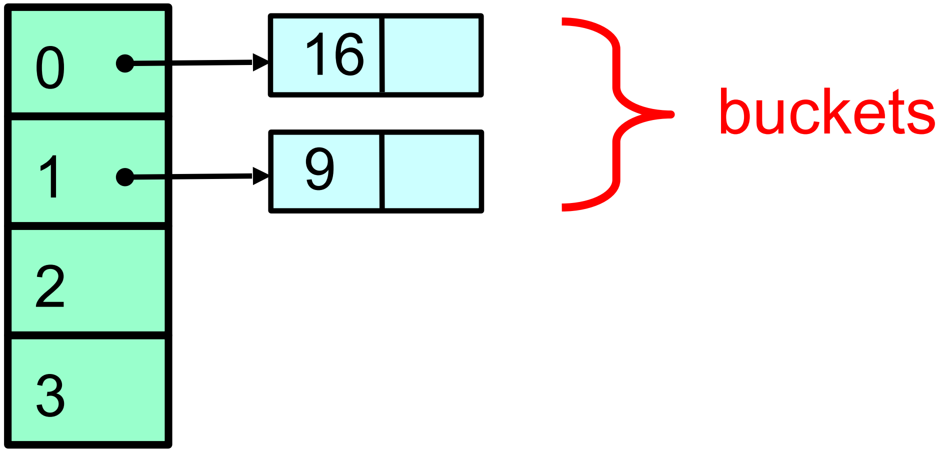
Hash Tables

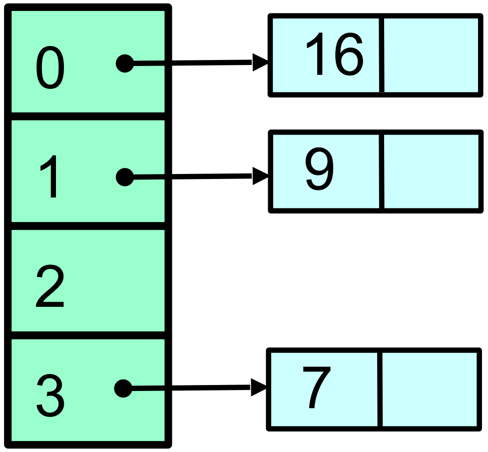
Sequential Closed Hash Map



2 Items

h(k) = k mod 4

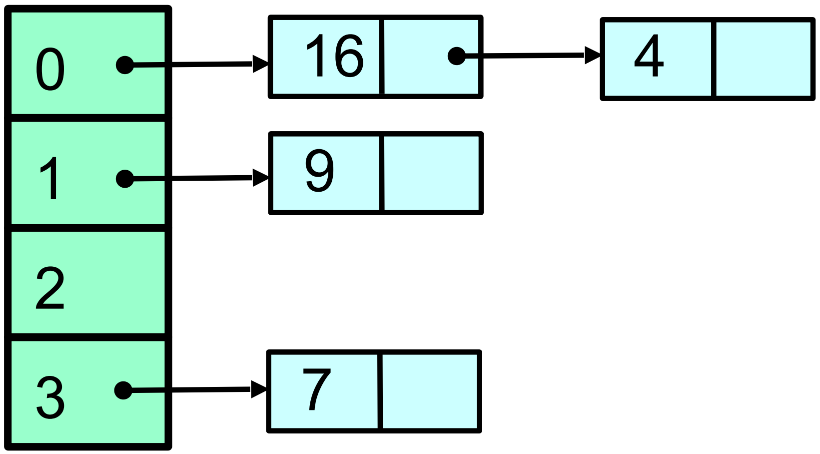
Add an Item



3 Items

h(k) = k mod 4

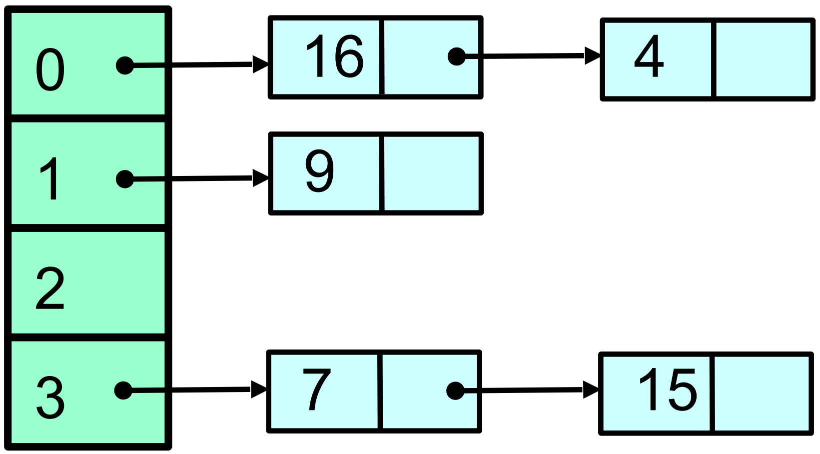
Add Another: Collision



4 Items

h(k) = k mod 4

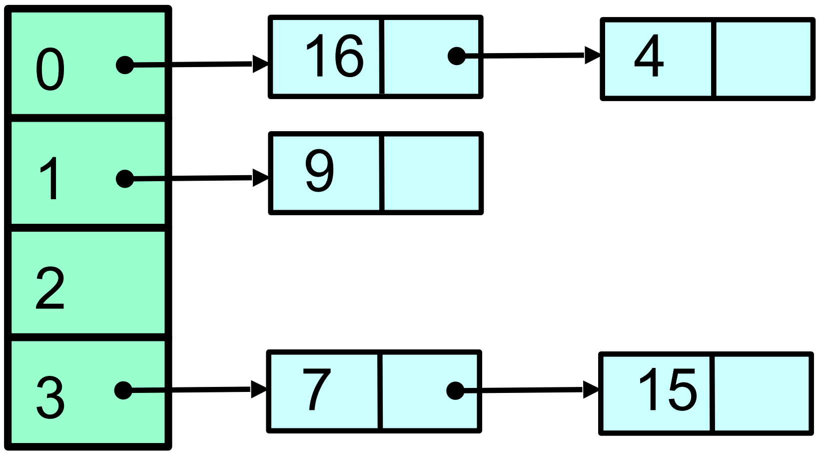
More Collisions



5 Items

h(k) = k mod 4

More Collisions

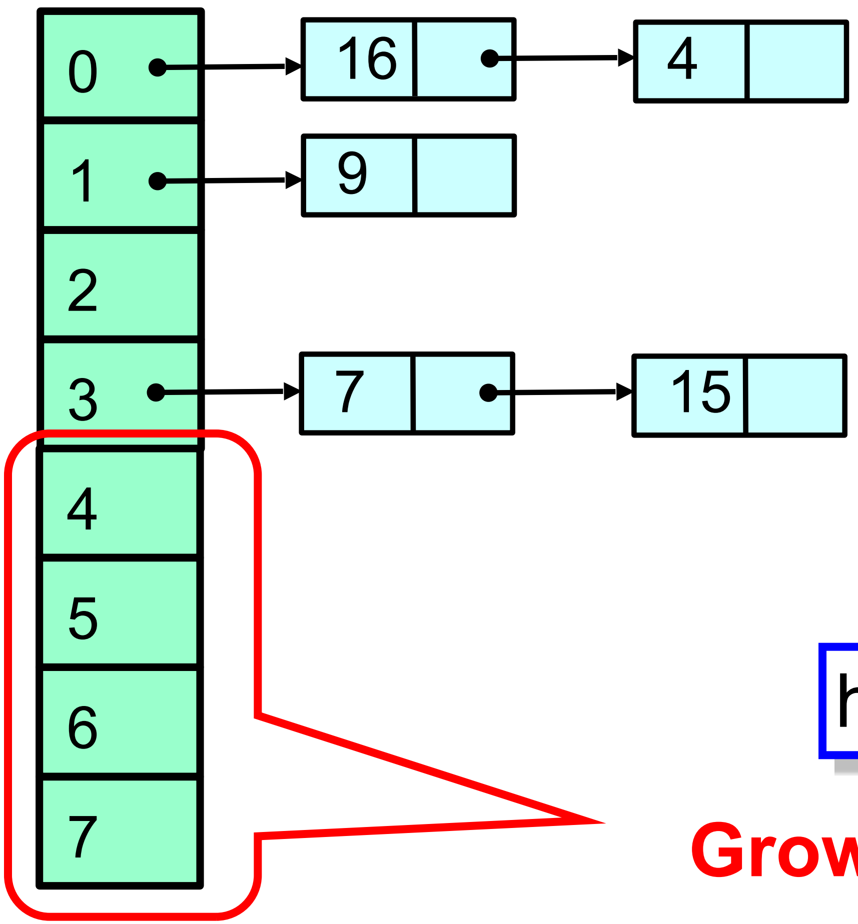


5 Items

$h(k) = k \bmod 4$

Problem:
buckets getting too long

Resizing

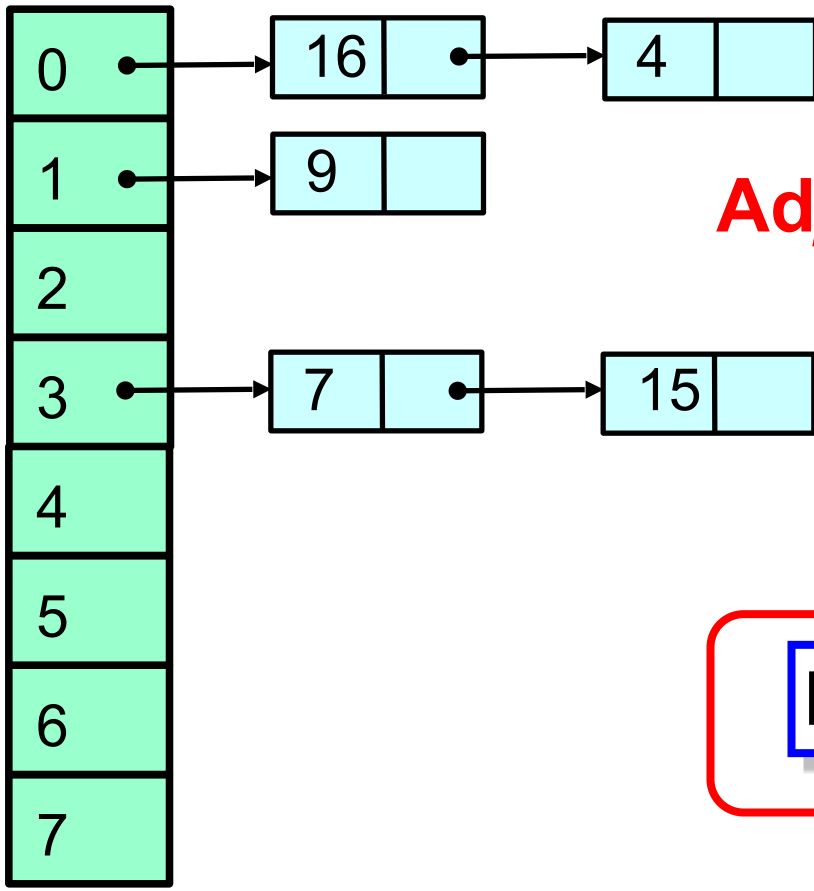


5 Items

$h(k) = k \bmod 8$

Grow the array

Resizing

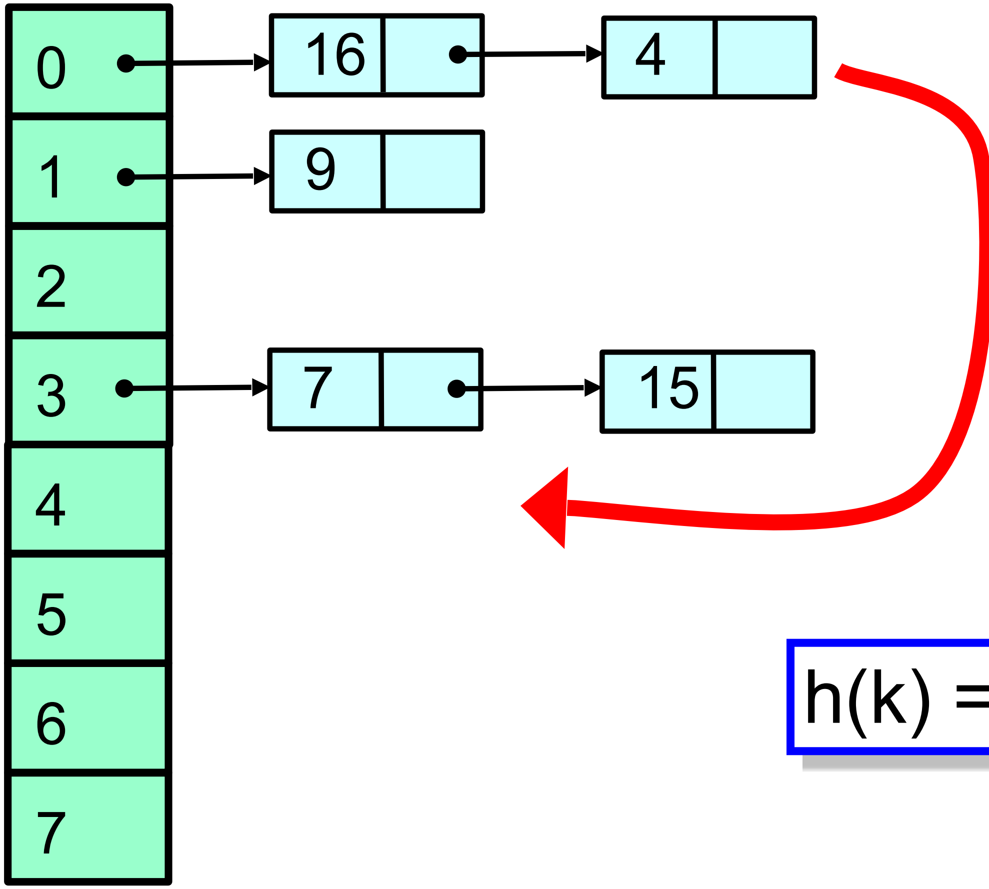


Adjust hash function

5 Items

h(k) = k mod 8

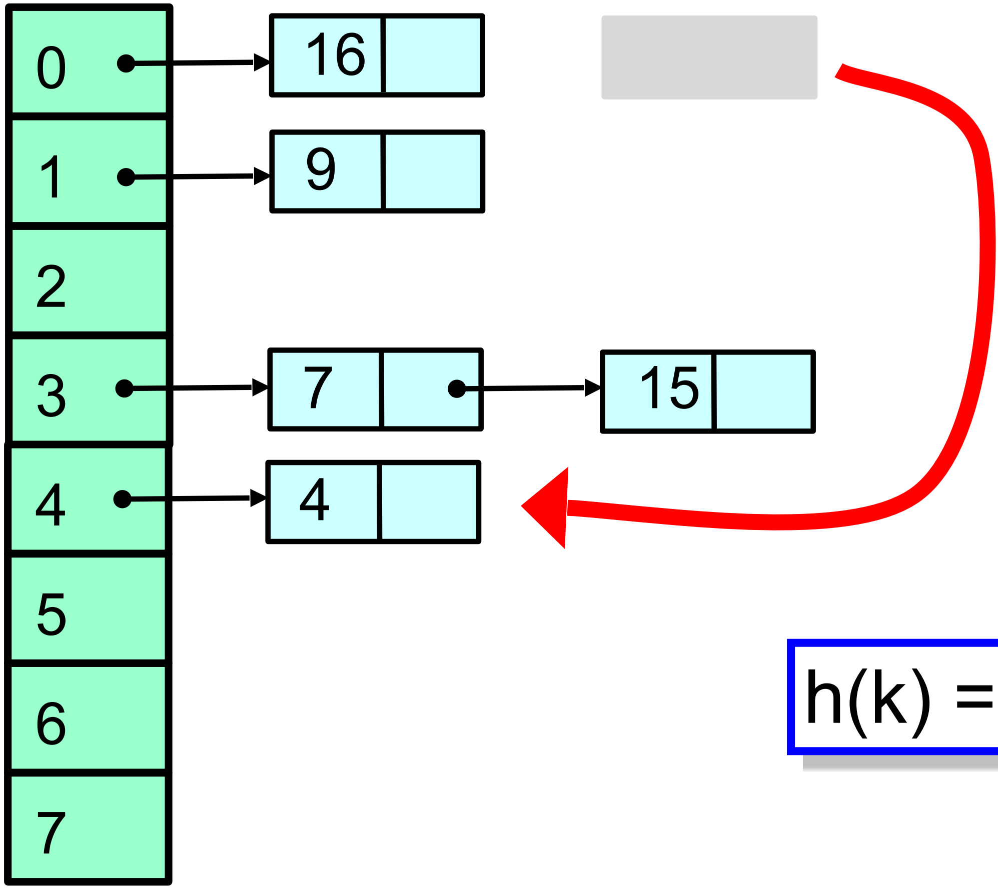
Resizing



h(4) = 0 mod 8

$h(k) = k \bmod 8$

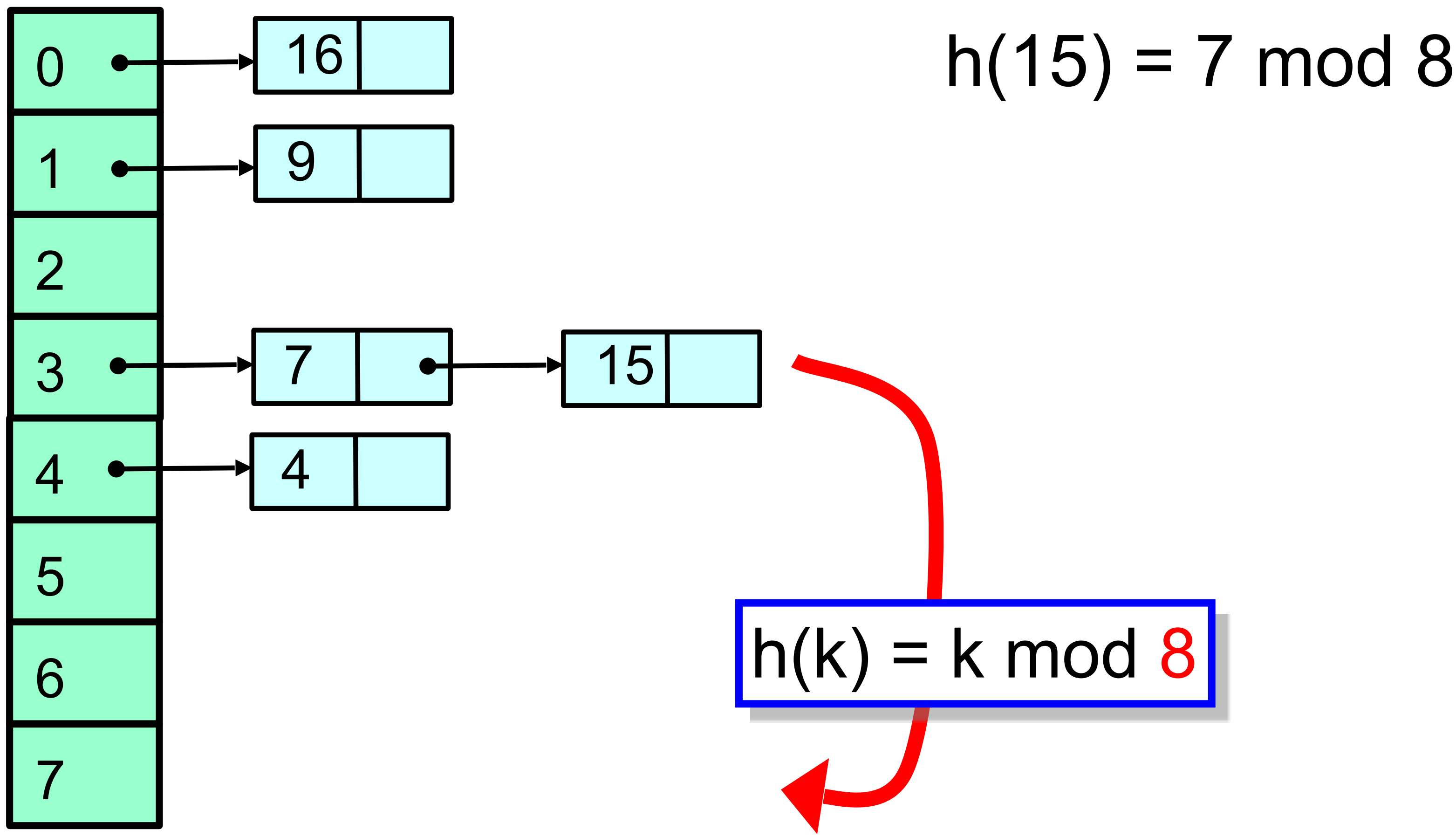
Resizing



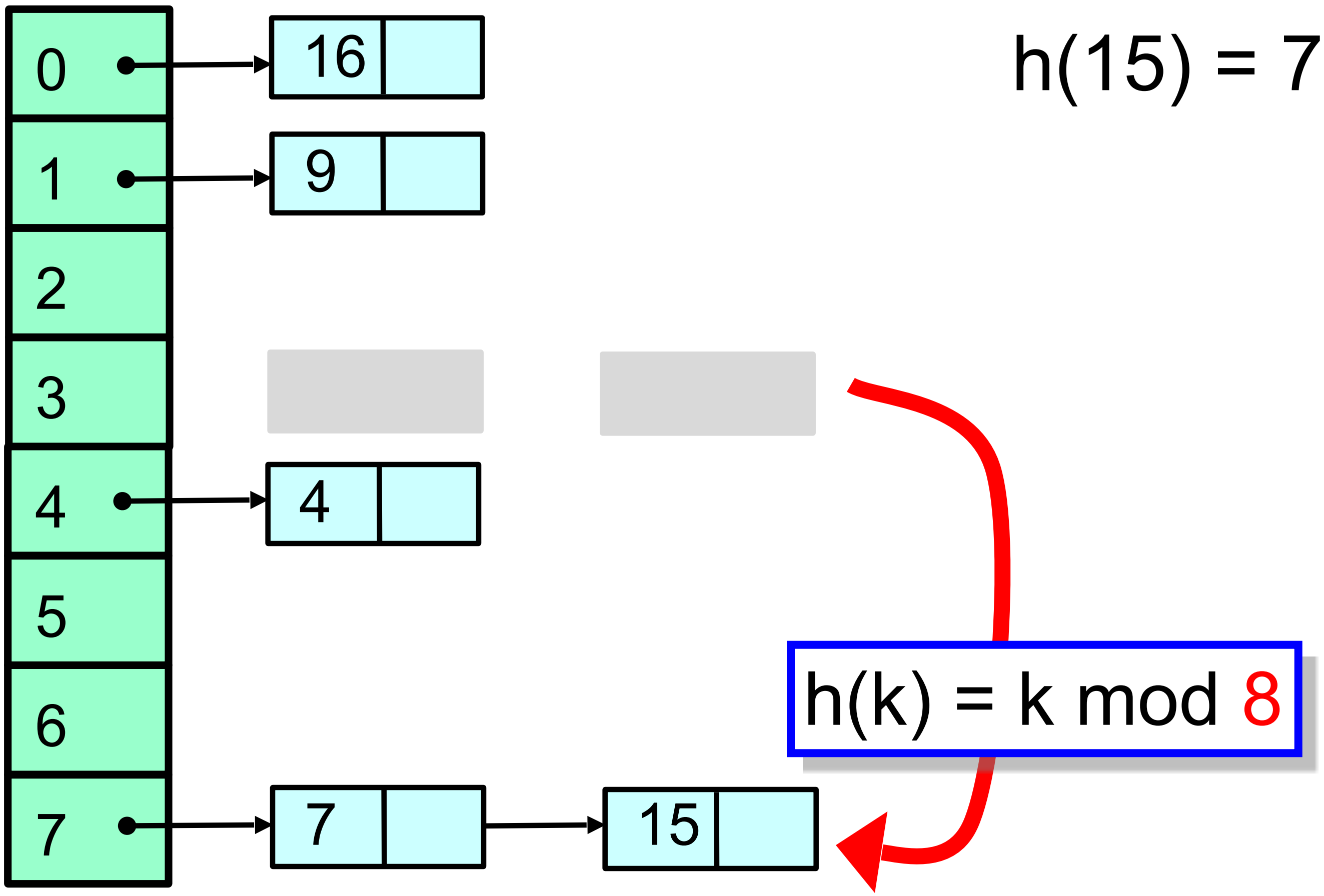
h(4) = 4 mod 8

$\boxed{h(k) = k \bmod 8}$

Resizing



Resizing



h(15) = 7 mod 8

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

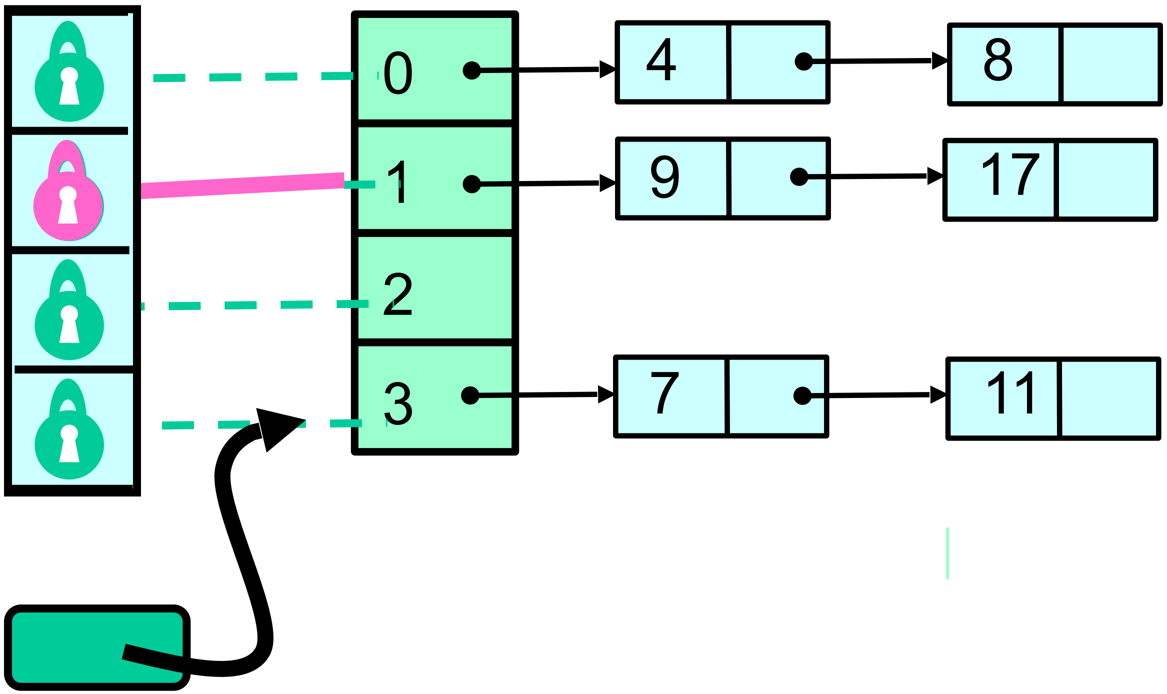
When to Resize?

- Many reasonable policies. Here's one.
- Pick a threshold on num of items in a bucket
- Global threshold
  - When ≥ ¼ buckets exceed this value
- Bucket threshold
  - When any bucket exceeds this value

