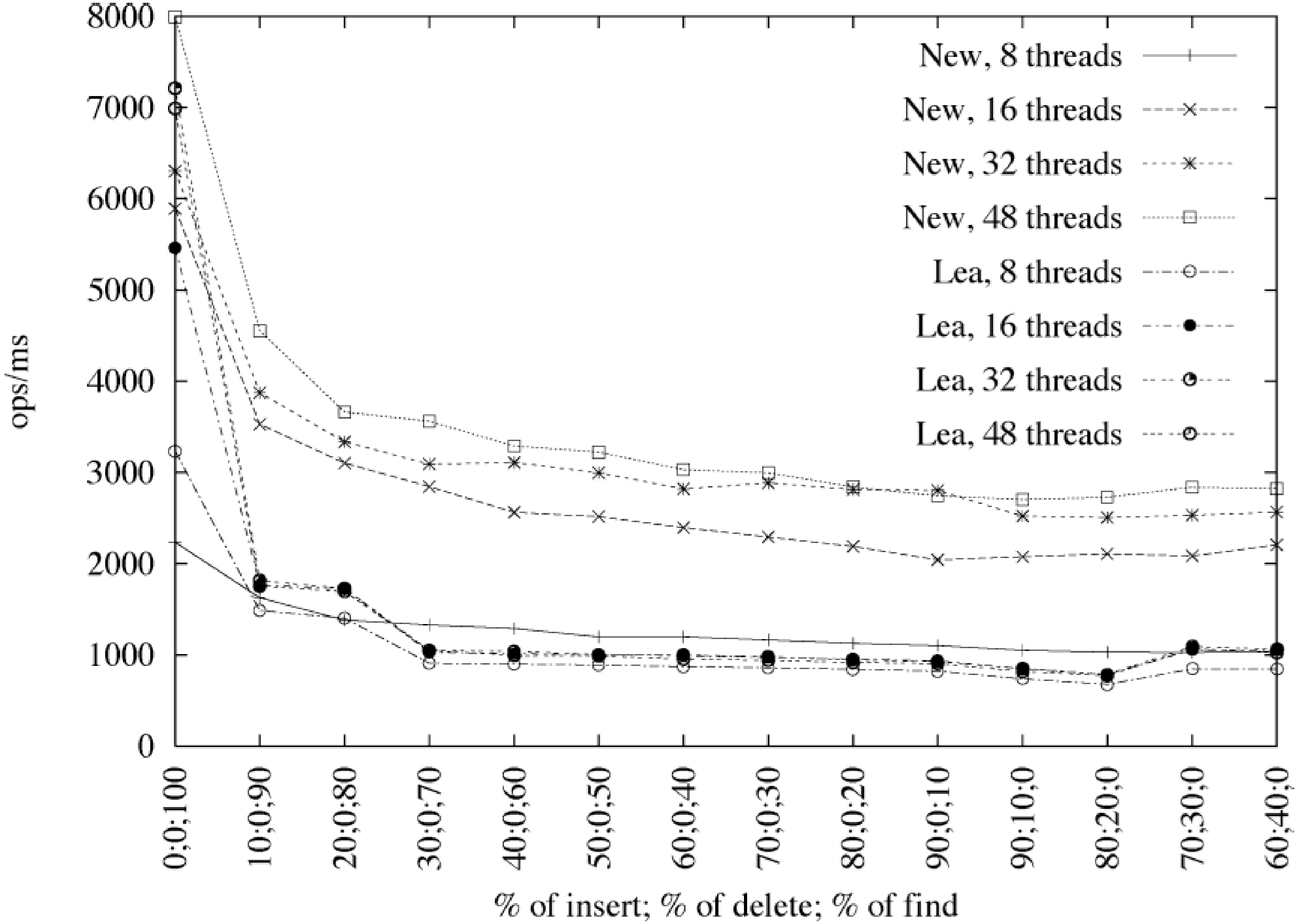


Hash Table Load Factor



(load factor = nodes per bucket)

Varying Operations



Hopscotch Hashing

Nir Shavit
Tel Aviv University

Joint work with *Maurice Herlihy* and *Moran Tzafrir*

BROWN

Our Results

- A new highly effective hash-map algorithm for Multicore Machines
- Great performance also on Uniprocessors

BROWN

Hash Tables

- *Add(), Remove(),* and *Contains()* with expected O(1) complexity
- Extensible/Resizable
- Typical hash table usage pattern: high fraction of *Contains()*, small fraction of *Add()* and *Remove()*.
- Assume universal hash function h(k) for key k.

BROWN

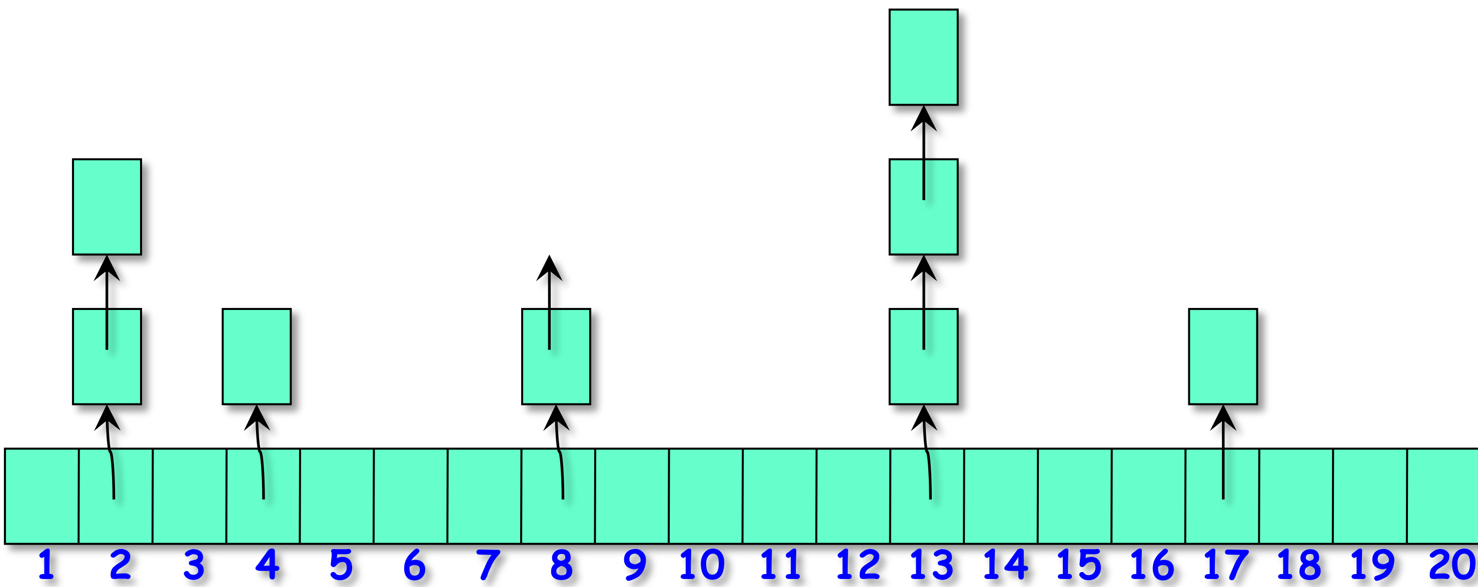
Concurrent Hash Tables

- Linearizable implementation of a Set.
- In theory, two adversaries, the scheduler and the distributer of keys.

Lets look at the state-of-the-art
Ignore resizing for now…

BROWN

Chained Hashing [Luhn]