Coarse-Grained Locking

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

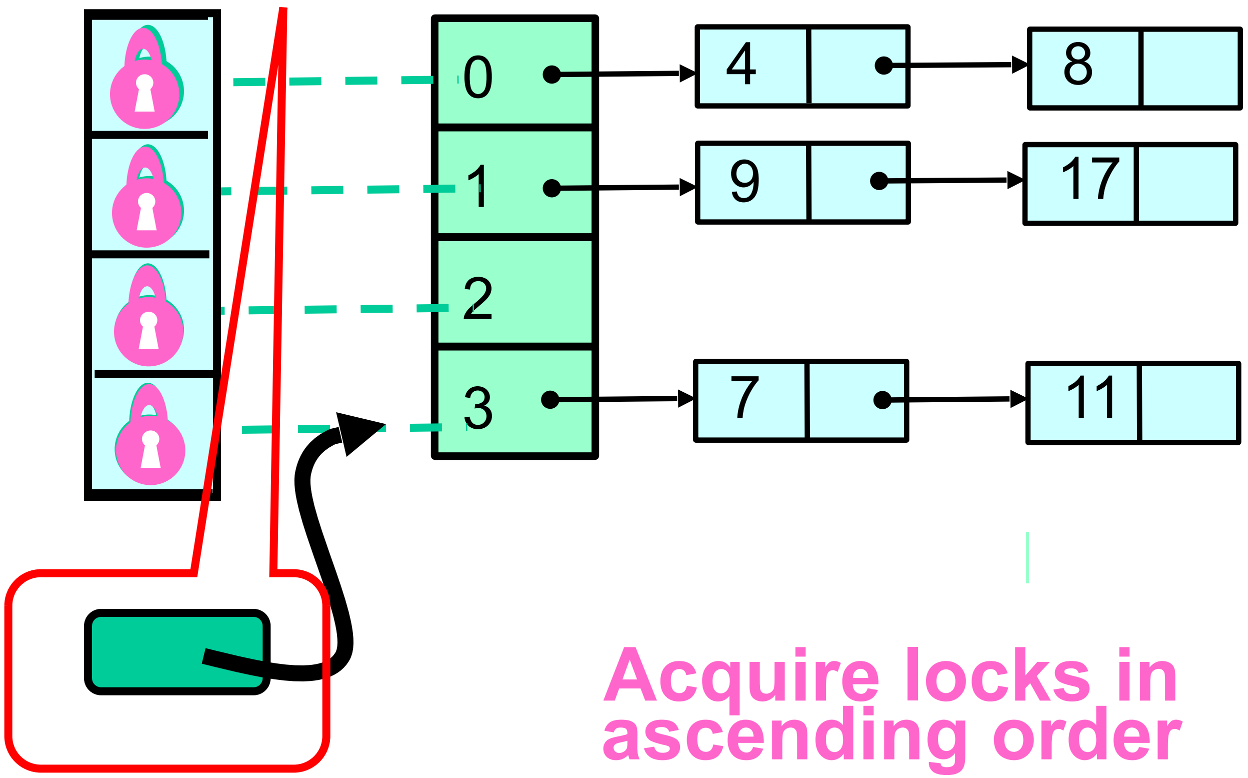
Fine-grained Locking



Each lock associated with one bucket

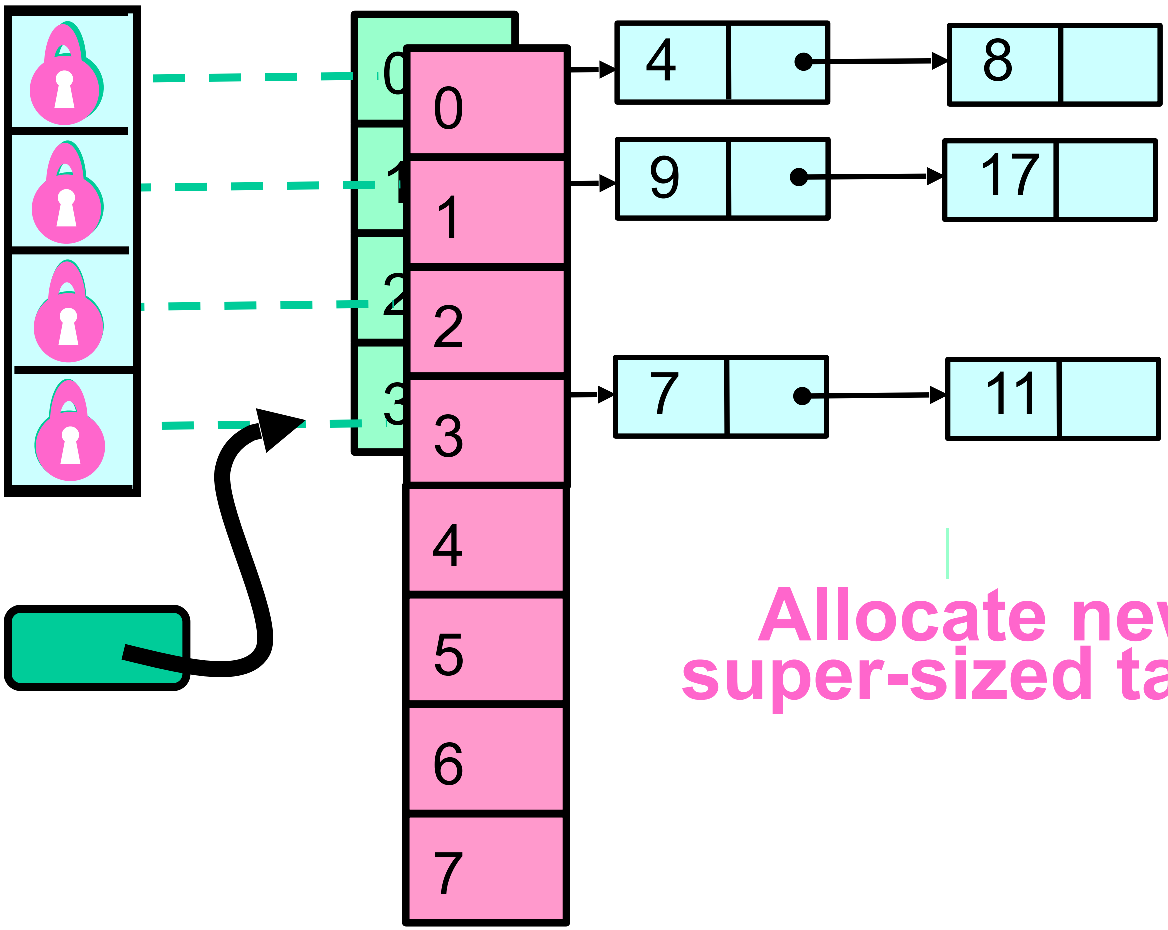
Resize This

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



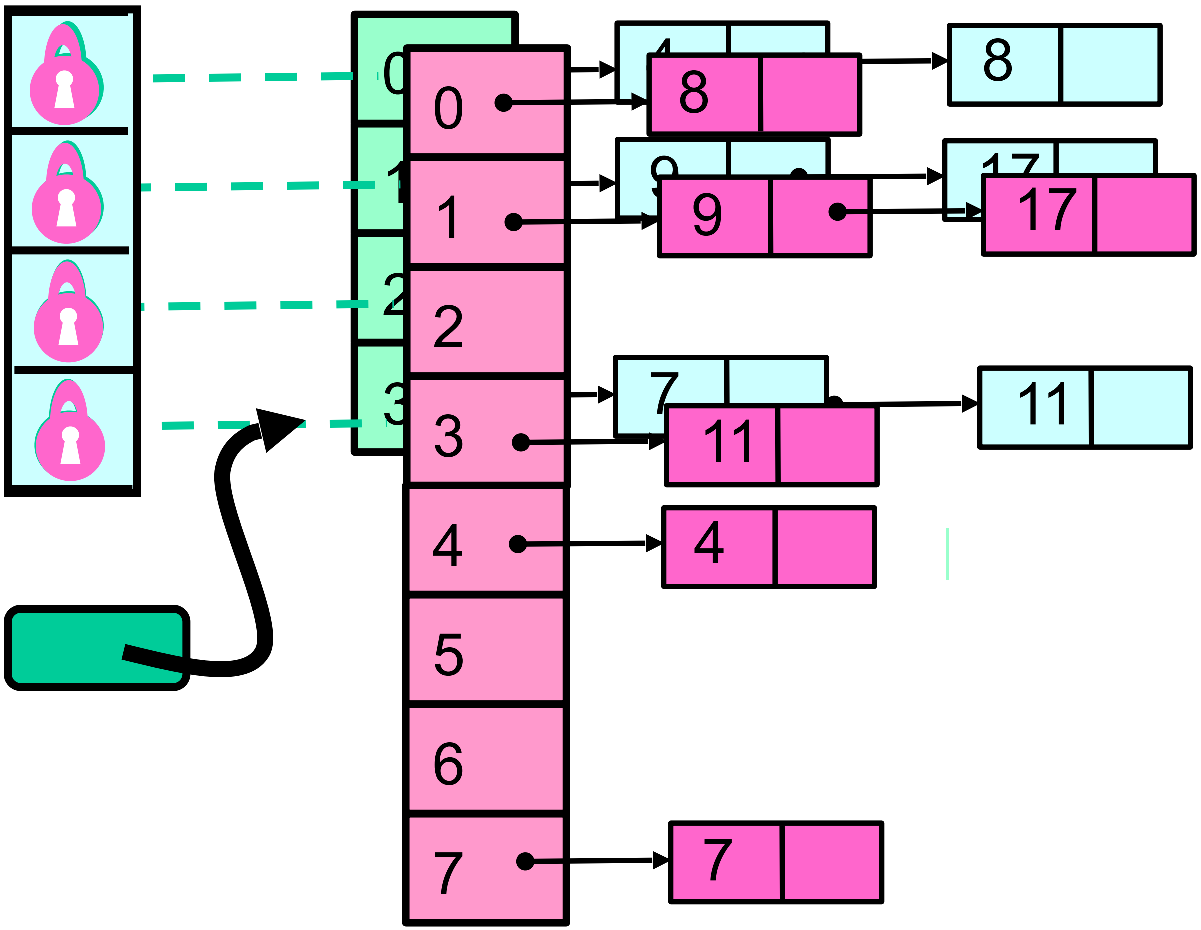
Acquire locks in ascending order

Resize This



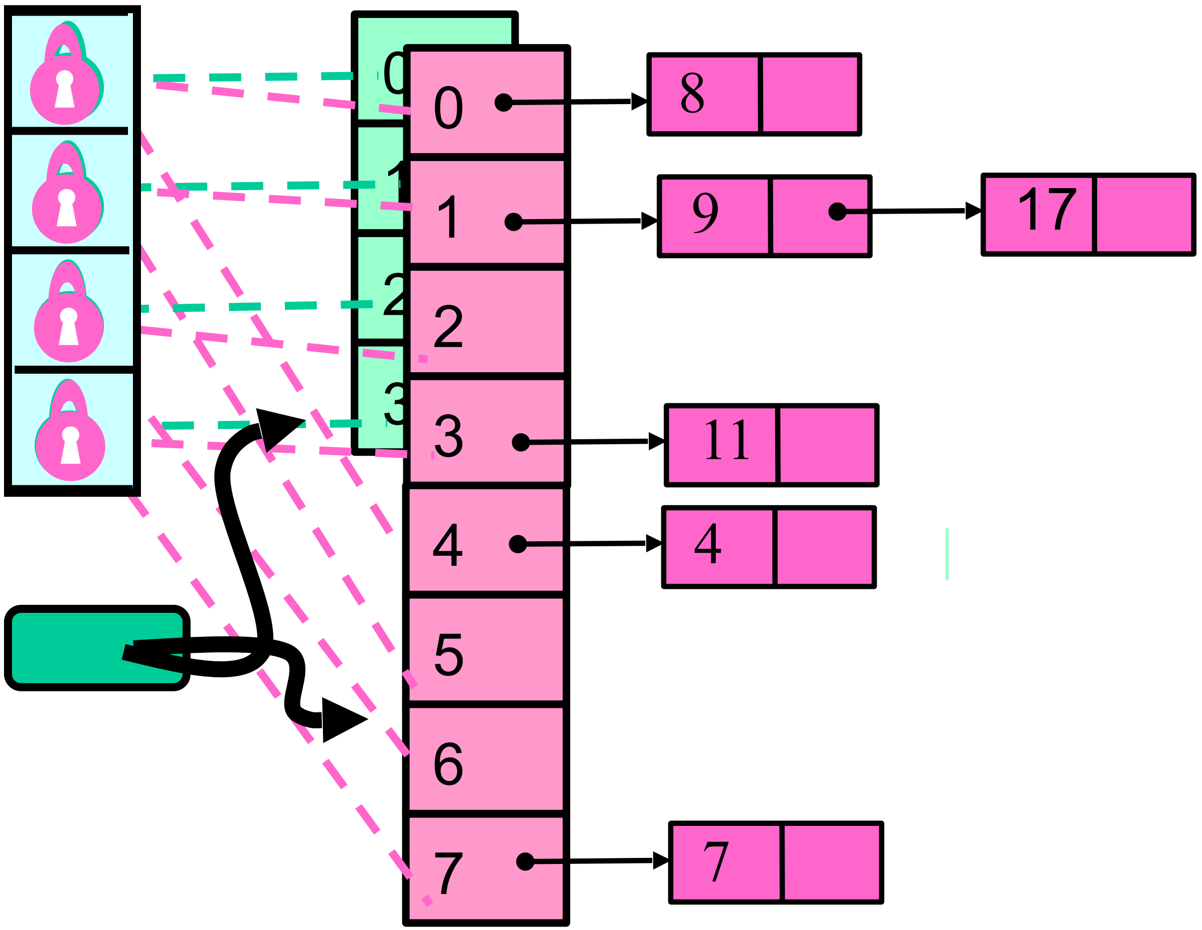
Allocate new super-sized table

Resize This



Resize This

Striped Locks: each lock now associated with two buckets



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?

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

- Read lock:
  - Locks out writers
  - Allows concurrent readers
- Write lock
  - Locks out writers
  - Locks out readers

Read/Write Lock

- Safety
  - If **readers > 0** then **writer == false**
  - If **writer == true** then **readers == 0**
- Liveness?
  - Will a continual stream of readers …
  - Lock out writers?

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

Optimistic Synchronization

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

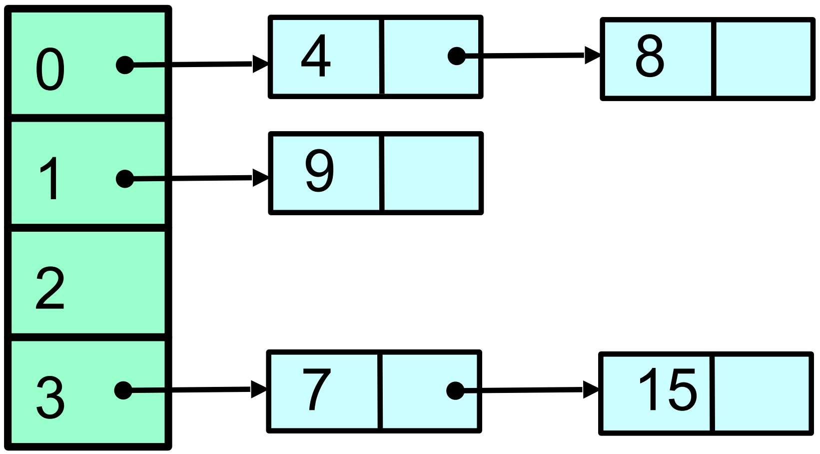
Optimistic Synchronization

- 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

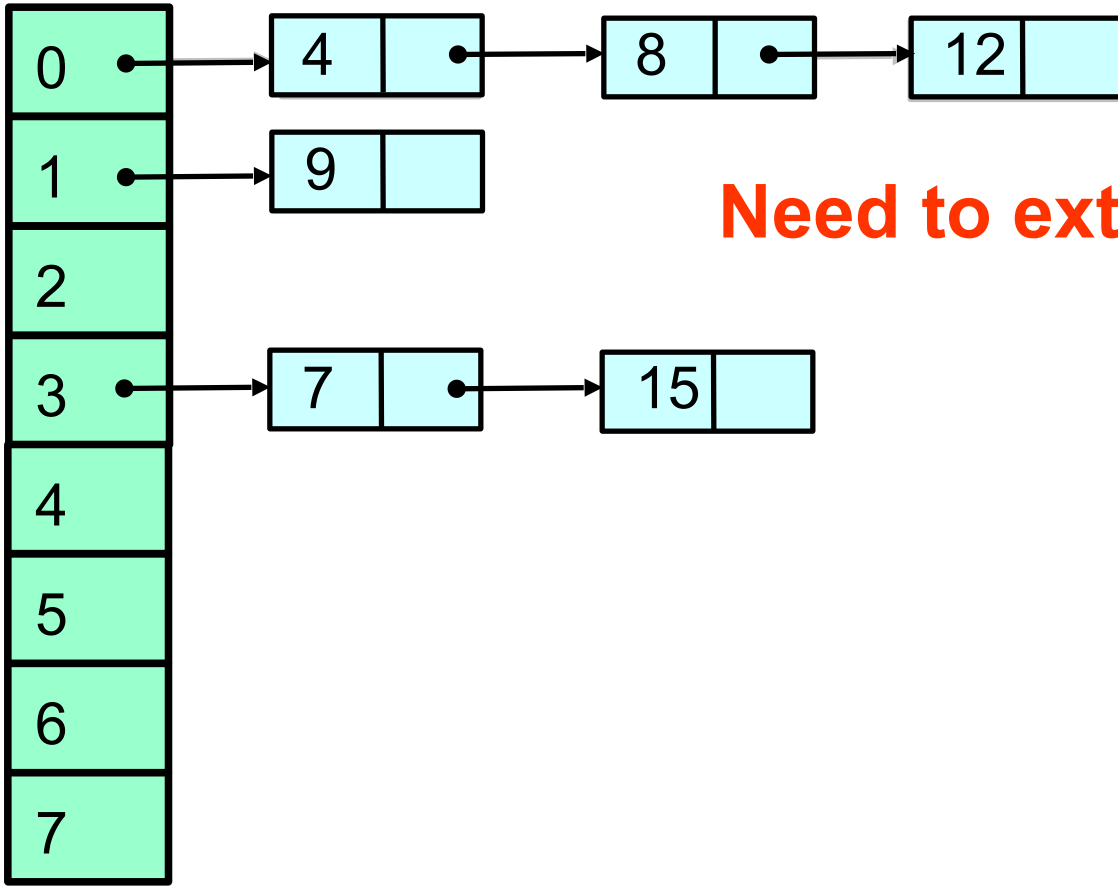
Stop The World Resizing

- Resizing stops all concurrent operations
- What about an incremental resize?
- Must avoid locking the table
- A lock-free table + incremental resizing?

Lock-Free Resizing Problem

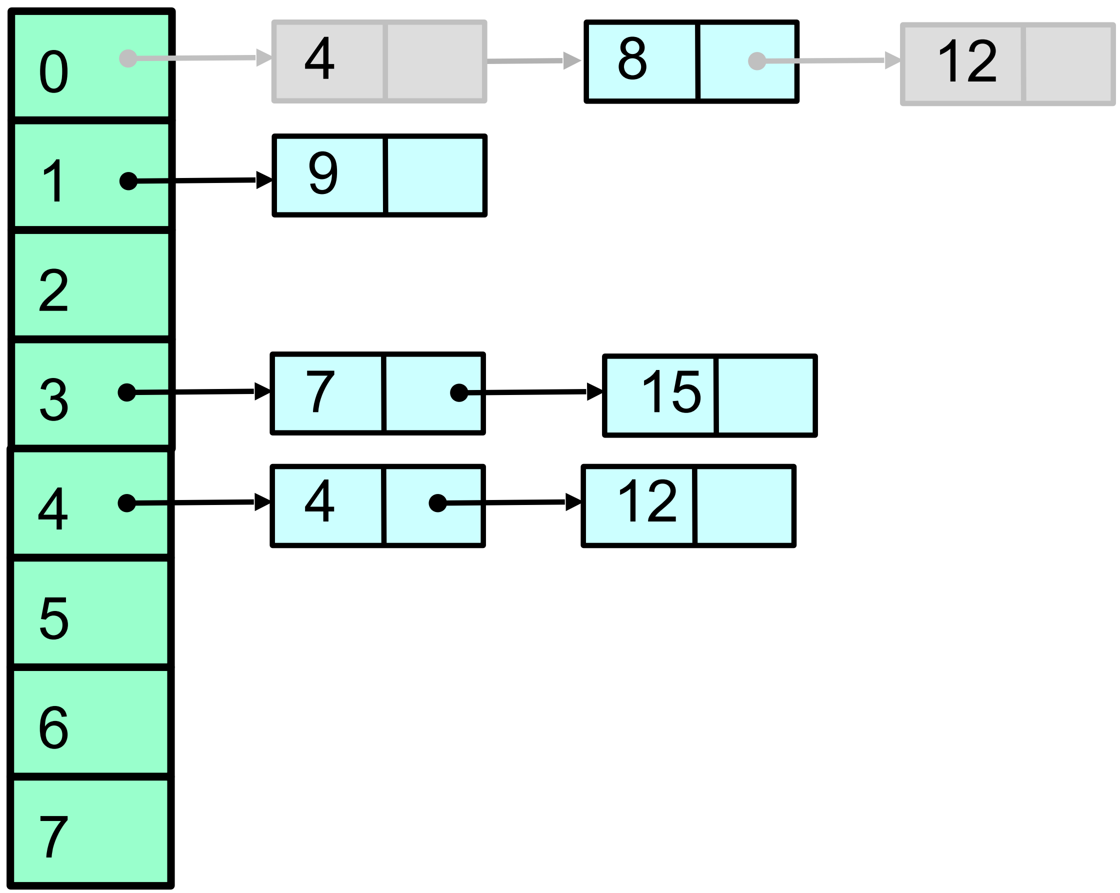


Lock-Free Resizing Problem

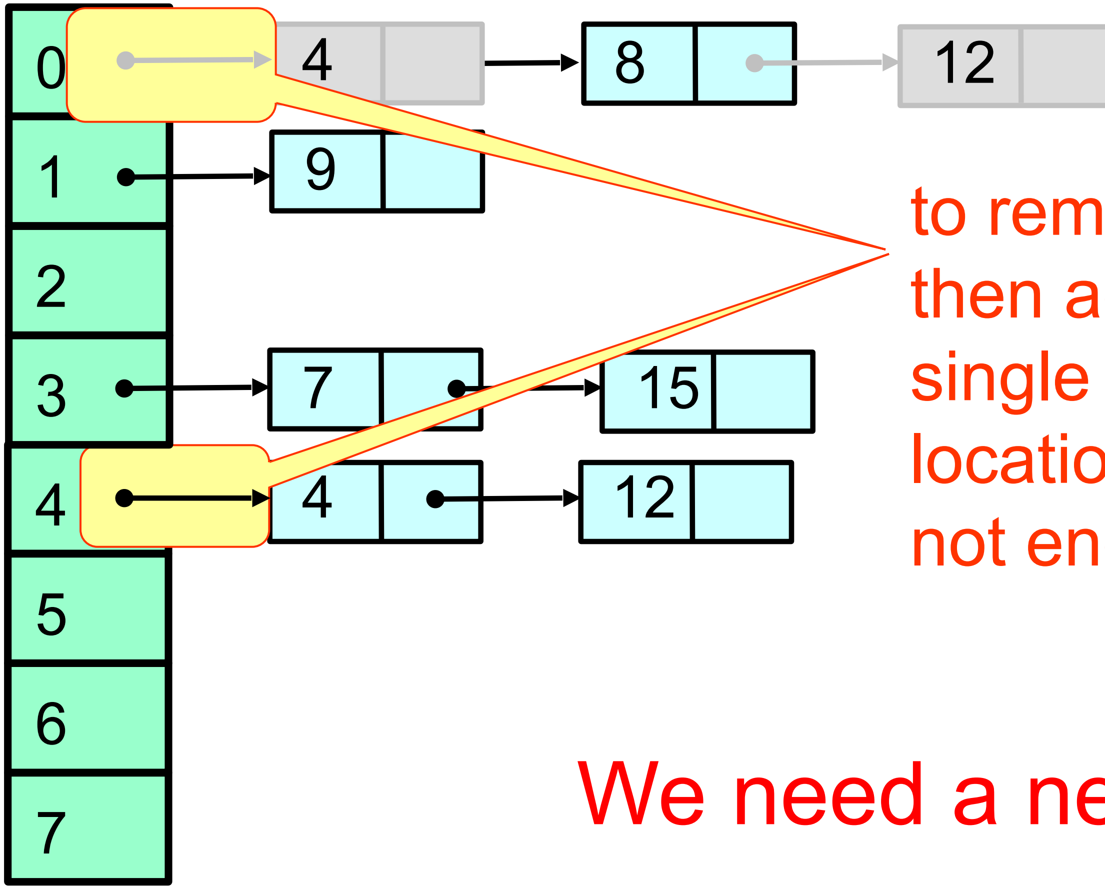


Need to extend table

Lock-Free Resizing Problem



Lock-Free Resizing Problem

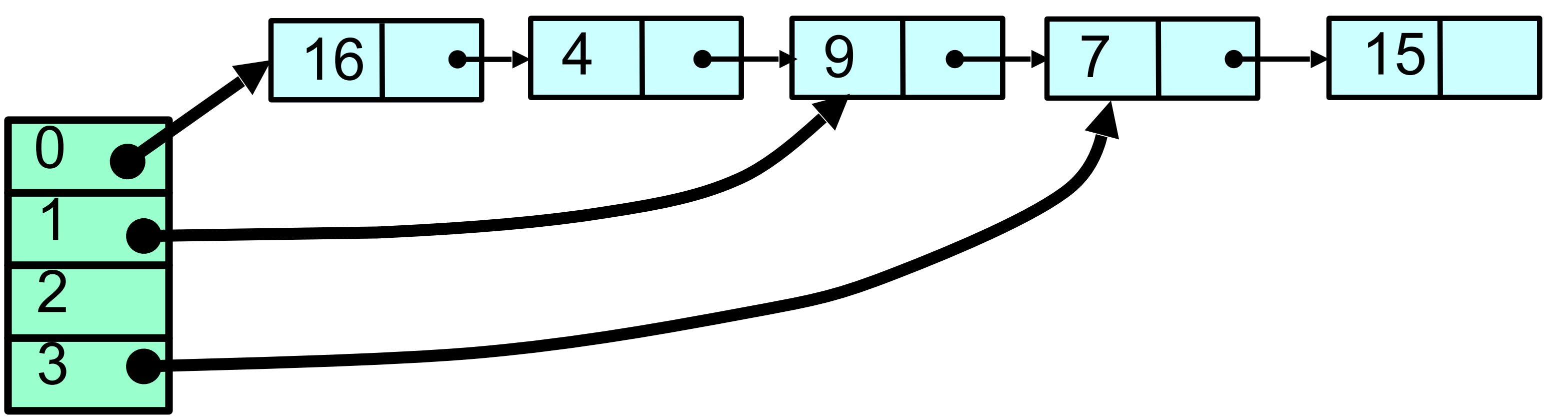


to remove and then add even a single item single location CAS not enough

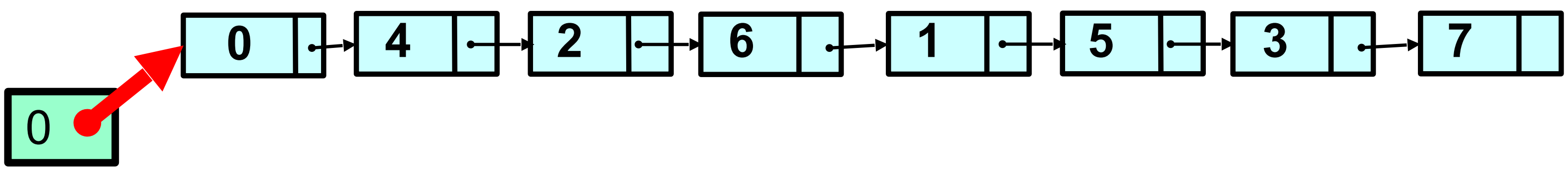
We need a new idea…

Don't move the items

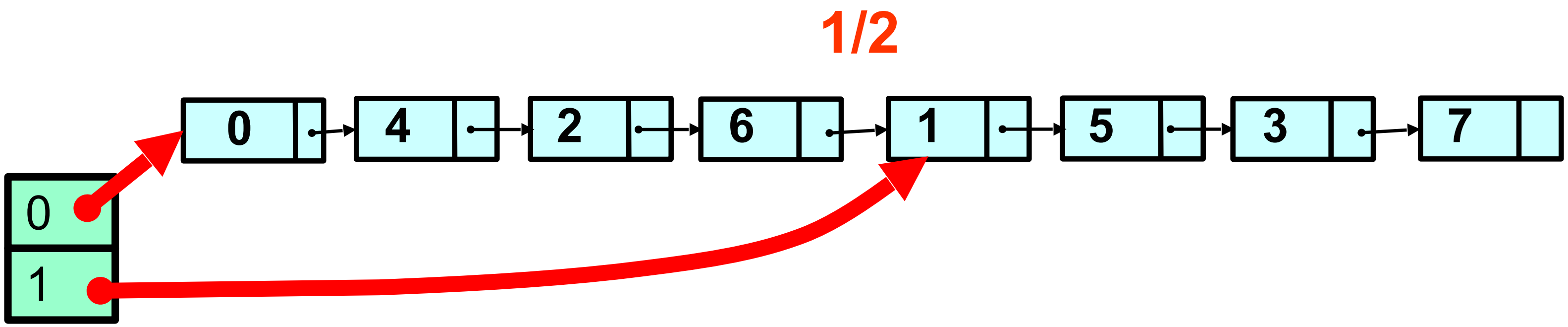
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