Add(x) → if *!Contains(x)* then
Malloc(x) and link in bucket *h(x)*

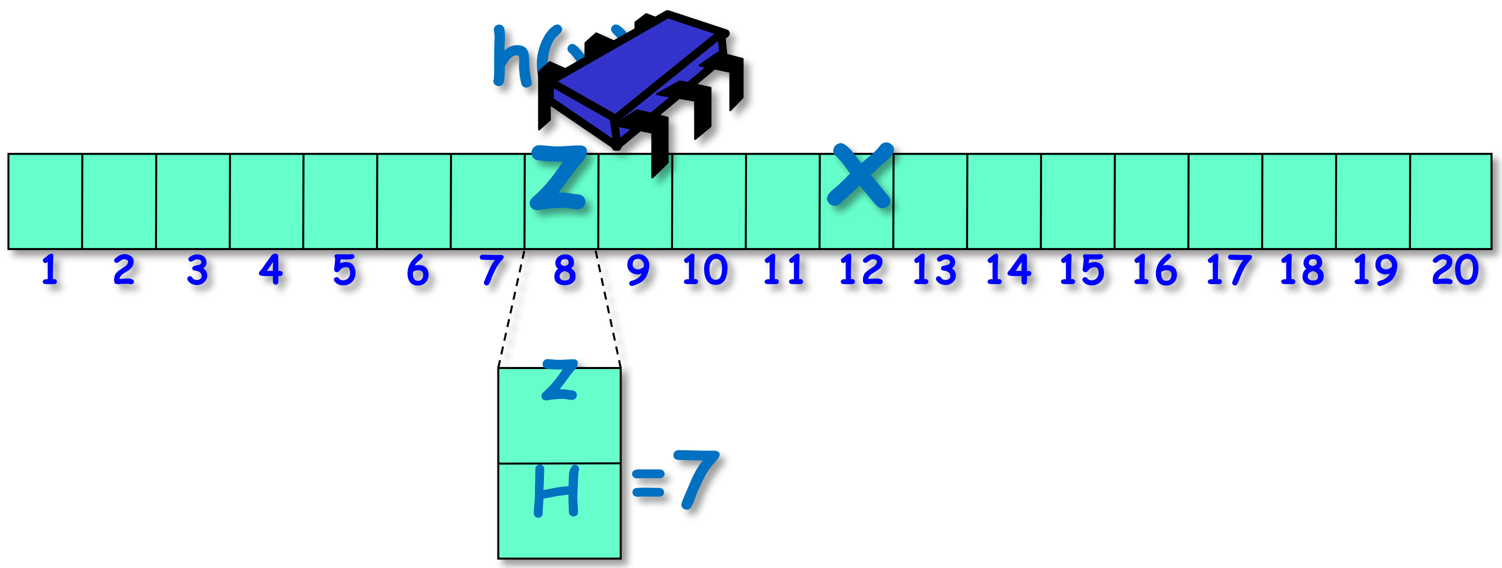
Chained Hashing

- N buckets, M items, Uniform h
- O(1) items expected in bucket (expected ~2.1 items) [Knuth]
- *Add(), Remove(), Contains()* in Expected O(1)

Chained Hashing

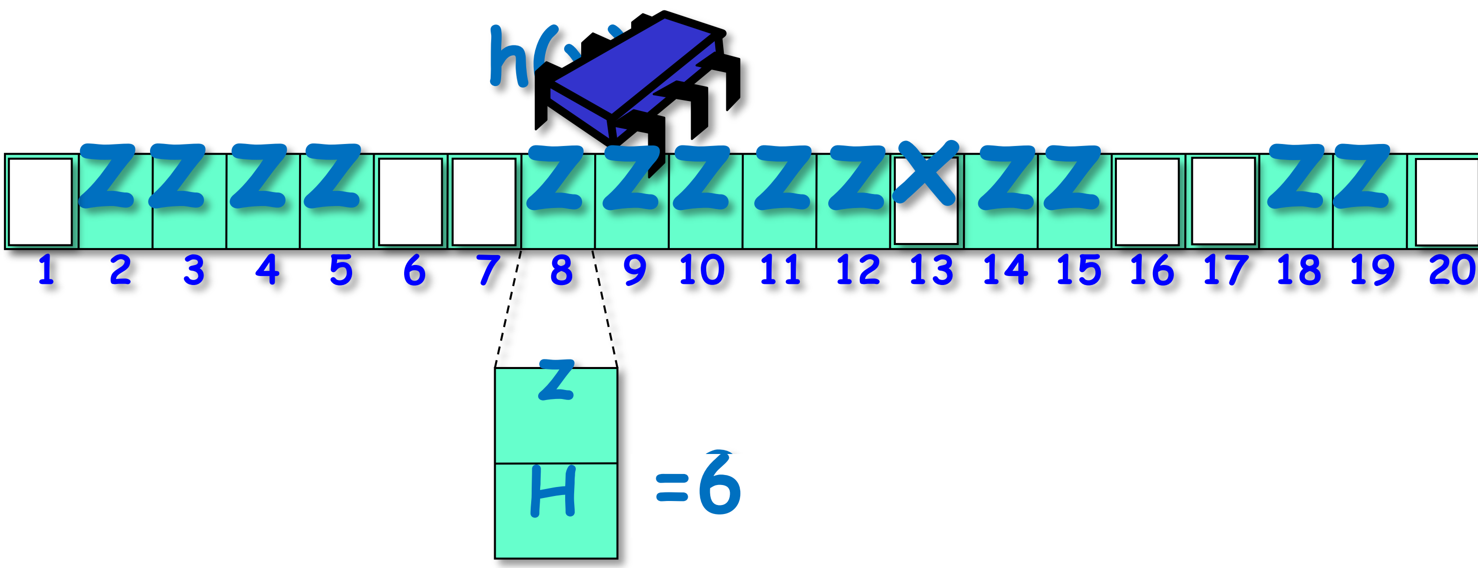
- Advantages:
  - retains good performance as table density (M/N) increases → less resizing
- Disadvantages:
  - dynamic memory allocation
  - extra full size word for pointer
  - bad cache behavior (no locality)

Linear Probing [Amdahl]



Contains(x) – search linearly from h(x) until last location H noted in bucket.

Linear Probing



Add(x) - add in first empty bucket and update its *H*.

BROWN

Linear Probing

- Open address means M <= N
- Expected items in bucket same as Chaining
- Expected distance till open slot [Knuth]:

$$\tfrac{1}{2}(1+(1/(1-M/N))^2)$$

M/N = 0.5 → search 2.5 buckets

M/N = 0.9 → search 50 buckets

BROWN

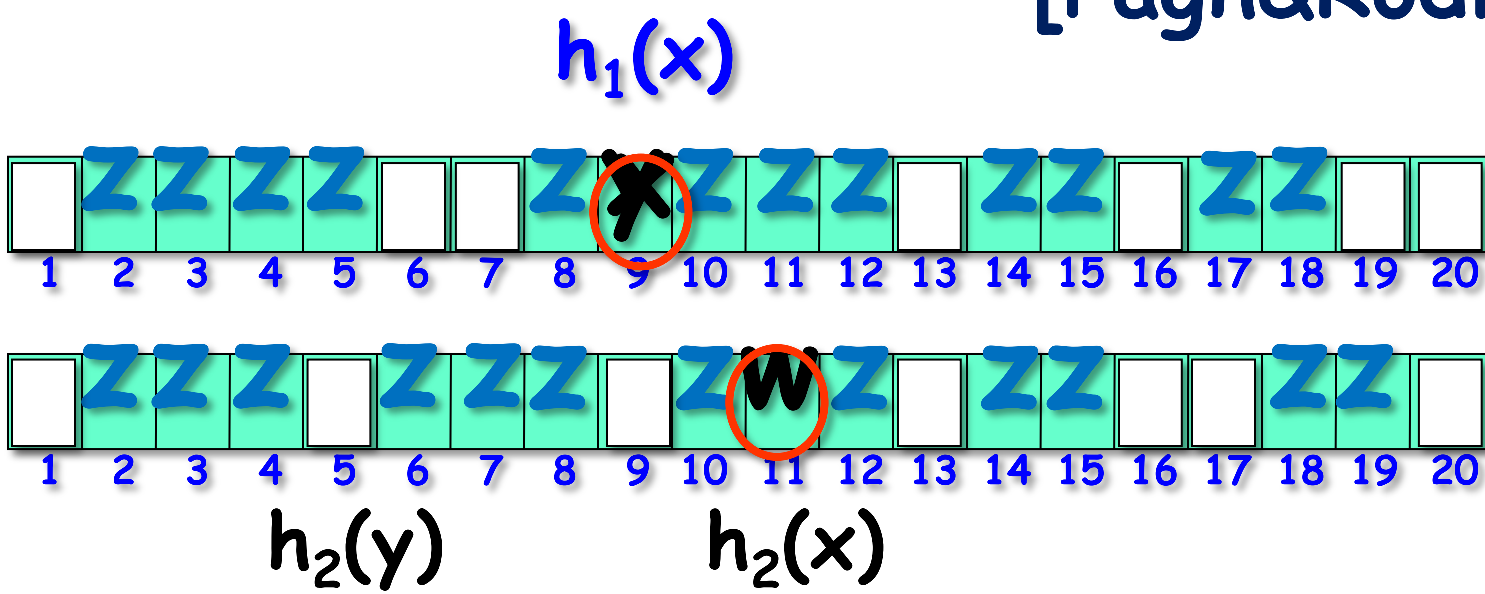
Linear Probing

- Advantages:
  - Good locality → less cache misses
- Disadvantages:
  - As M/N increases more cache misses
    - searching 10's of unrelated buckets
    - * "Clustering" of keys into neighboring buckets
  - * As computation proceeds "Contamination" by deleted items → more cache misses

BROWN

Cuckoo Hashing

[Pagh&Rodler]



Add(x) – if $h_1(x)$ and $h_2(x)$ full evict y and move it to $h_2(y) \neq h_2(x)$. Then place x in its place.