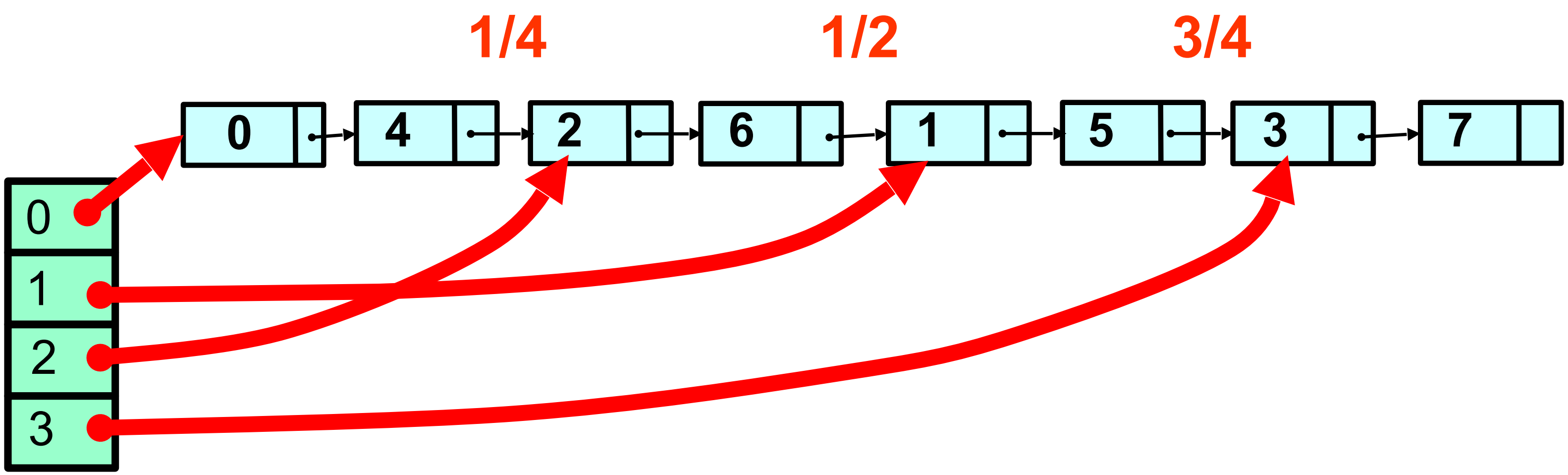
Recursive Split Ordering



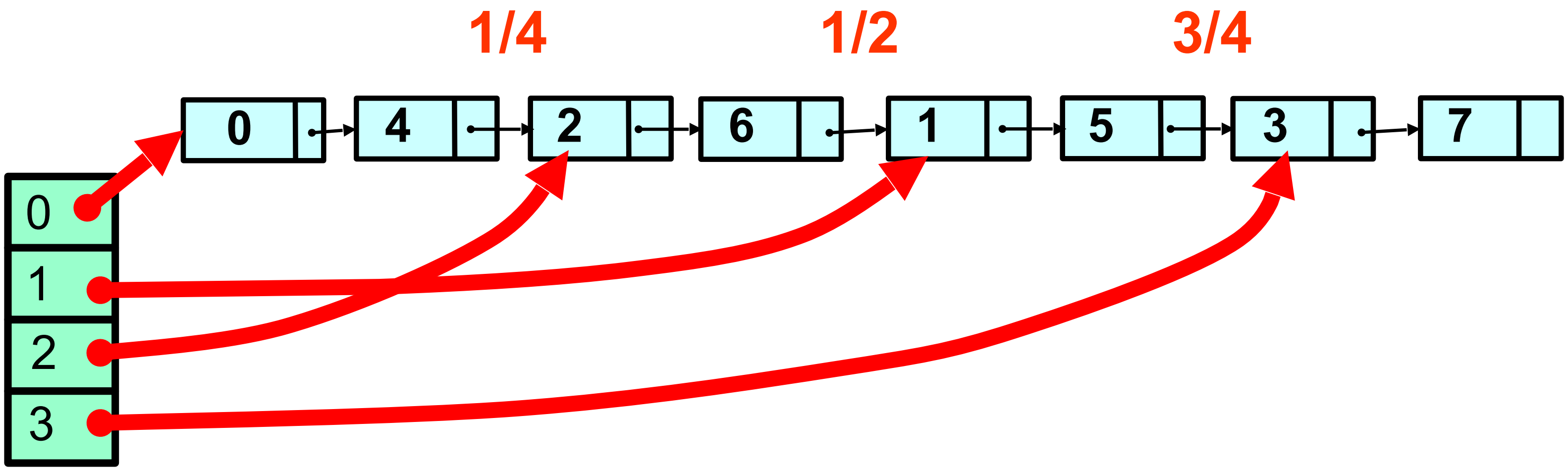
Recursive Split Ordering



Recursive Split Ordering

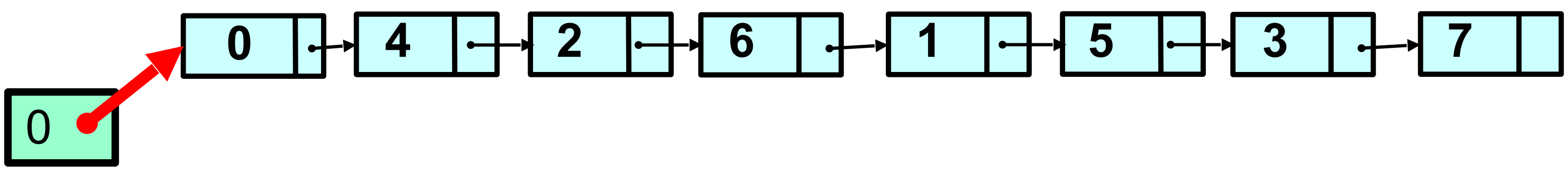


Recursive Split Ordering

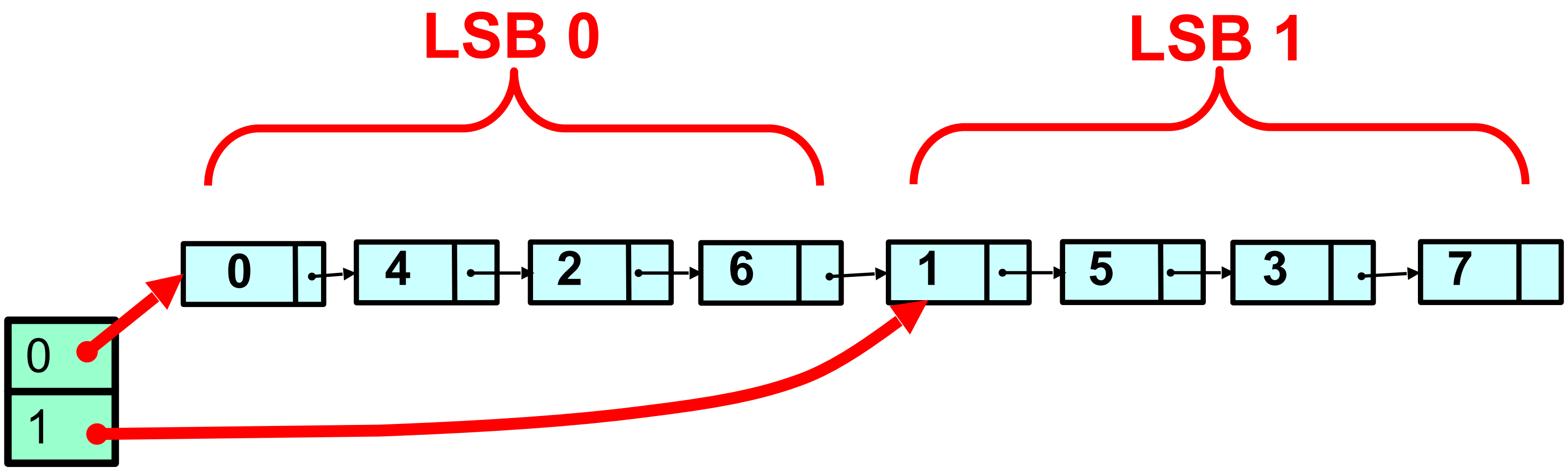


List entries sorted in order that allows recursive splitting. How?

Recursive Split Ordering

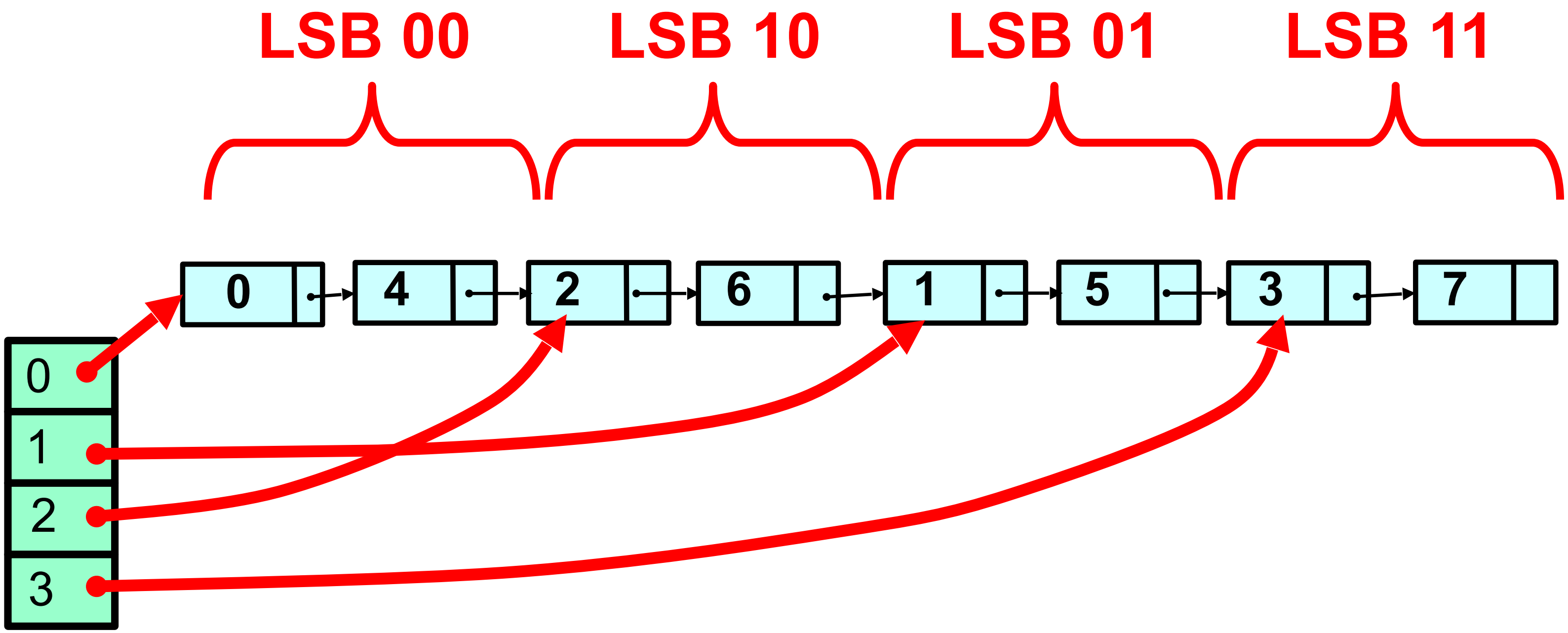


Recursive Split Ordering



LSB = Least significant Bit

Recursive Split Ordering



Split-Order

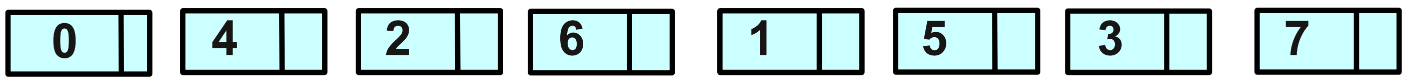
- If the table size is 2^i,
  - Bucket b contains keys k
    - $k = b \pmod{2^i}$
  - bucket index consists of key's i LSBs

When Table Splits

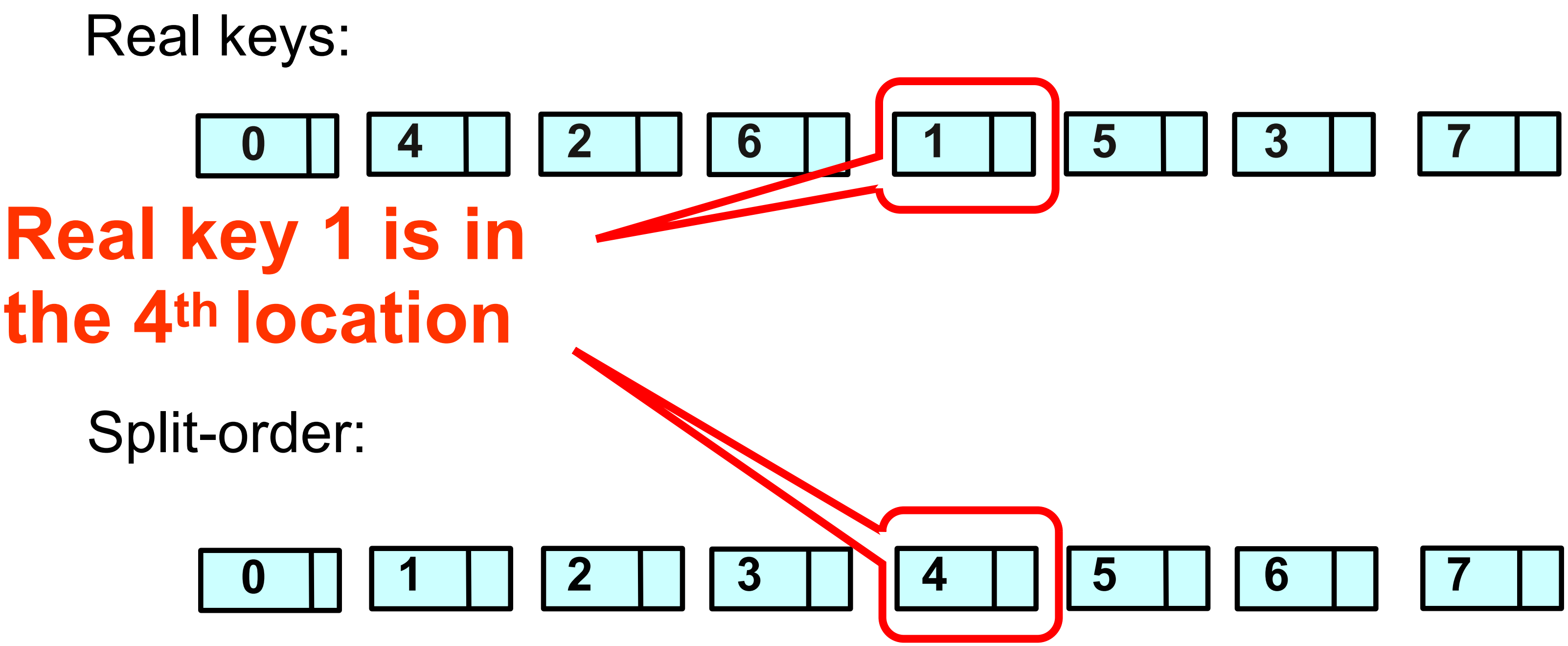
- Some keys stay
  - $b = k \bmod (2^{i+1})$
- Some move
  - $b+2^i = k \bmod (2^{i+1})$
- Determined by $(i+1)^{st}$ bit
  - Counting backwards
- Key must be accessible from both
  - Keys that will move must come later

A Bit of Magic

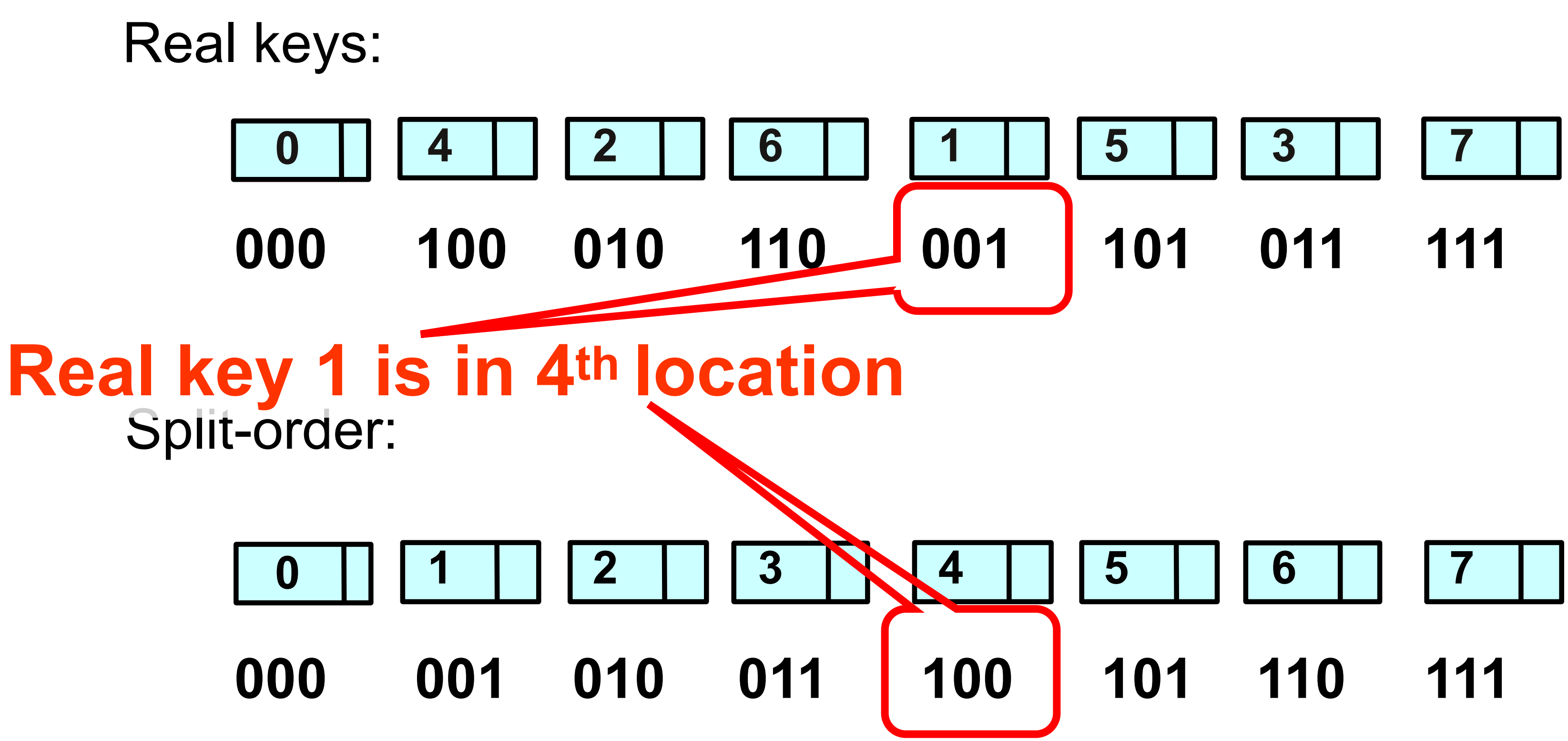
Real keys:



A Bit of Magic



A Bit of Magic



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:

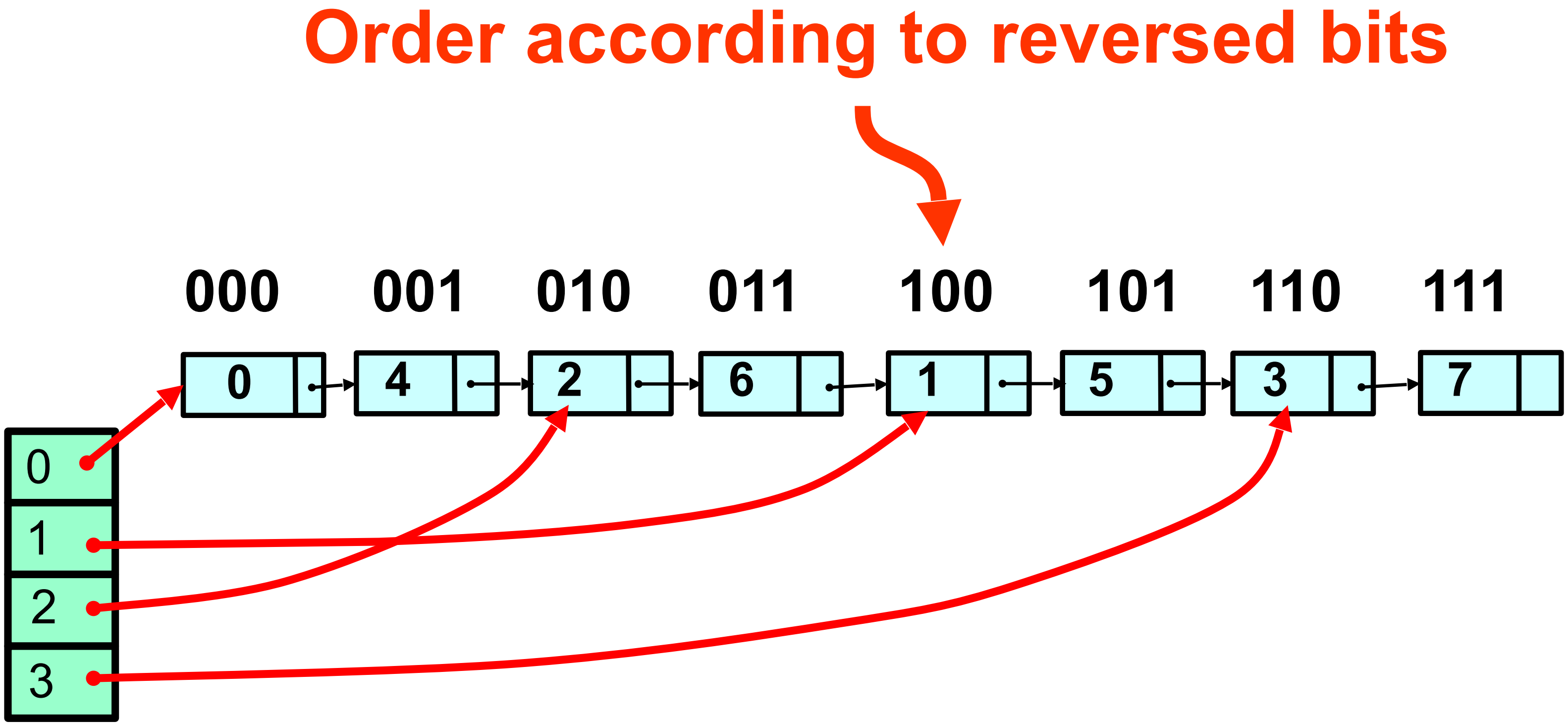
000 100 010 110 001 101 011 111

Split-order:

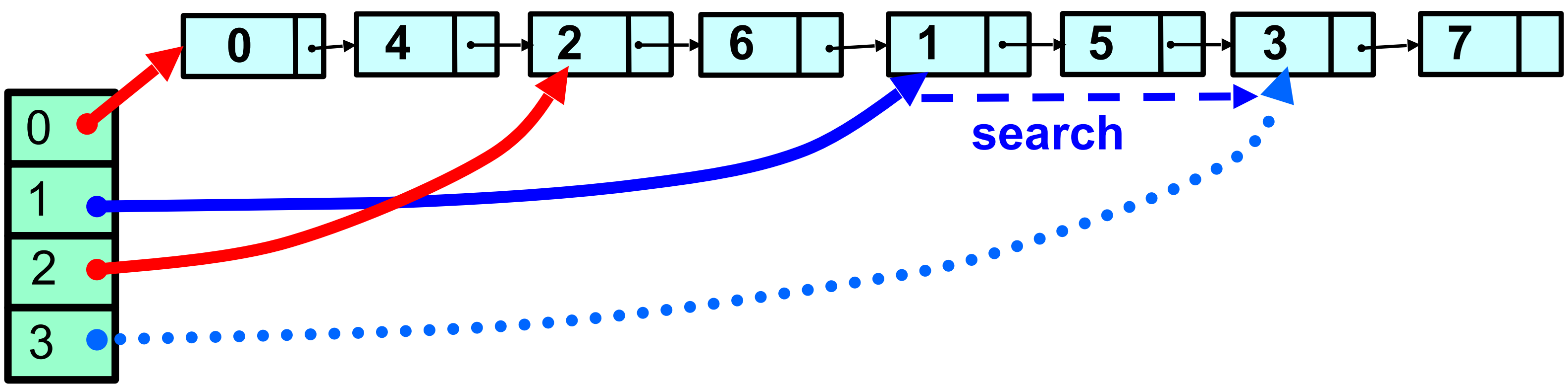
000 001 010 011 100 101 110 111

Just reverse the order of the key bits

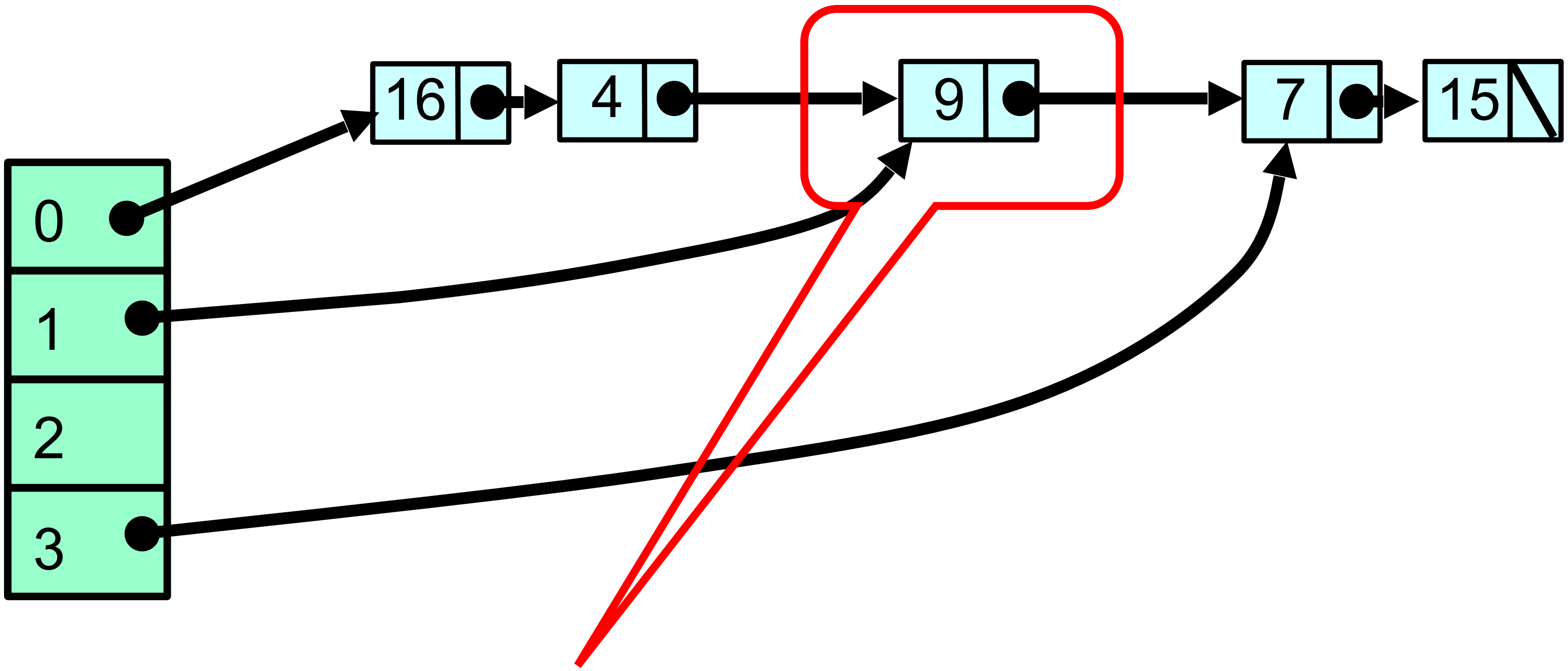
Split Ordered Hashing



Parent Always Provides a Short Cut

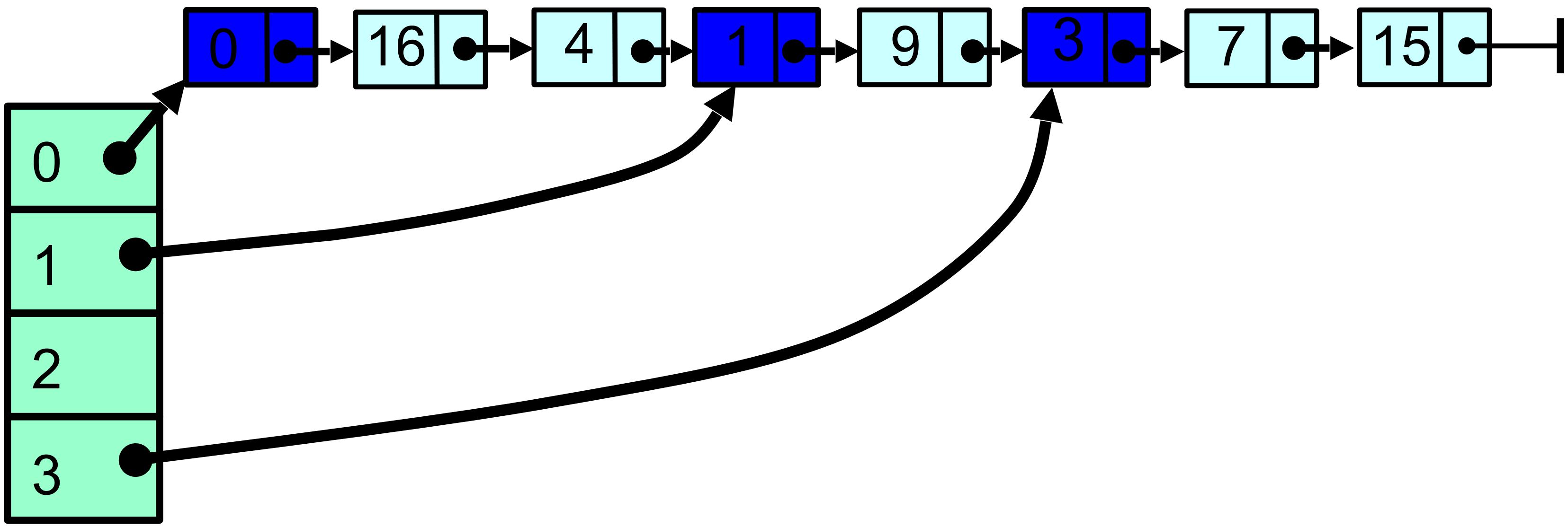


Sentinel Nodes



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

Sentinel Nodes



Solution: use a Sentinel node for each bucket

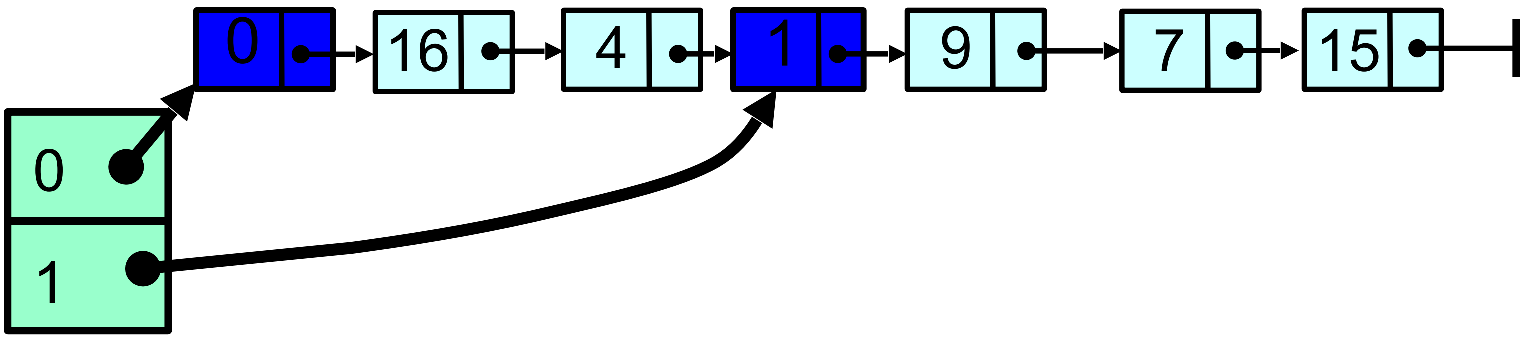
Sentinel vs Regular Keys

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

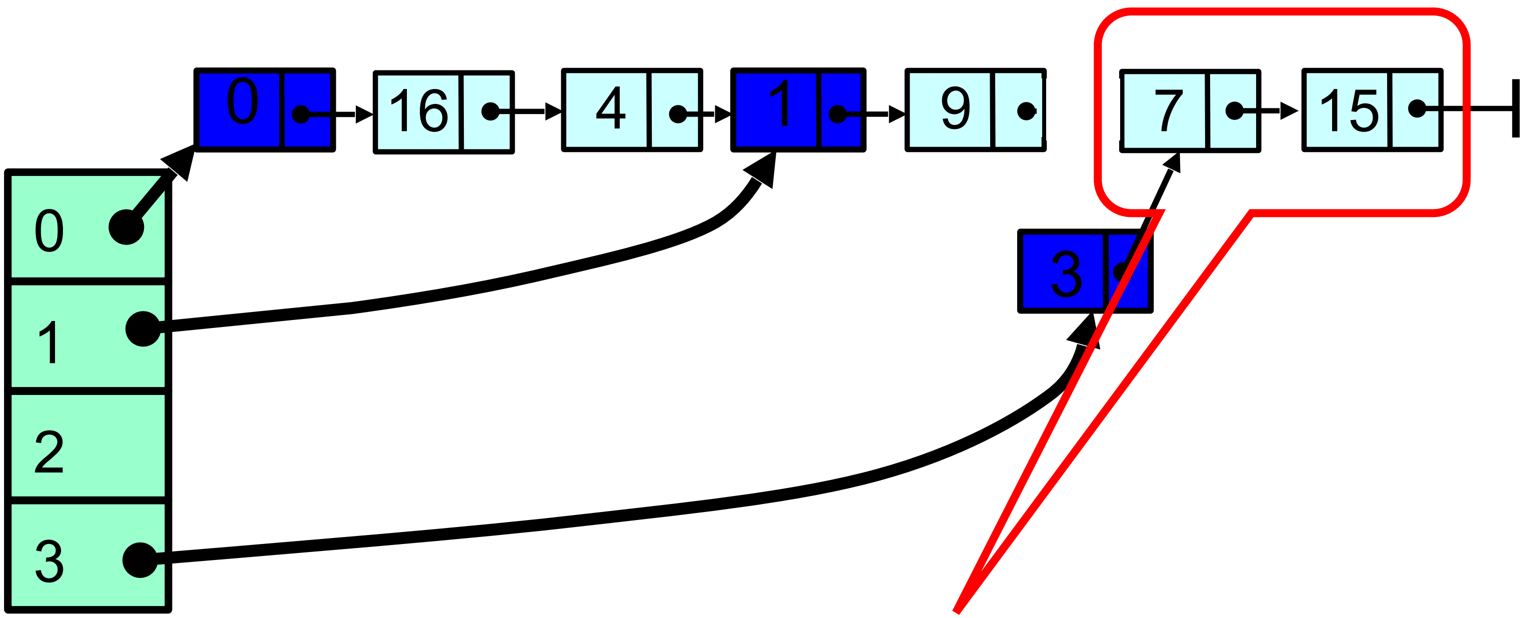
Splitting a Bucket

- We can now split a bucket
- In a lock-free manner
- Using two CAS() calls ...
  - One to add the sentinel to the list
  - The other to point from the bucket to the sentinel