BROWN

Cuckoo Hashing

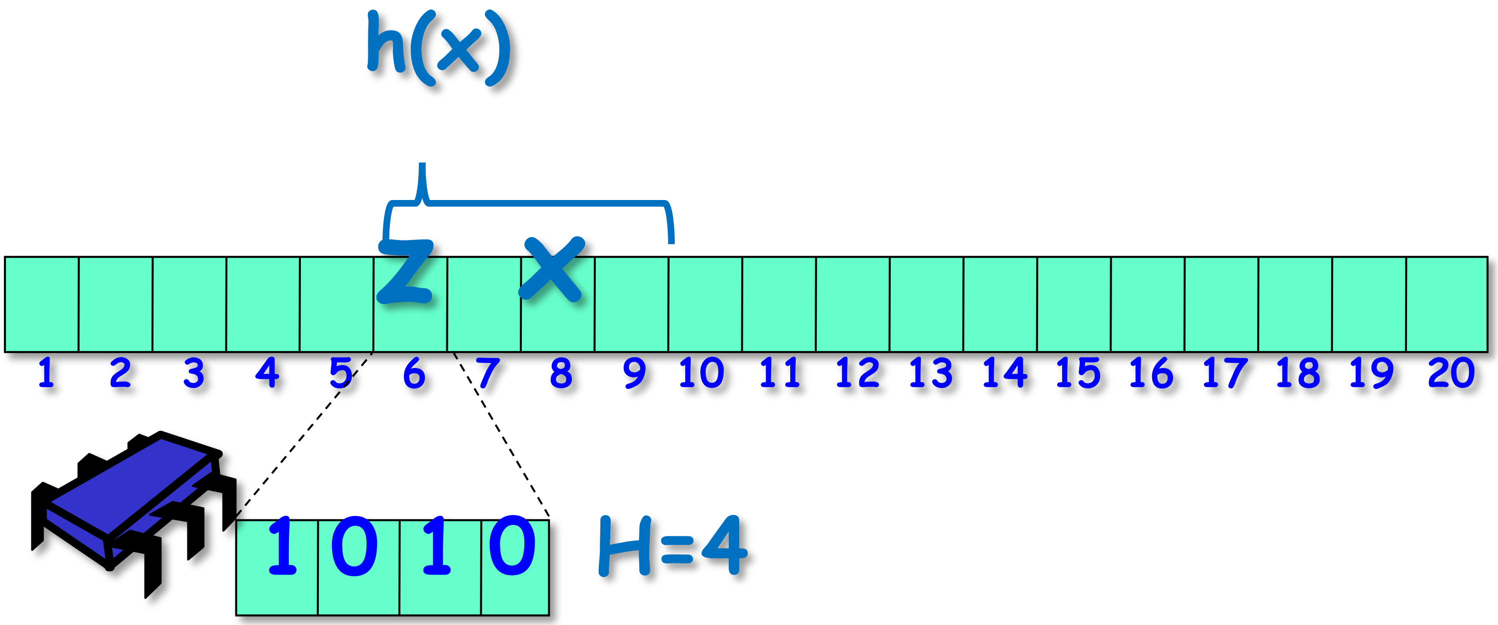
- Advantages:
  - *Contains()* : deterministic 2 buckets
  - No clustering or contamination
- Disadvantages:
  - 2 tables
  - $h_i(x)$ are complex
  - As M/N increases → relocation cycles
  - Above M/N = 0.5 Add() does not work!

Can we do better?

- Deterministic constant time *Contains()*
- With good locality
- Using simple hash hunctions

BROWN

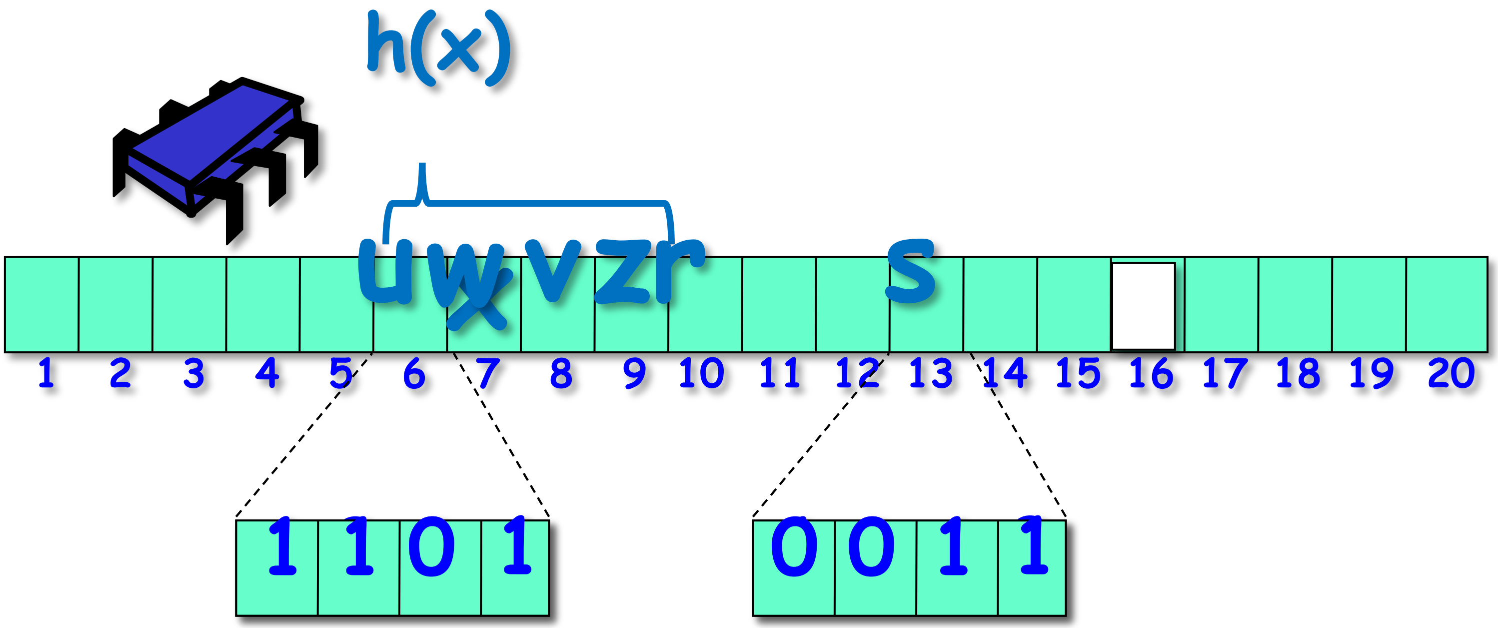
Hopscotch Hashing



Contains(x) – search in at most H buckets (the hop-range) based on hop-info bitmap. H is a constant.

BROWN

Hopscotch Hashing



Add(x) – probe linearly to find open slot. Move the empty slot via sequence of displacements into the hop-range of h(x).

BROWN

Add(x)

- Starting at $h(x) = i$, use linear probing to find an empty slot j.
- If j is within $H-1$ of i, place x and return.
- Otherwise,
  - Find y with $h(y) = k$ within $H-1$ left of j. Displacing y to j creates a new empty slot k closer to i. Repeat.
  - If no such item exists, or if the bucket i already contains H items, *Resize()*.

BROWN

Hopscotch Hashing

- *Contains():*
- Max number of items H is a constant as in Cuckoo.
- Expected items in bucket ~2.1 same as Chaining, <u>but they have a good chance of sitting in the same cache line!</u>

Hopscotch Hashing

- *Add()*: Expected distance till open slot same as in linear probing
- What are the chances of a Resize() because more than H items are hashed to a bucket?

  Lemma [following Knuth] : same as num items in chained bucket being greater than H, which is 1/H!

BROWN

Hopscotch Hashing

- So what are the chances of such a *Resize()*?
- On modern machines max bitmap size H=32, so

$$1/H! = 1/32! < 10^{-35}$$

- So start with 4, then increase to 8, 16, 32. Chances of overflow causing a *Resize()* decrease exponentially.

Hopscotch Hashing

- Advantages:
  - Good locality and cache behavior
  - Good performance as table density (M/N) increases → less resizing
  - Withstands clustering and contamination
  - Pay price in *Add()* not in frequent *Contains()*

Hopscotch Hashing

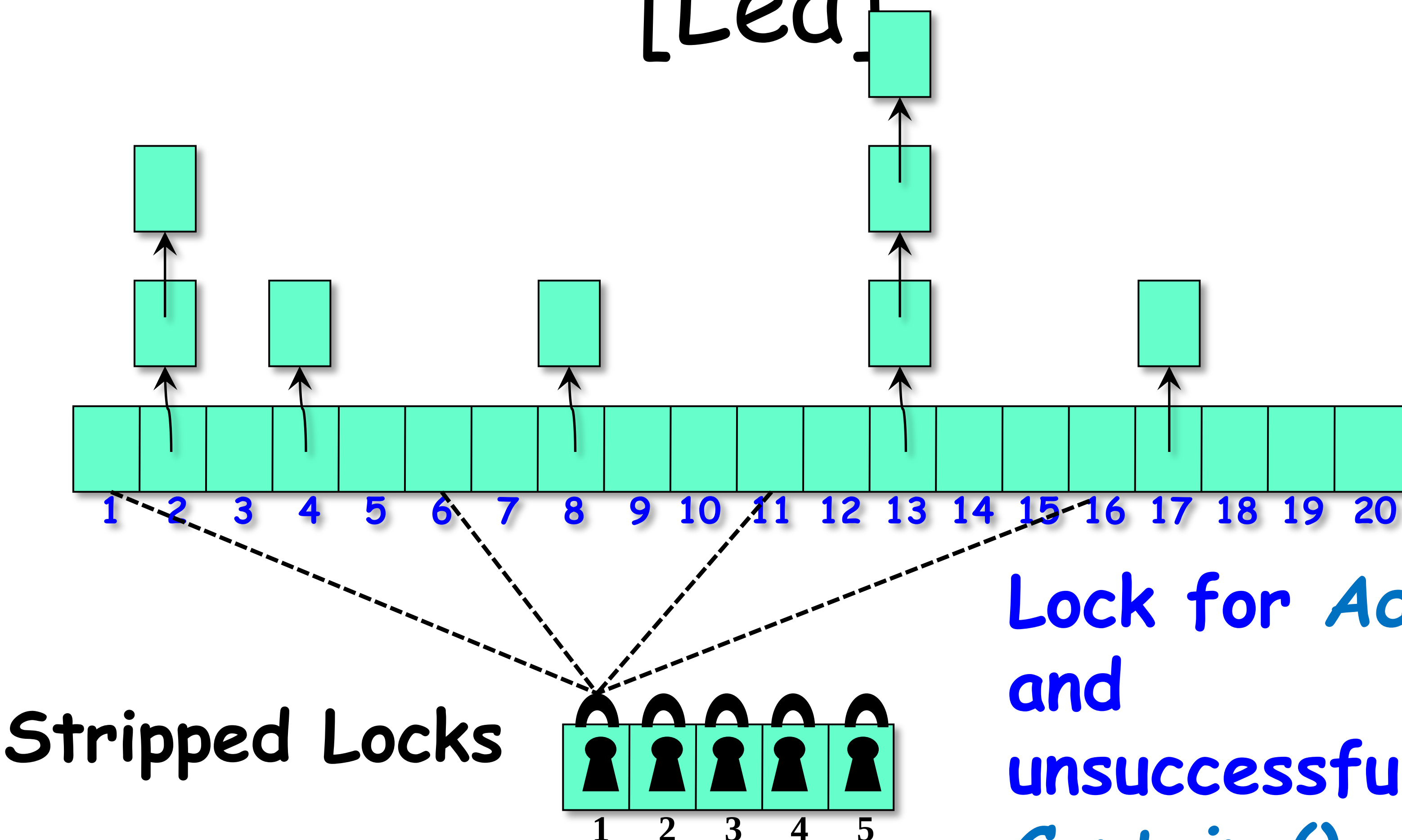
- Disadvantages:
  - As in Linear probing *Add()* may have to search 10s of buckets
  - But we pay price in *Add()* not in frequent *Contains()*

Concurrent Hash Tables

- State-of-the-art is Lea's ConcurrentHashMap from Java.util.concur: lock-based chaining
- Also lock-free split-ordered incremental chaining algorithm [Shalev&Shavit]
- Non-resizable lock-free linear probing [Purcell&Harris]
- Resizable Lock-based Cuckoo [Herlihy,Shavit, Zafrir]

BROWN

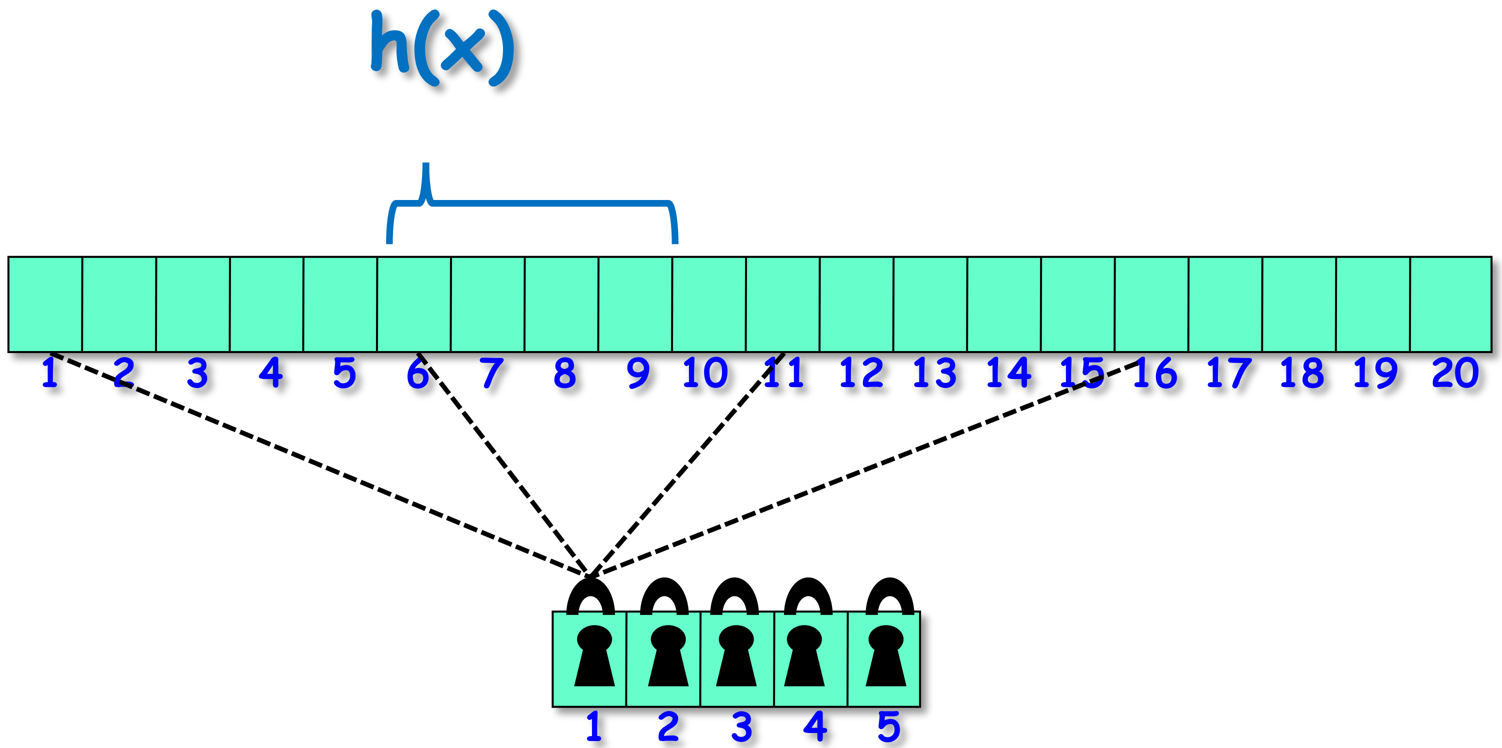
Concurrent Chained Hashing [Lea]