Initialization of Buckets

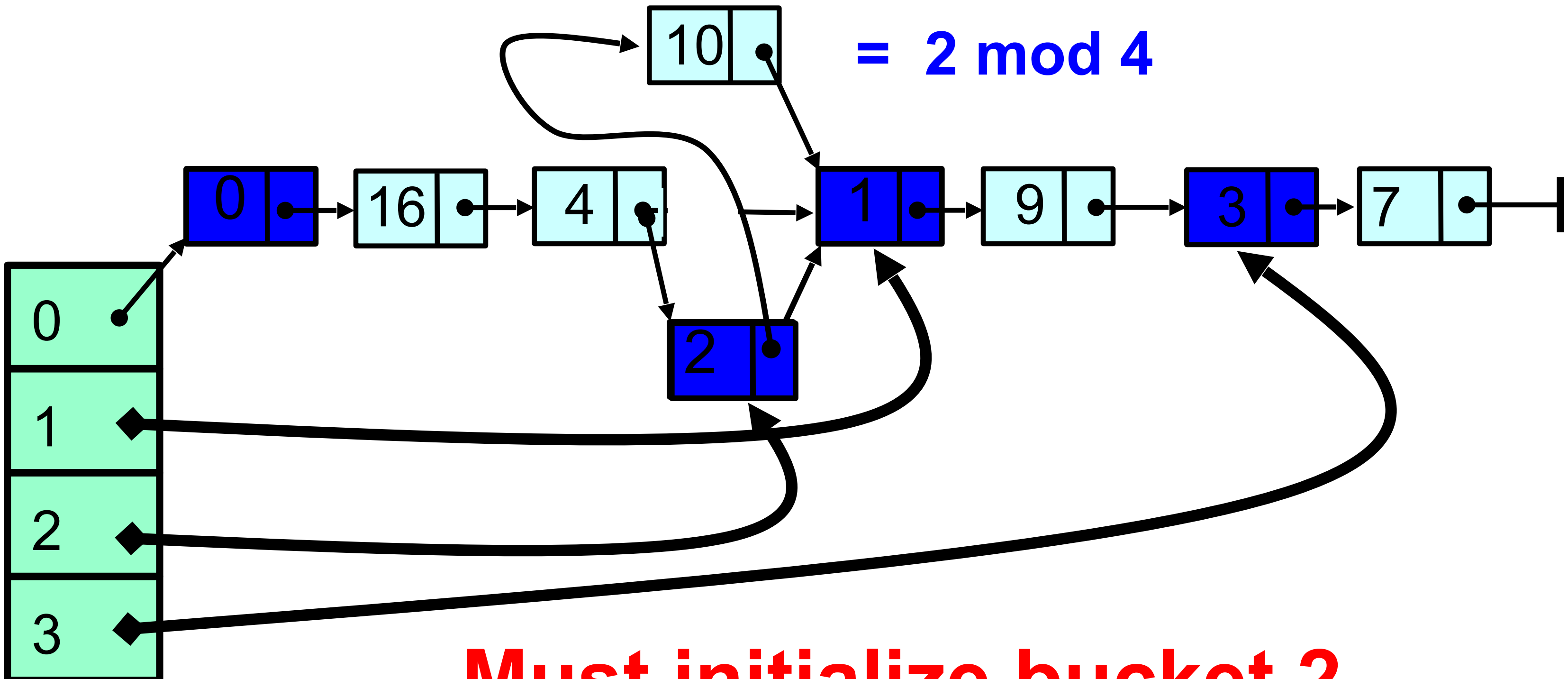


Initialization of Buckets



Need to initialize bucket 3 to split bucket 1

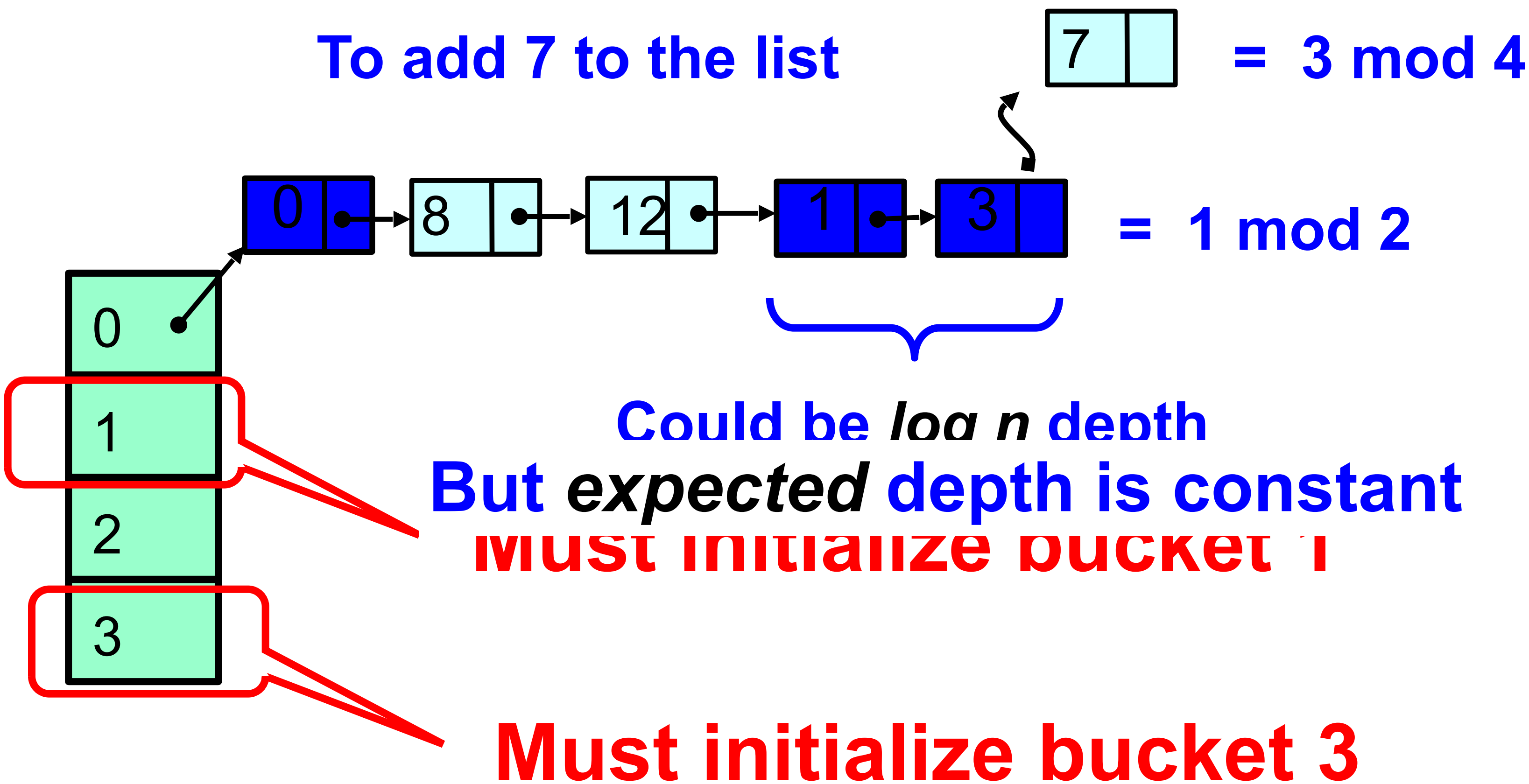
Adding 10



Must initialize bucket 2

Before adding 10

Recursive Initialization



Resize

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

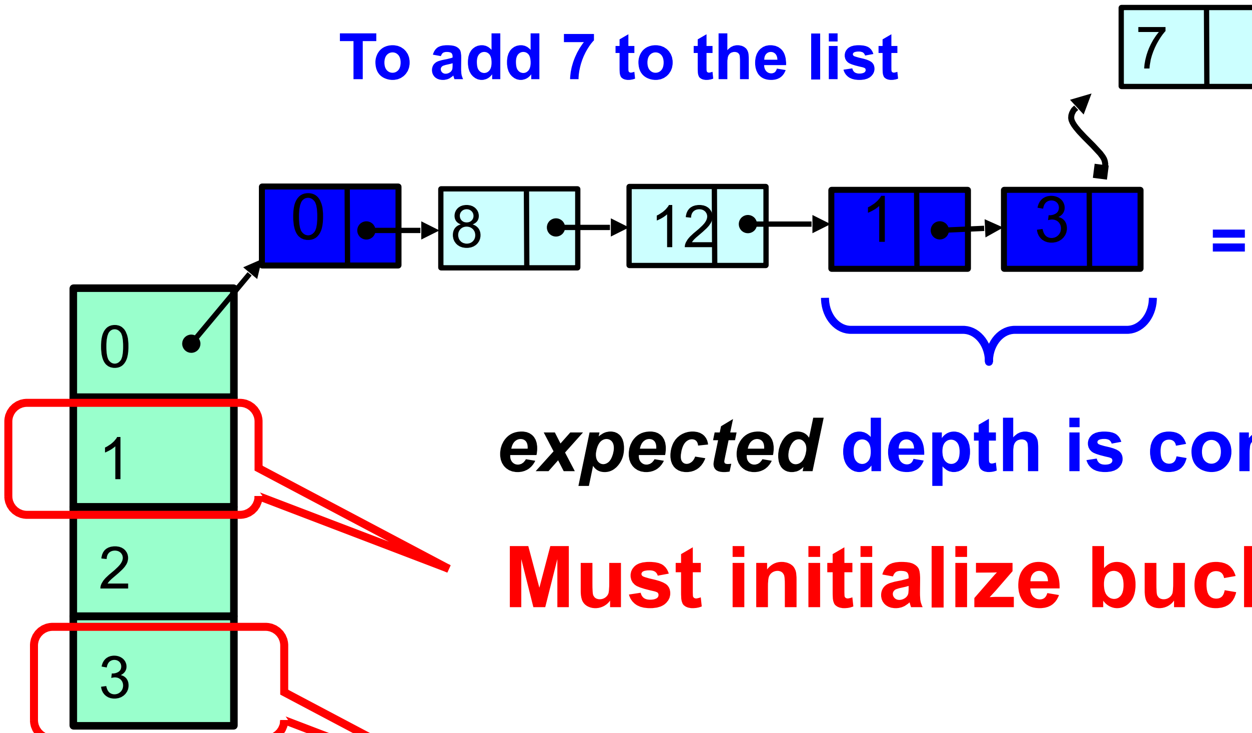
Initialize Buckets

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

Recall: Recursive Initialization

To add 7 to the list

= 3 mod 4



= 1 mod 2

expected **depth is constant**

Must initialize bucket 1

Must initialize bucket 3

Correctness

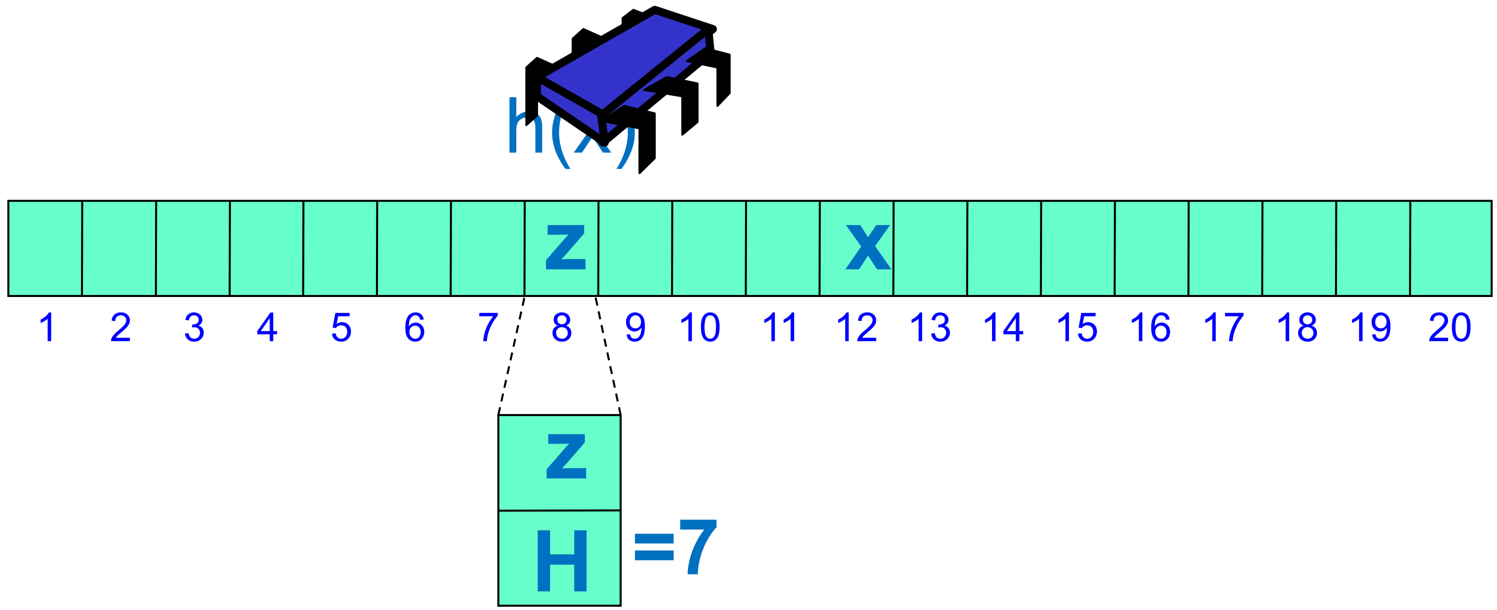
- Linearizable concurrent set
- Theorem: O(1) expected time
  - No more than O(1) items expected between two dummy nodes on average
  - Lazy initialization causes at most O(1) expected recursion depth in `initializeBucket()`

Closed (Chained) Hashing

- Advantages:
  - with N buckets, M items, Uniform h
  - retains good performance as table density (M/N) increases → less resizing
- Disadvantages:
  - dynamic memory allocation
  - bad cache behavior (no locality)

Oh, did we mention that cache behavior matters on a multicore?

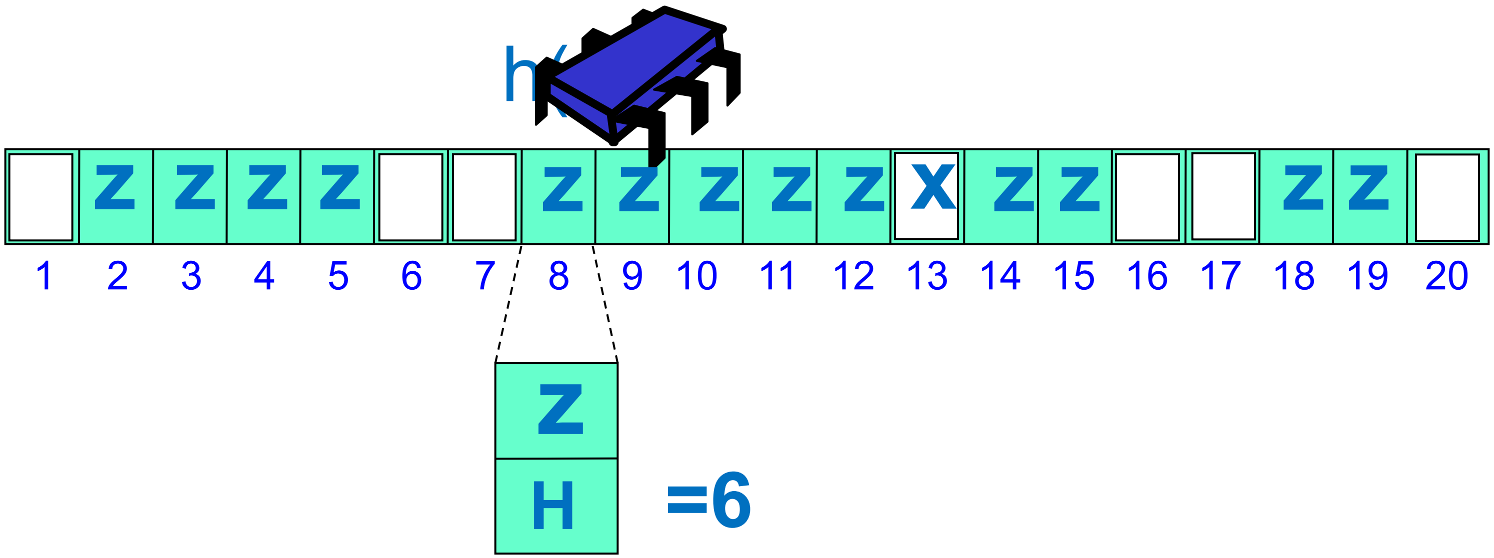
Linear Probing*



contains(x) – search linearly from h(x) to h(x) + H recorded in bucket.

*Attributed to Amdahl…

Linear Probing



add(x) – put in first empty bucket, and update H.

Linear Probing

- Open address means M · N
- Expected items in bucket same as Chaining
- Expected distance till open slot:

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

M/N = 0.5 ➔ *search 2.5 buckets*

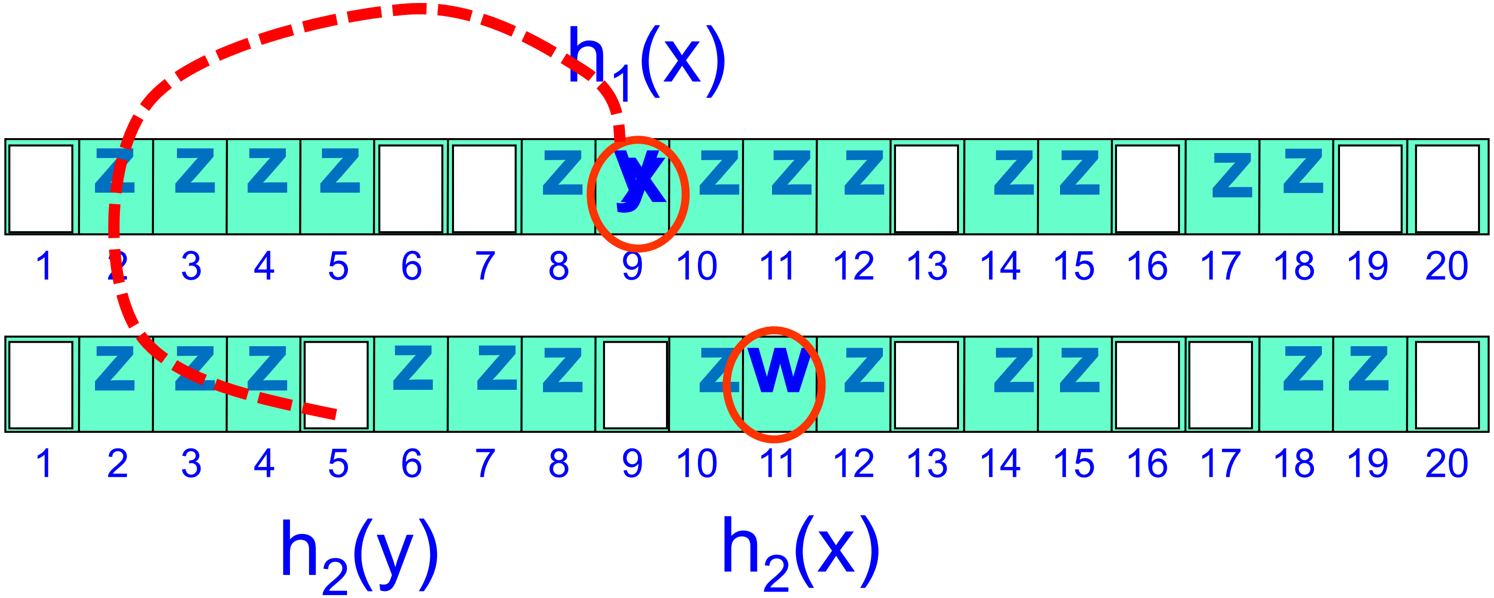
M/N = 0.9 ➔ *search 50 buckets*

Linear Probing

- Advantages:
  - Good locality ➔ fewer cache misses
- Disadvantages:
  - As M/N increases more cache misses
    - searching 10s of unrelated buckets
    - “Clustering” of keys into neighboring buckets
  - As computation proceeds “Contamination” by deleted items ➔ more cache misses

Cuckoo Hashing

But cycles can form



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.