Lock for *Add()* and unsuccessful *Contains()*

BROWN

Concurrent Hopscotch

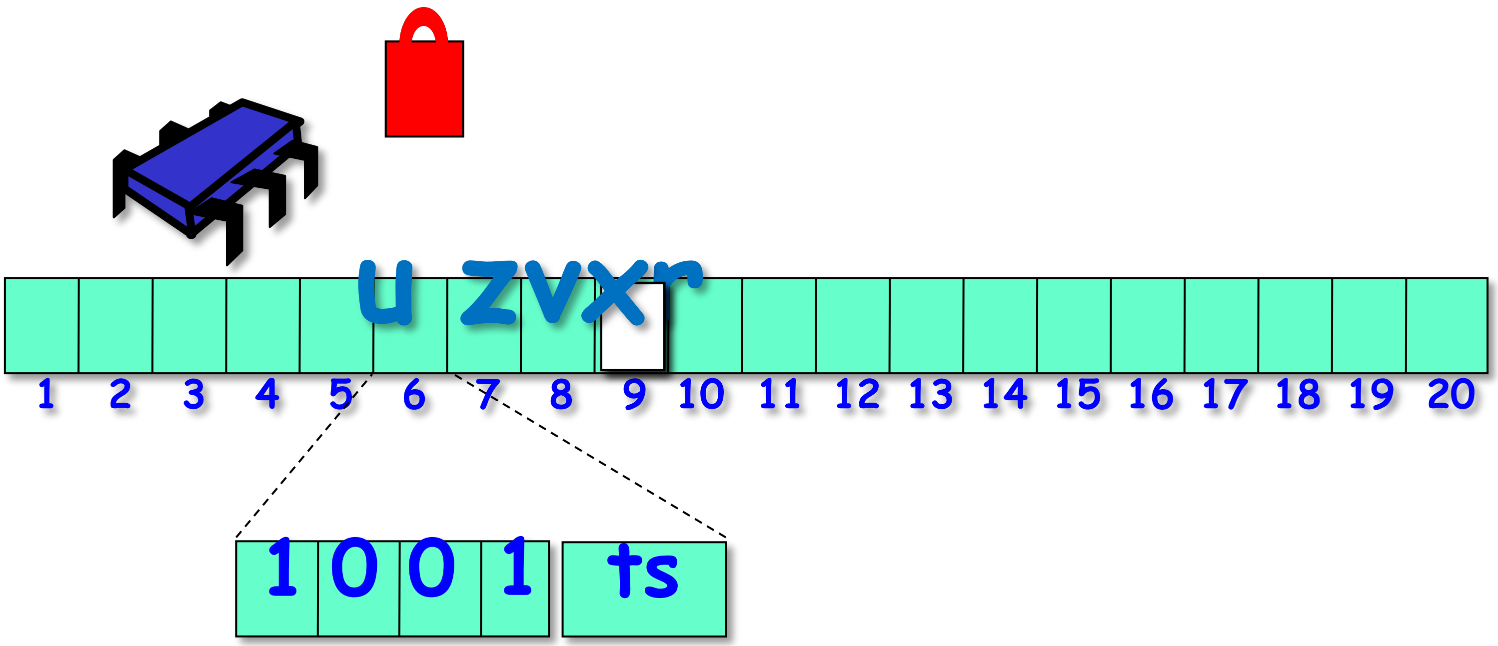


Stripped locking same as Lea

Contains() is wait-free

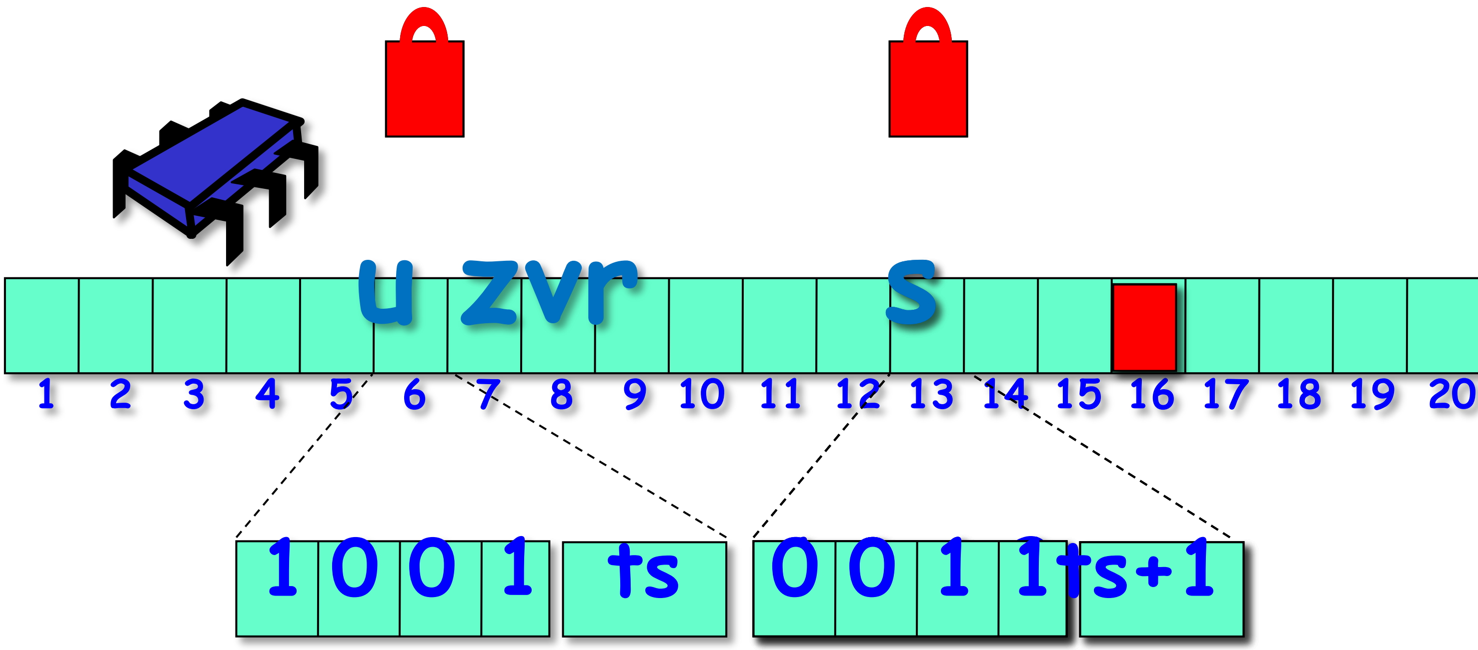
BROWN

Concurrent Hopscotch



Add(x) **– lock bucket, mark empty slot using CAS, add x erasing mark**

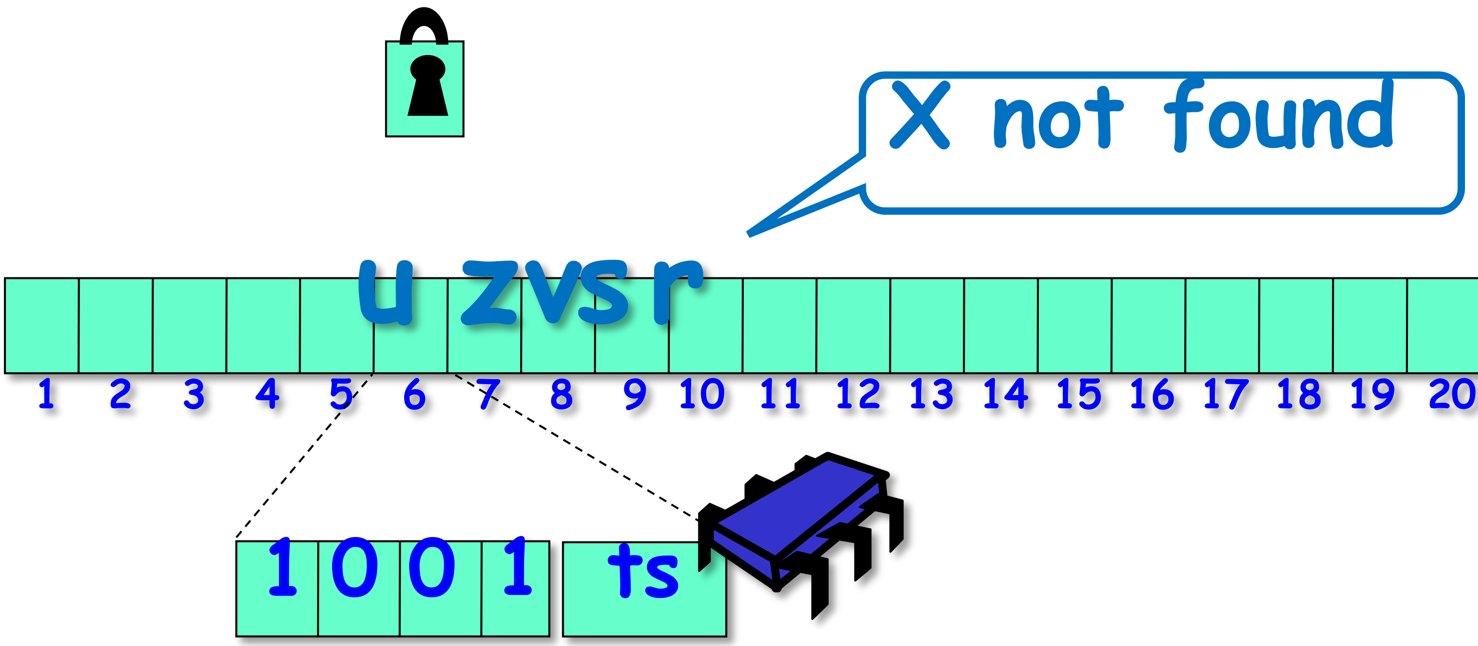
Concurrent Hopscotch



Add(x) – lock bucket, mark empty slot using CAS, lock bucket and update timestamp of bucket being displaced before erasing old value

BROWN

Concurrent Hopscotch



Wait-free Contains(x) – read ts, hop-info, goto marked buckets, if no x compare ts, if diff repeat, after k attempts search all H buckets

BROWN

Performance

- We ran a series of benchmarks on state of the art multicores and uniprocessors:
  - Sun 64 way Niagara II, and
  - Sun 128 way Miramba server and,
  - Intel 3GHz Xeon
- Grain of Salt: we used standard microbenchmarks, not real application data

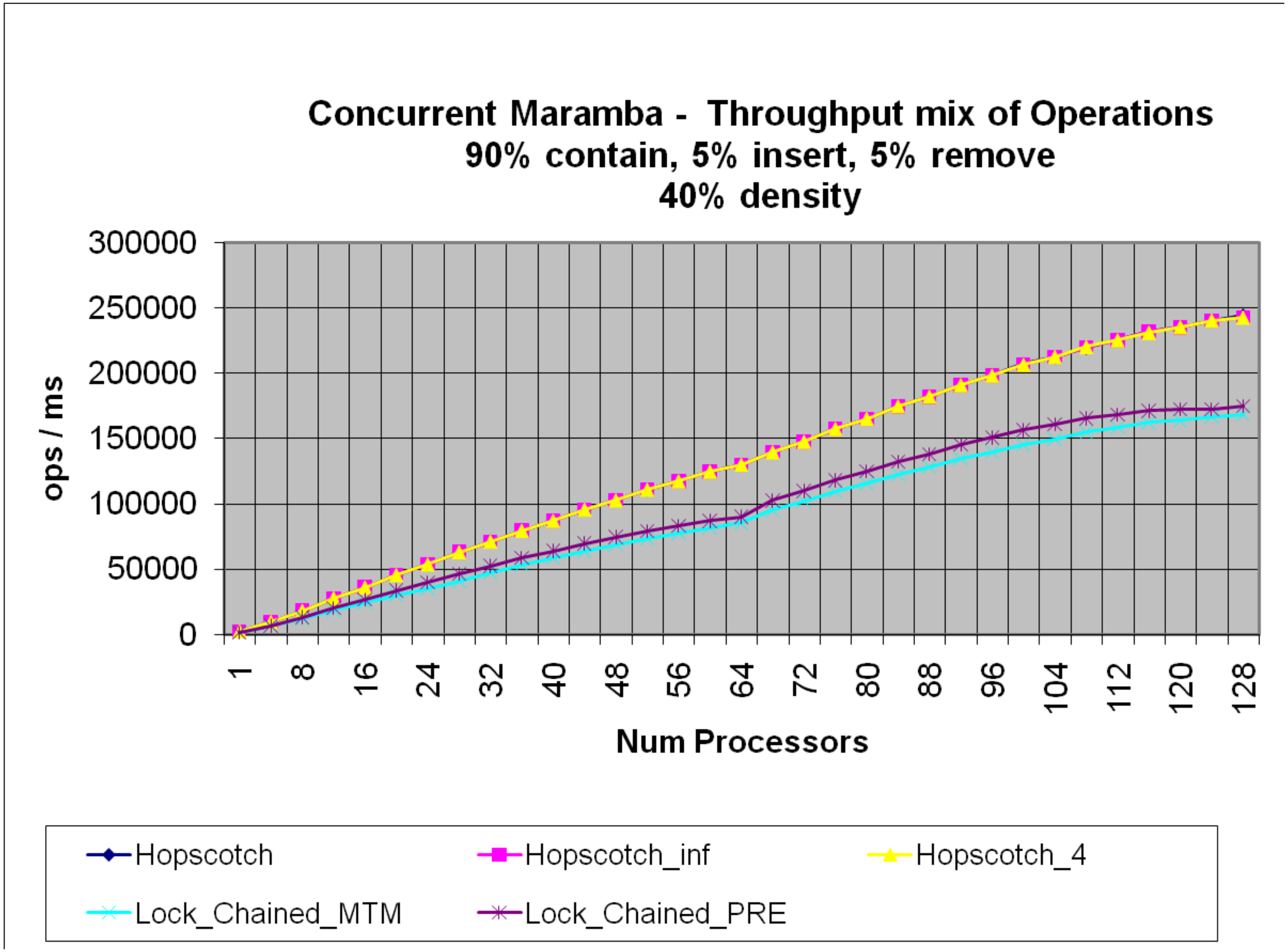
BROWN

Benchmarking

- We used the same locking structure for concurrent algs
- We show graphs with pre-allocated memory to eliminate effects of memory management