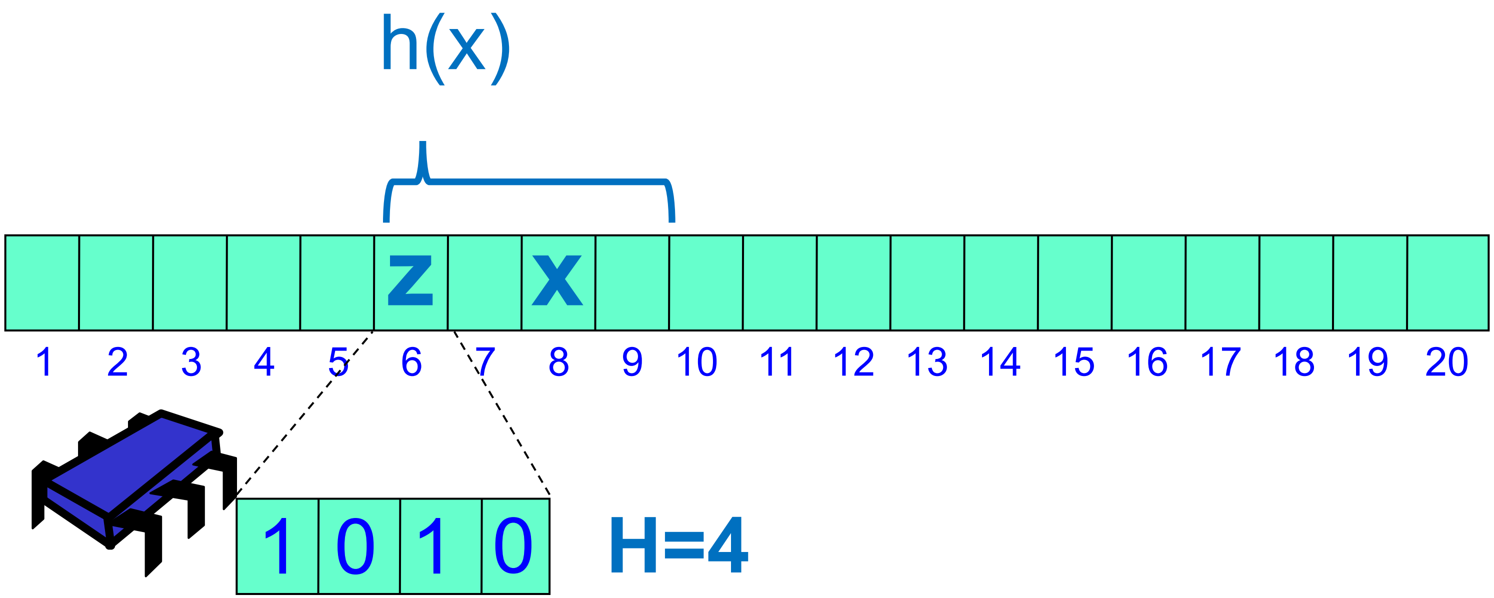
Cuckoo Hashing

- 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!

Hopscotch Hashing

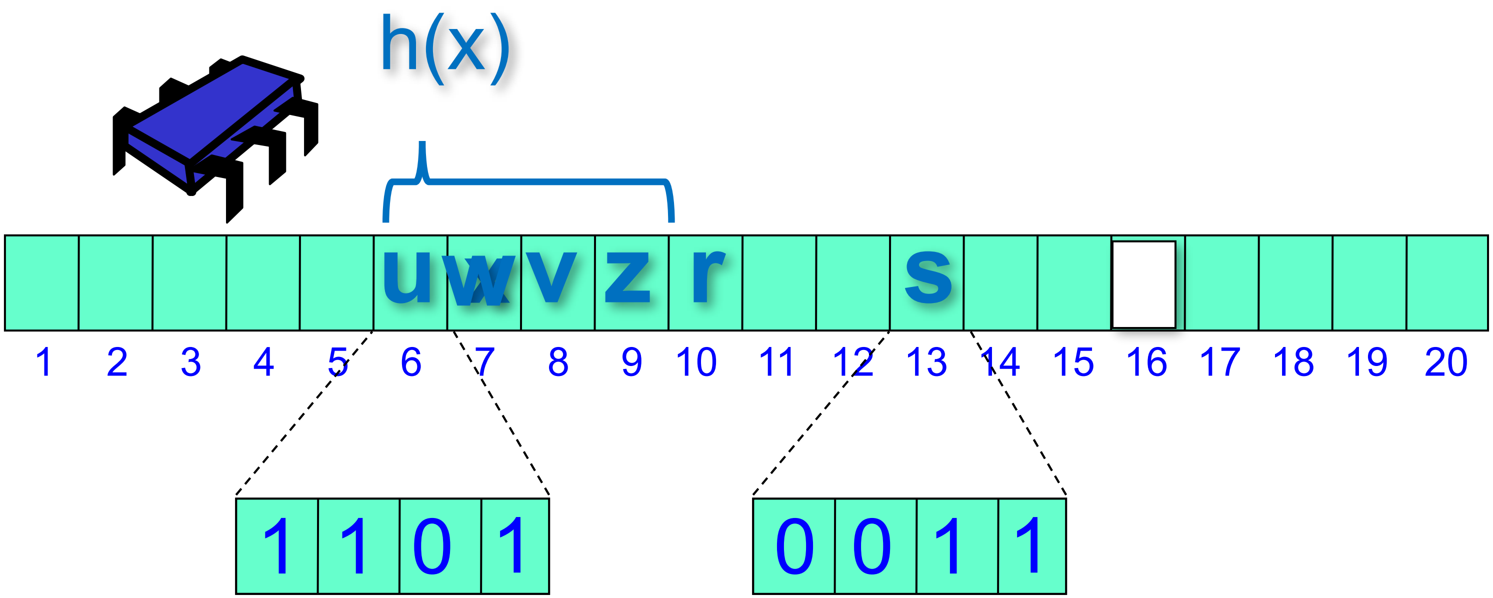
- Single Array, Simple hash function
- Idea: define neighborhood of original bucket
- In neighborhood items found quickly
- Use sequences of displacements to move items into their neighborhood

Hopscotch Hashing



contains(x) – search in at most H buckets (the hop-range) based on hop-info bitmap. In practice pick H to be 32.

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).

Hopscotch Hashing

- *contains*
  - wait-free, just look in neighborhood

Hopscotch Hashing

- *contains*
  - wait-free, just look in neighborhood
- *add*
  - expected distance same as in linear probing

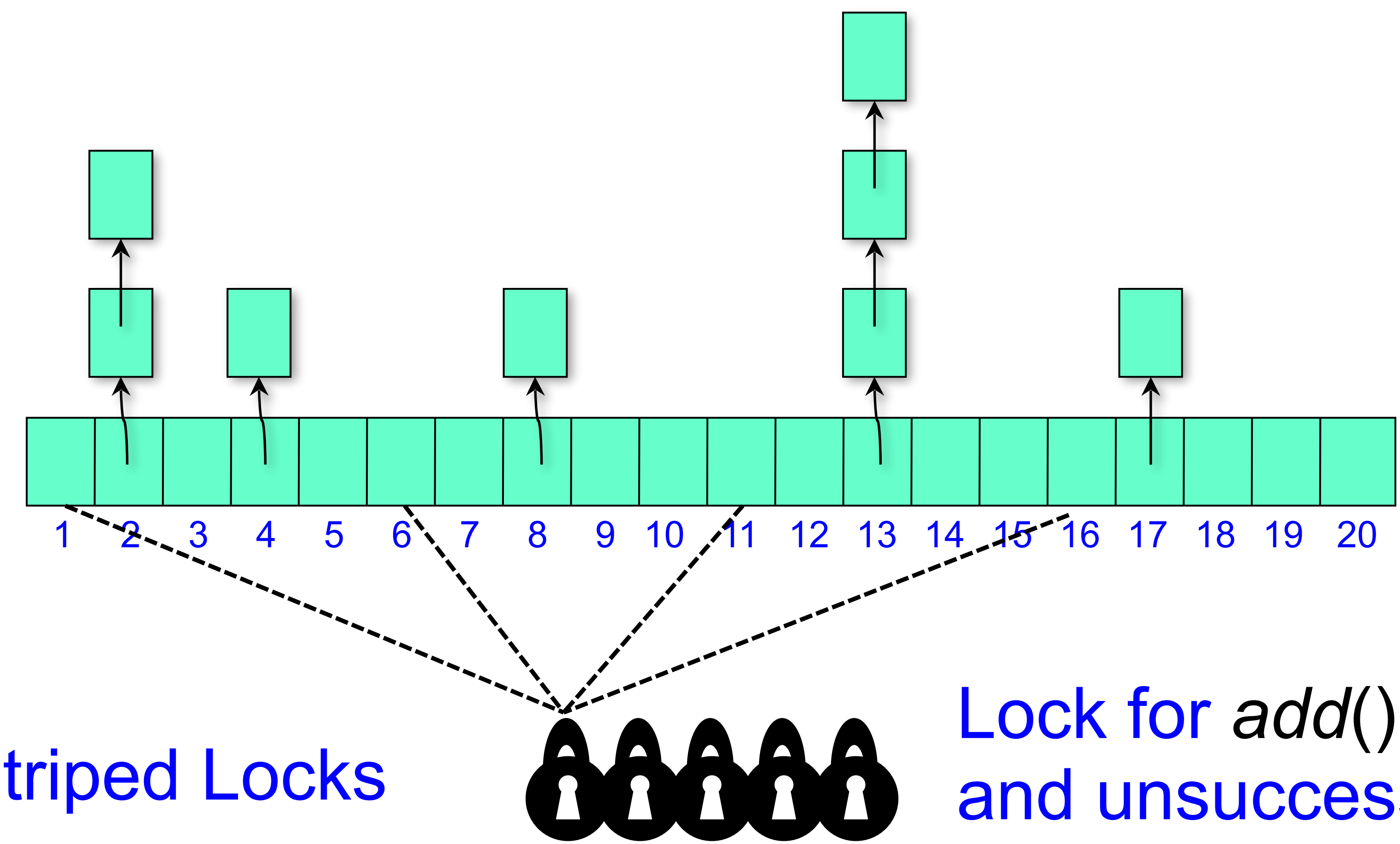
Hopscotch Hashing

- *contains*
  - wait-free, just look in neighborhood
- *add*
  - Expected distance same as in linear probing
- *resize*
  - neighborhood full less likely as H → log n
  - one word hop-info bitmap, or use smaller H and default to linear probing

Advantages

- Good locality and cache behavior
- As table density (M/N) increases → less resizing
- Move cost to *add*() from *contains*()
- Easy to parallelize

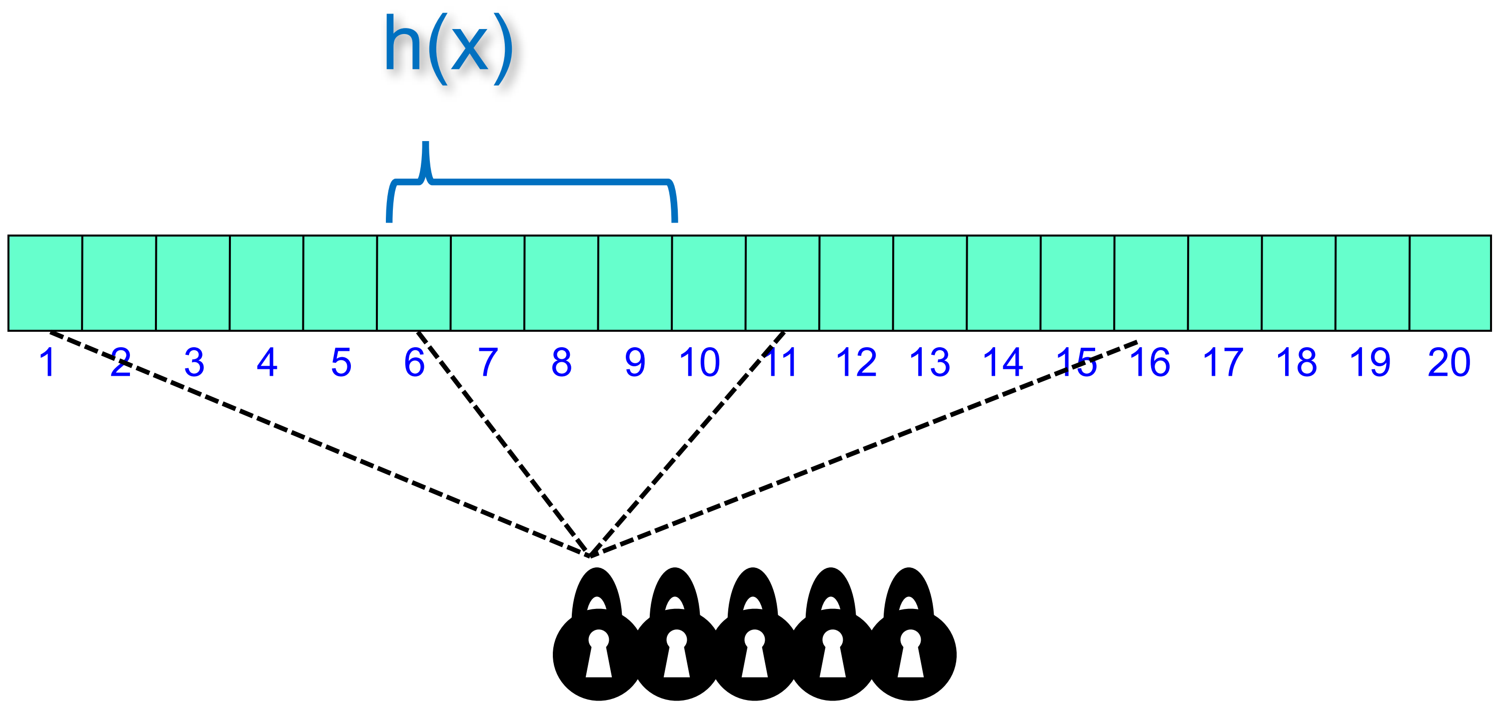
Recall: Concurrent Chained Hashing



Striped Locks

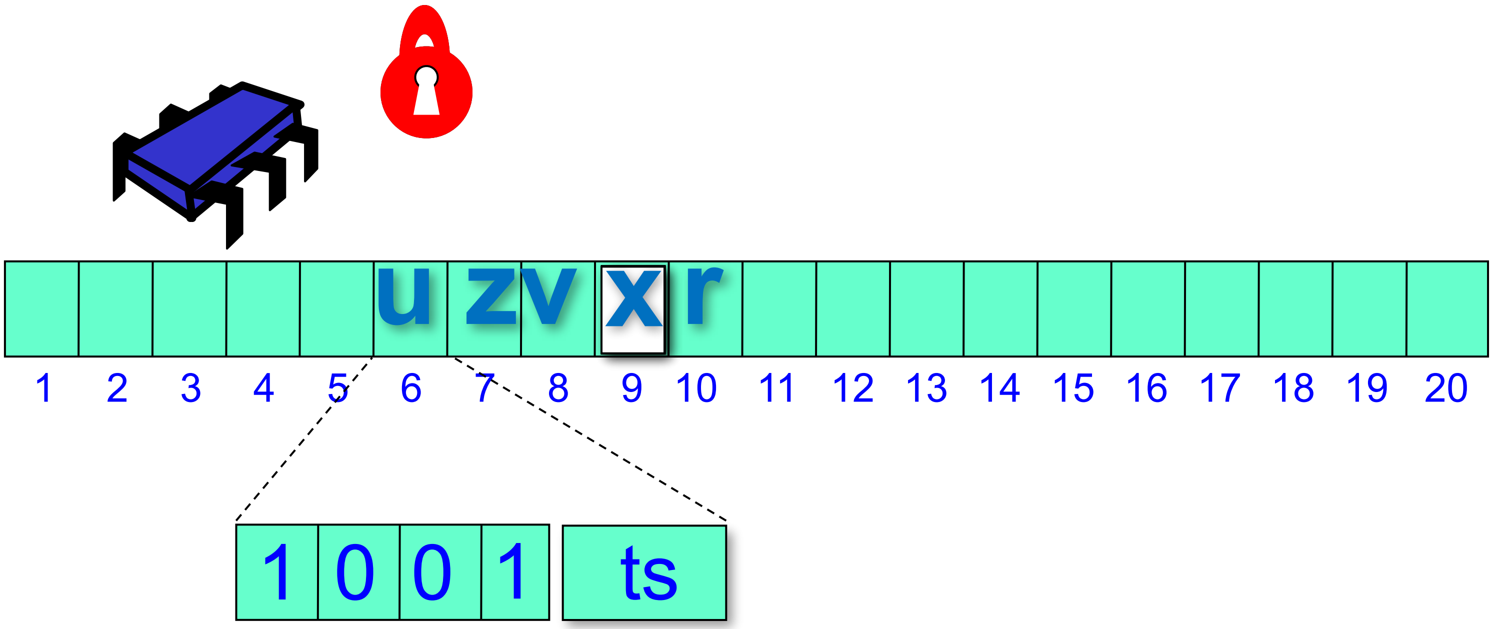
Lock for *add*() and unsuccessful *contains*()