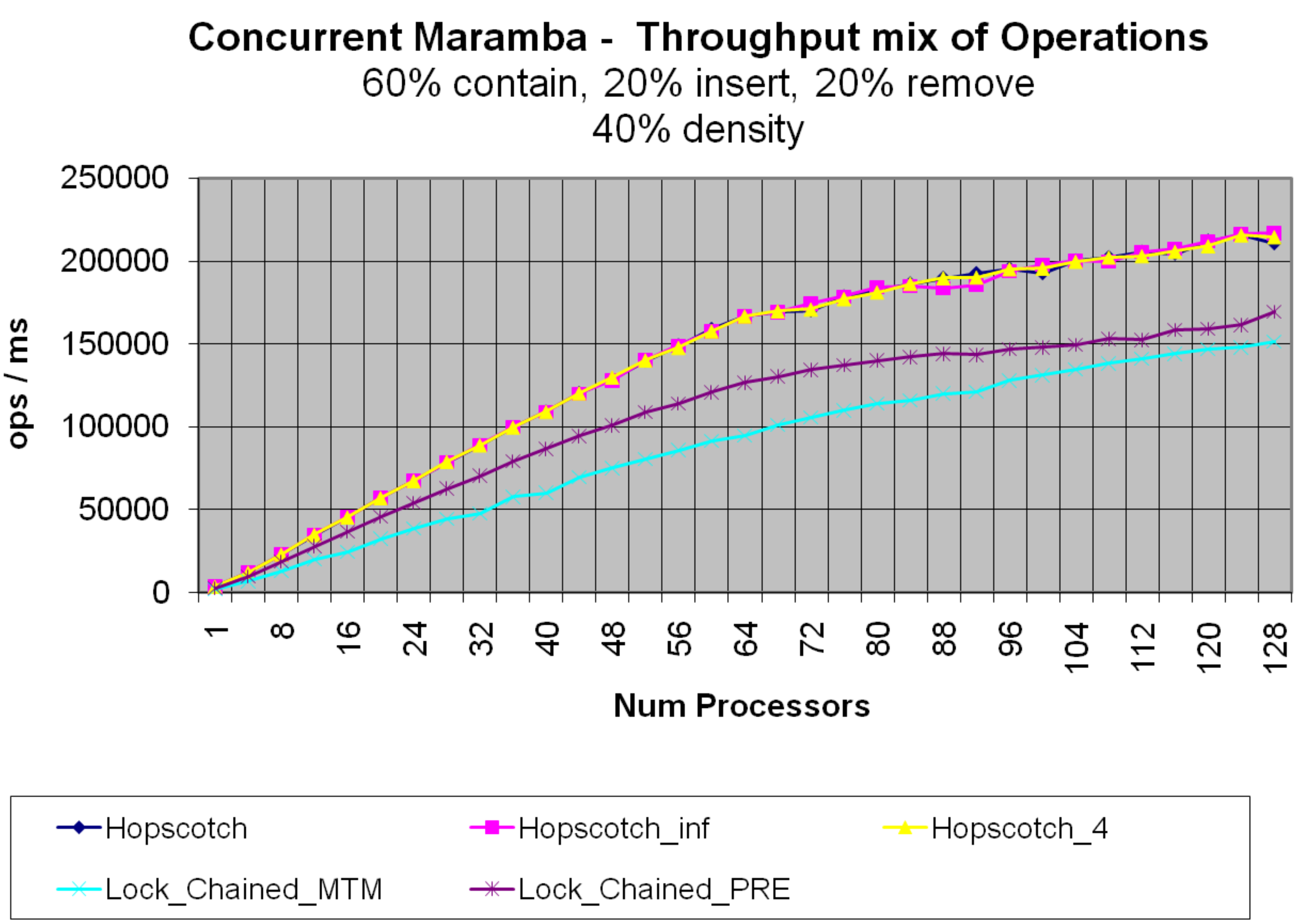
BROWN

Maramba Concurrent



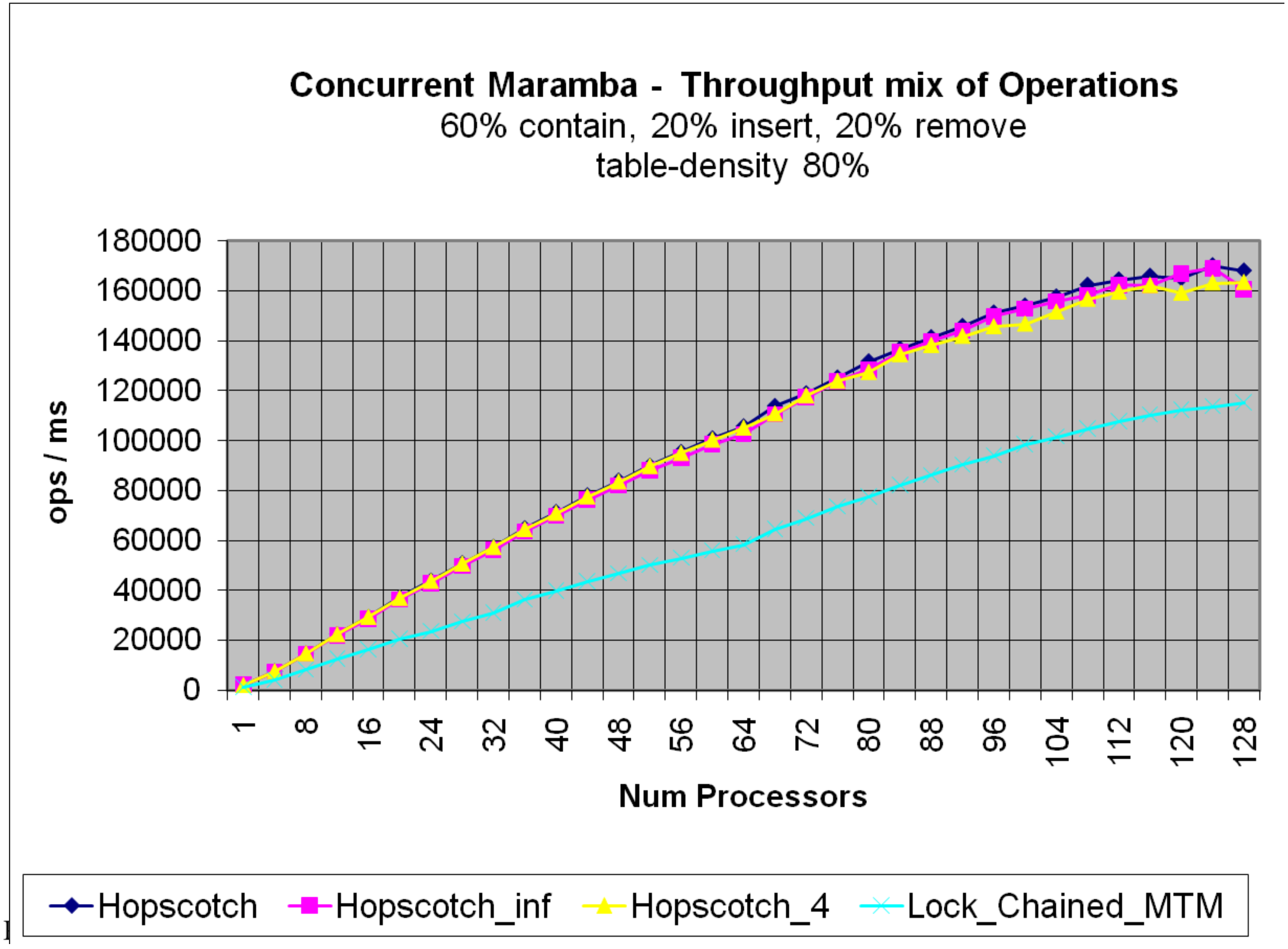
BROWN

Maramba Concurrent

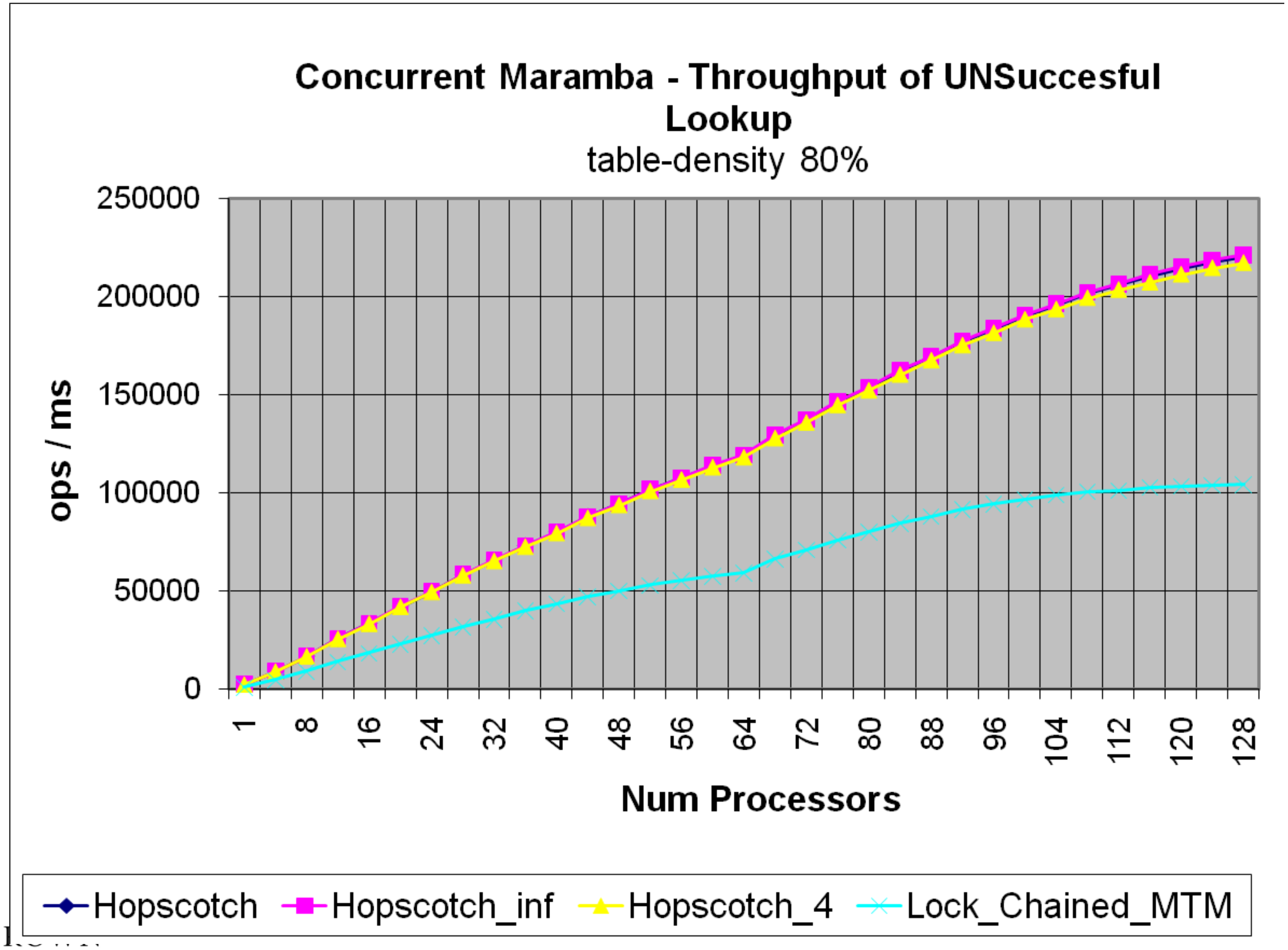


BRO

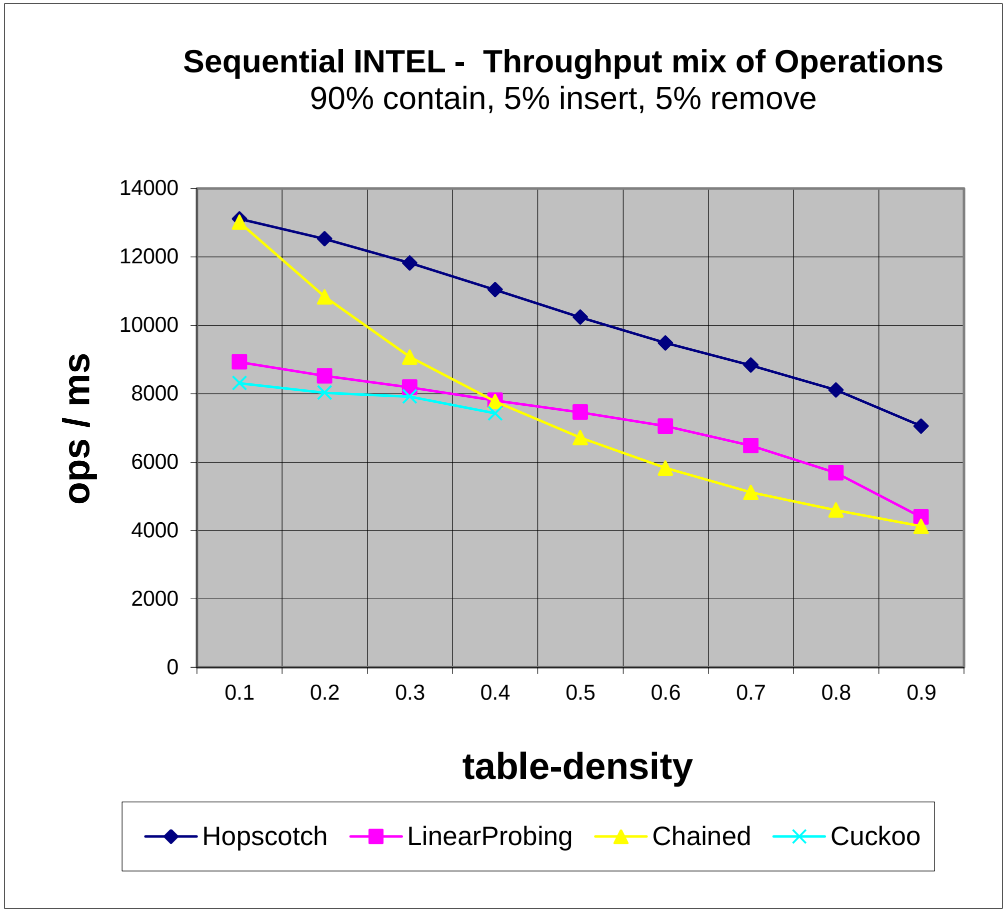
Concurrent Maramba



Maramba Concurrent

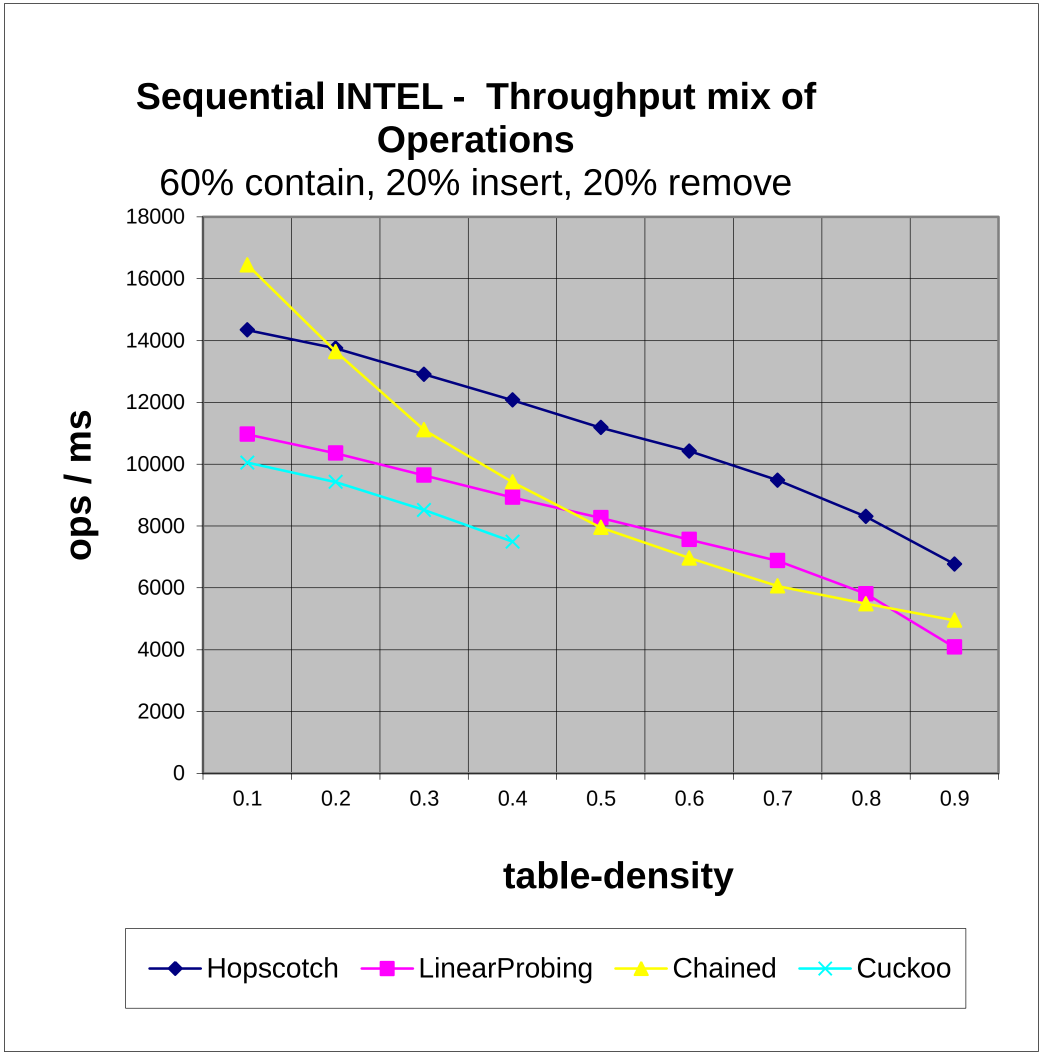


Xeon Sequential



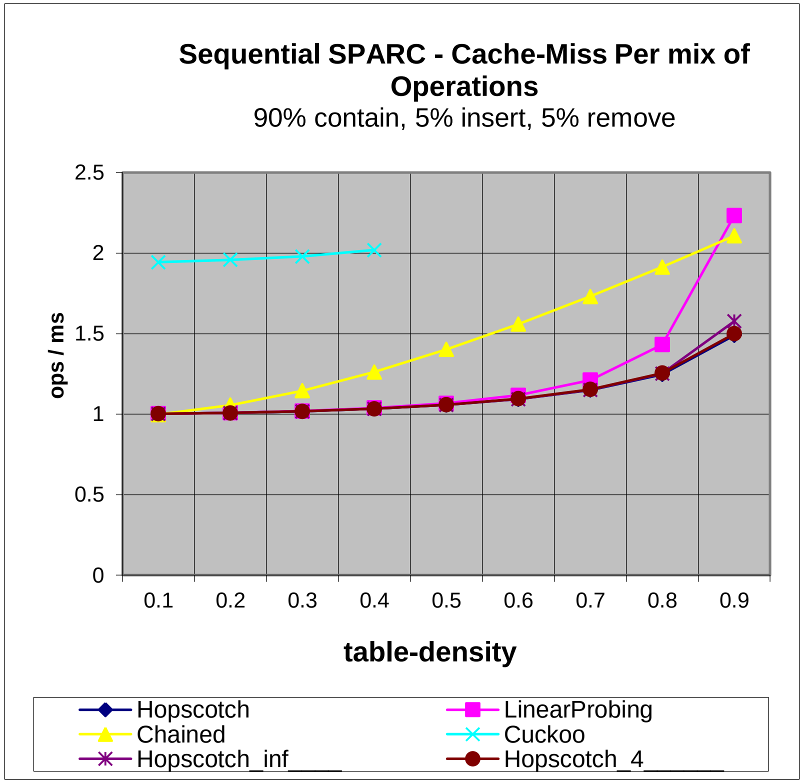
BROWN

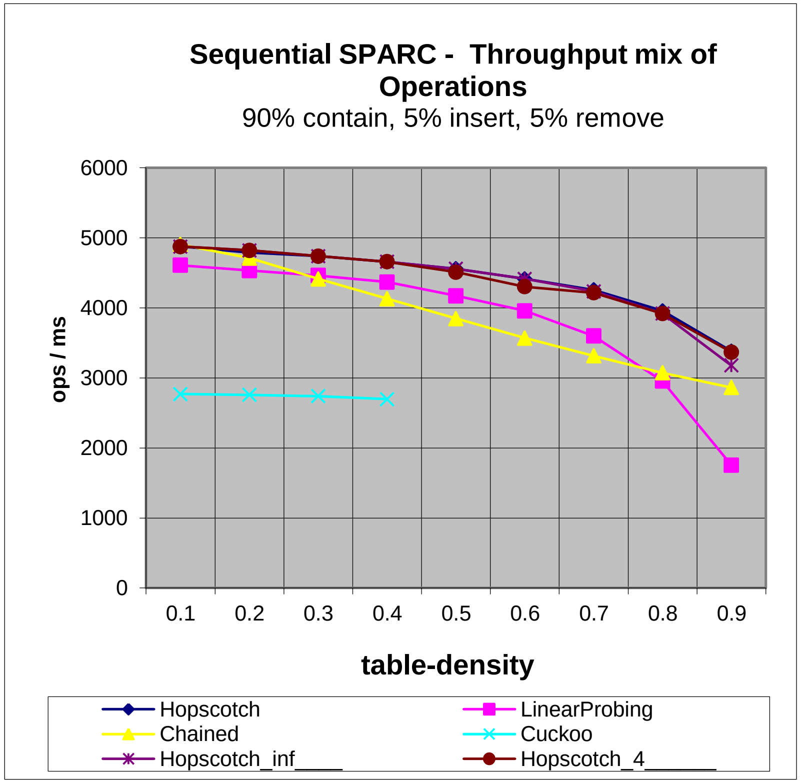
Xeon Sequential



BROWN

SPARC Sequential





BROWN

Conclusions & Future

- New Hopscotch : great cache behavior – great fit with multicores
- Need to better understand performance with collisions and contamination, the biggest drawbacks of linear probing
- Multicores are here, but we have a lot of work understanding how to program them

BROWN