Concurrent Simple Hopscotch



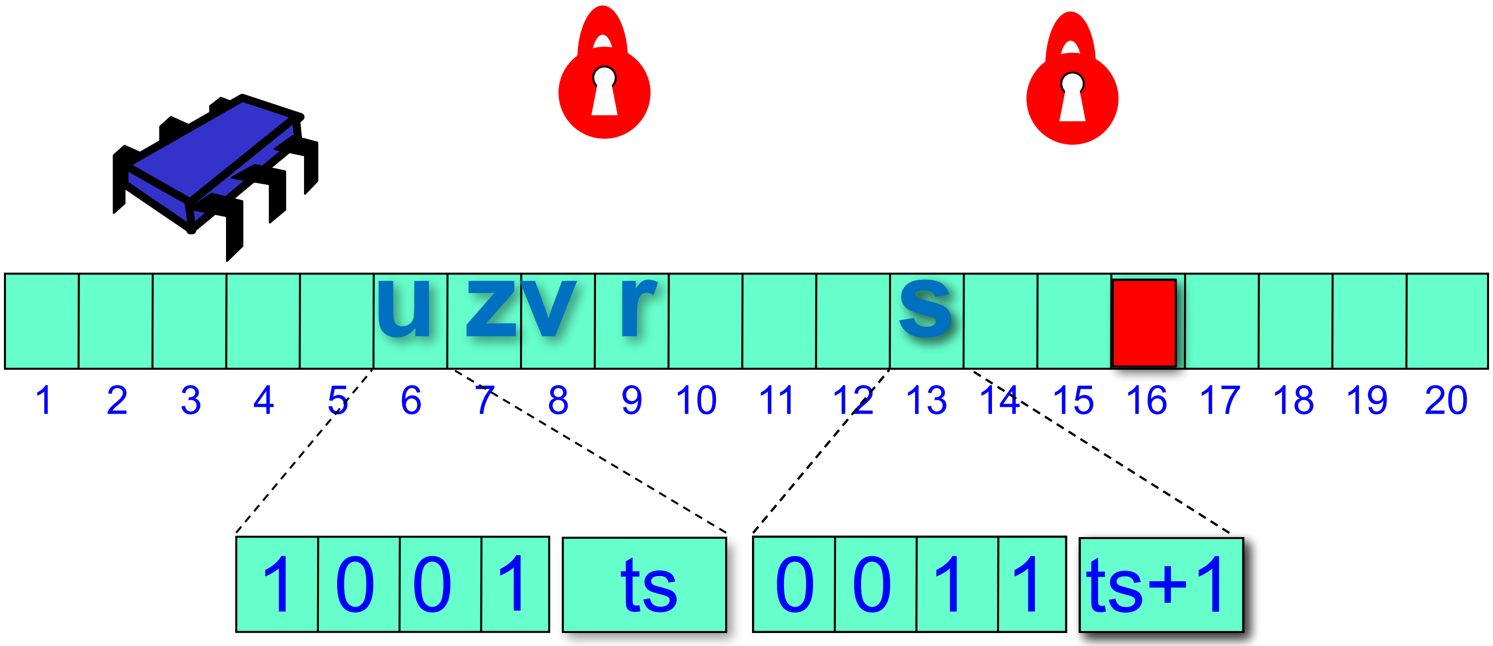
contains() is wait-free

Concurrent Simple Hopscotch



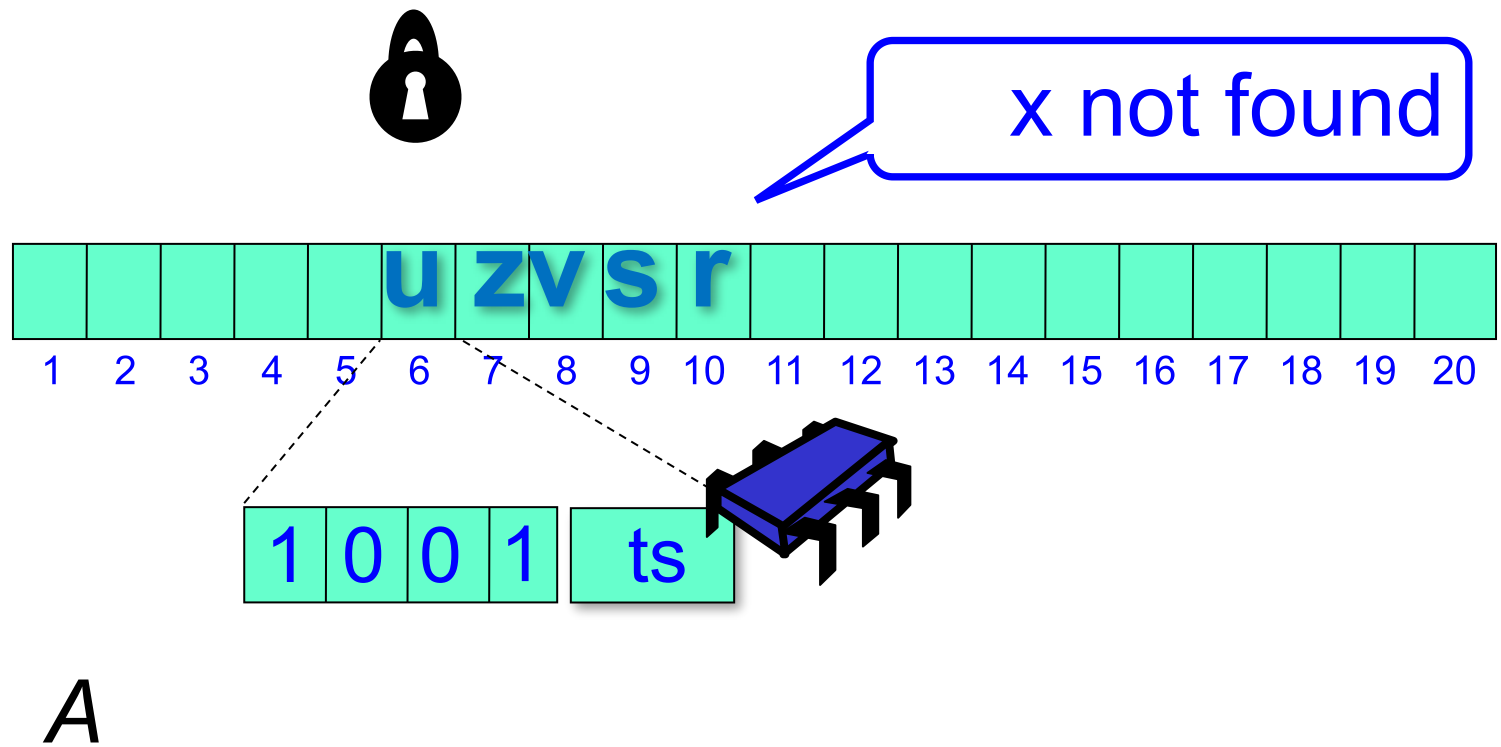
Add(x) – lock bucket, mark empty slot using CAS, add x erasing mark

Concurrent Simple Hopscotch



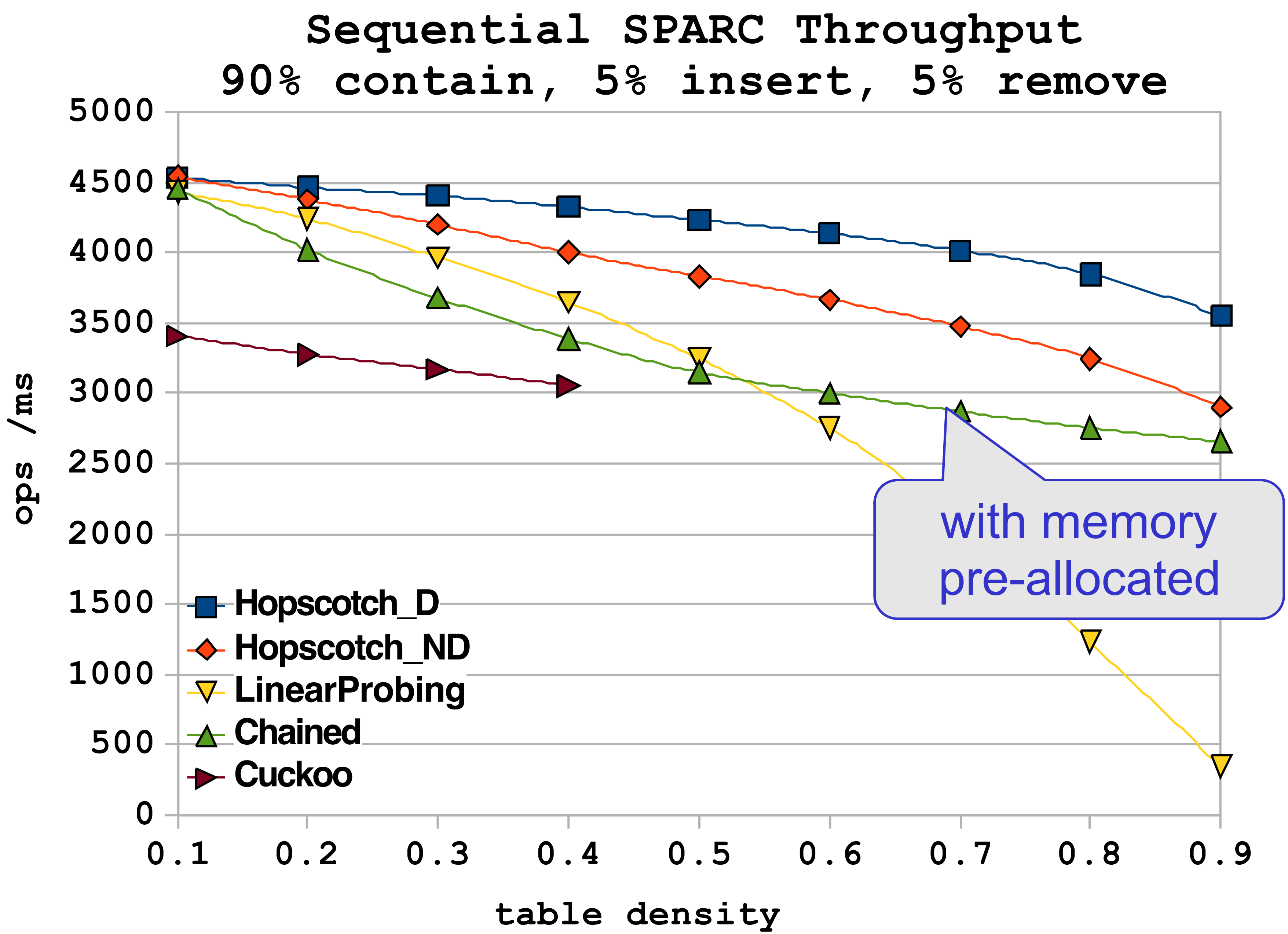
add(x) – lock bucket, mark empty slot using CAS, lock bucket and update timestamp of bucket being displaced before erasing old value

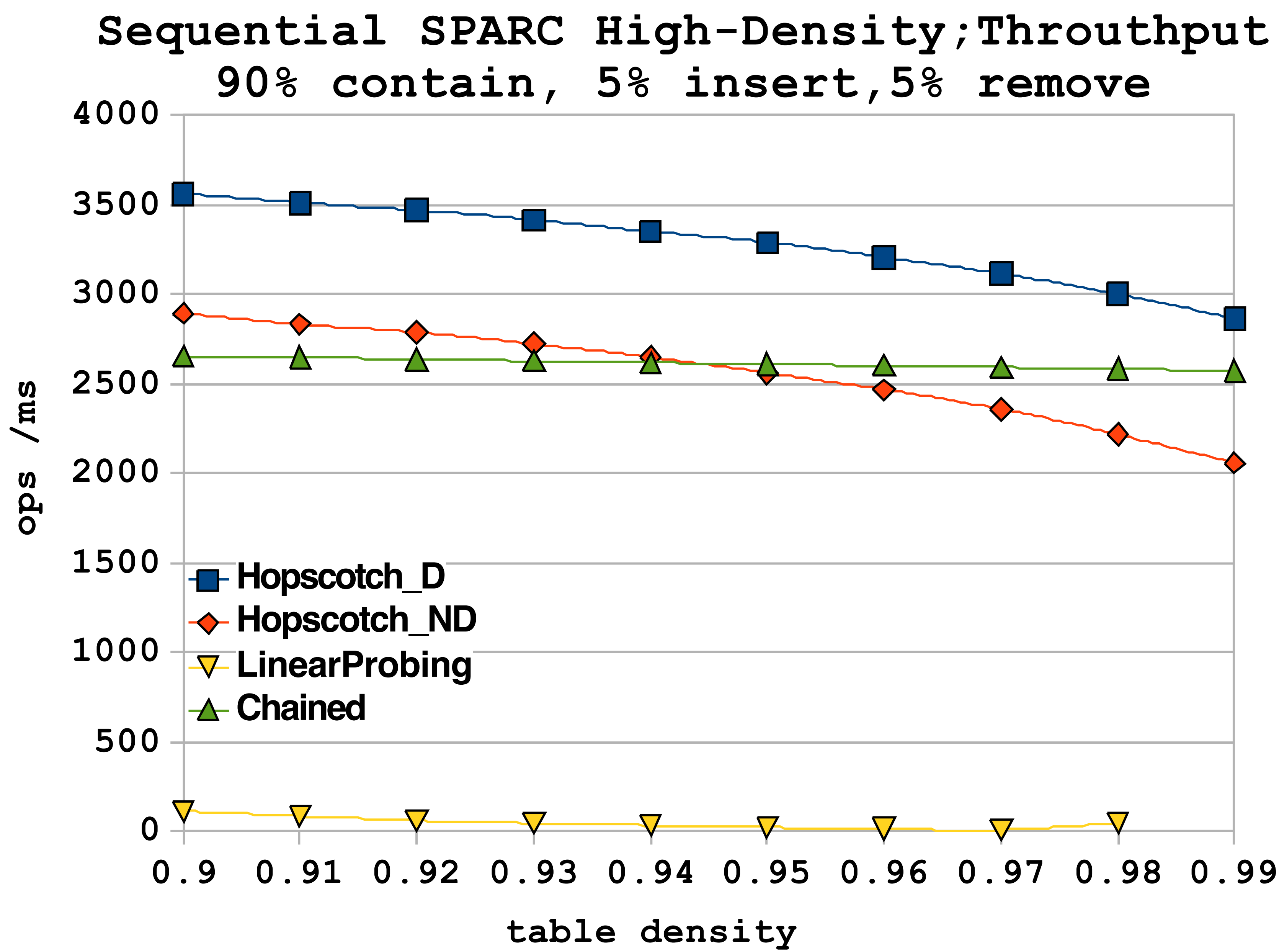
Concurrent Simple Hopscotch



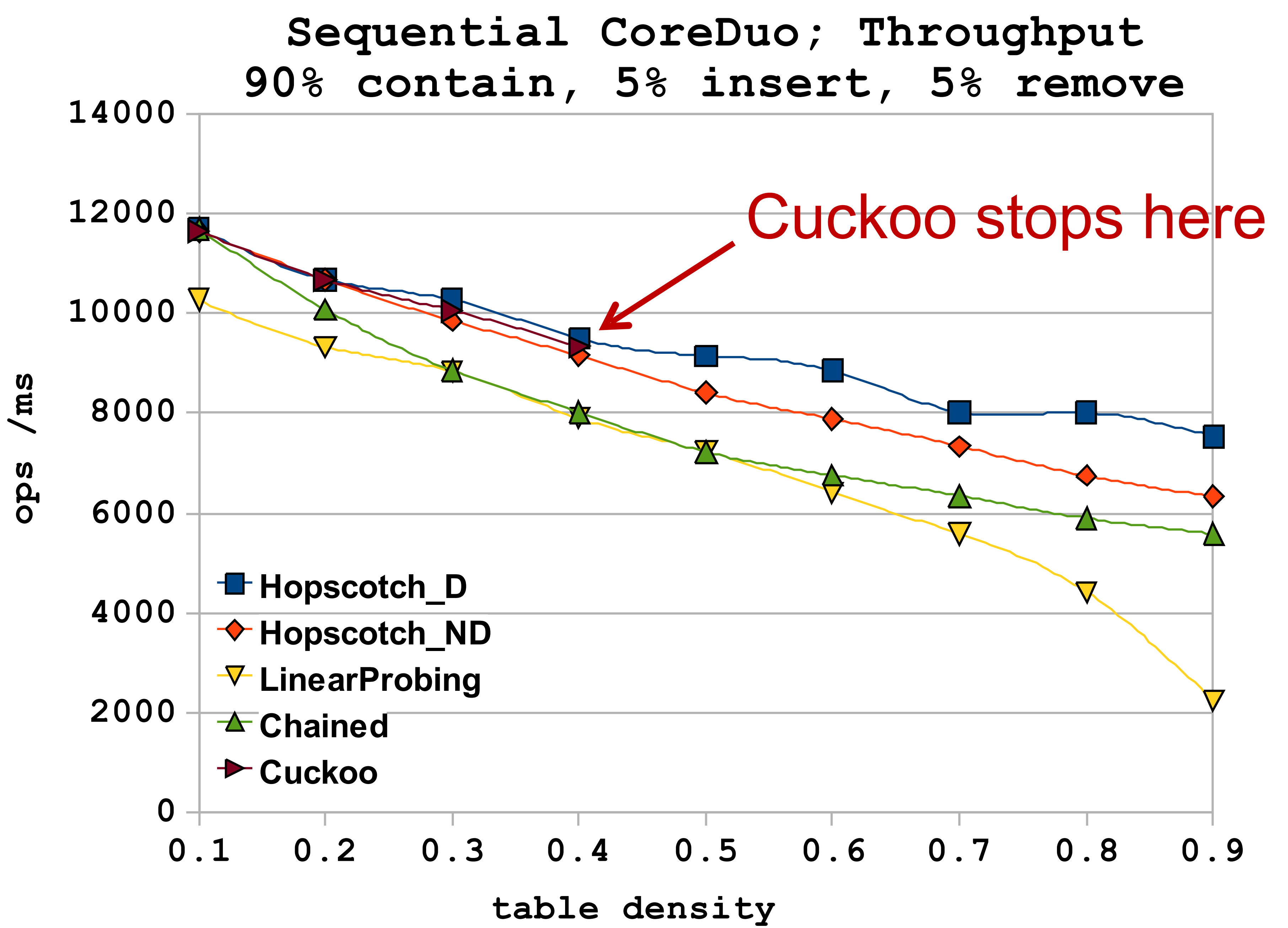
Is performance dominated by cache behavior?

- Run algs on state of the art multicores and uniprocessors:
  - Sun 64 way Niagara II, and
  - Intel 3GHz Xeon
- Benchmarks pre-allocated memory to eliminate effects of memory management

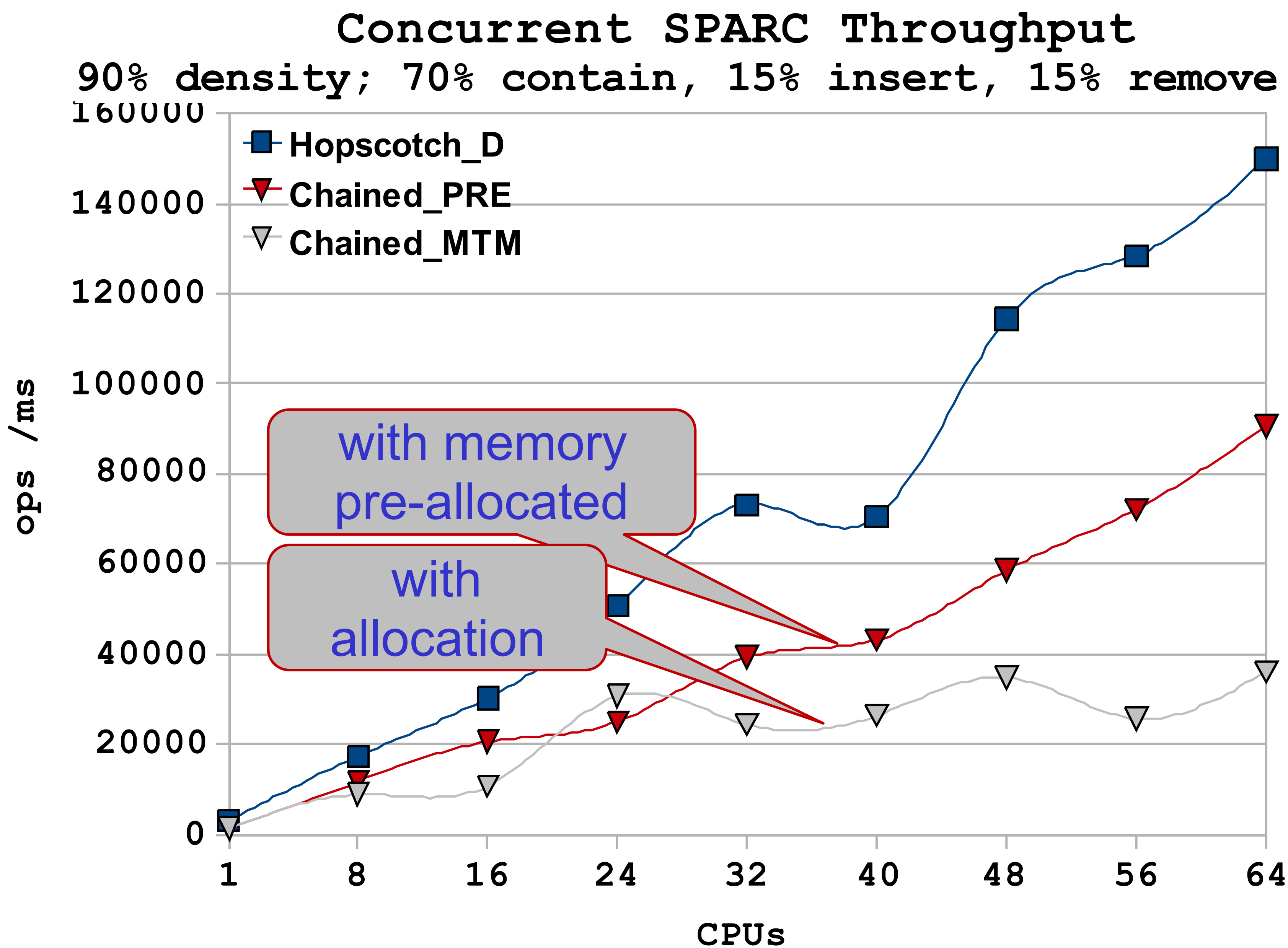

Sequential SPARC Throughput
90% contain, 5% insert, 5% remove
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
ops /ms
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
table density
Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo
with memory pre-allocated

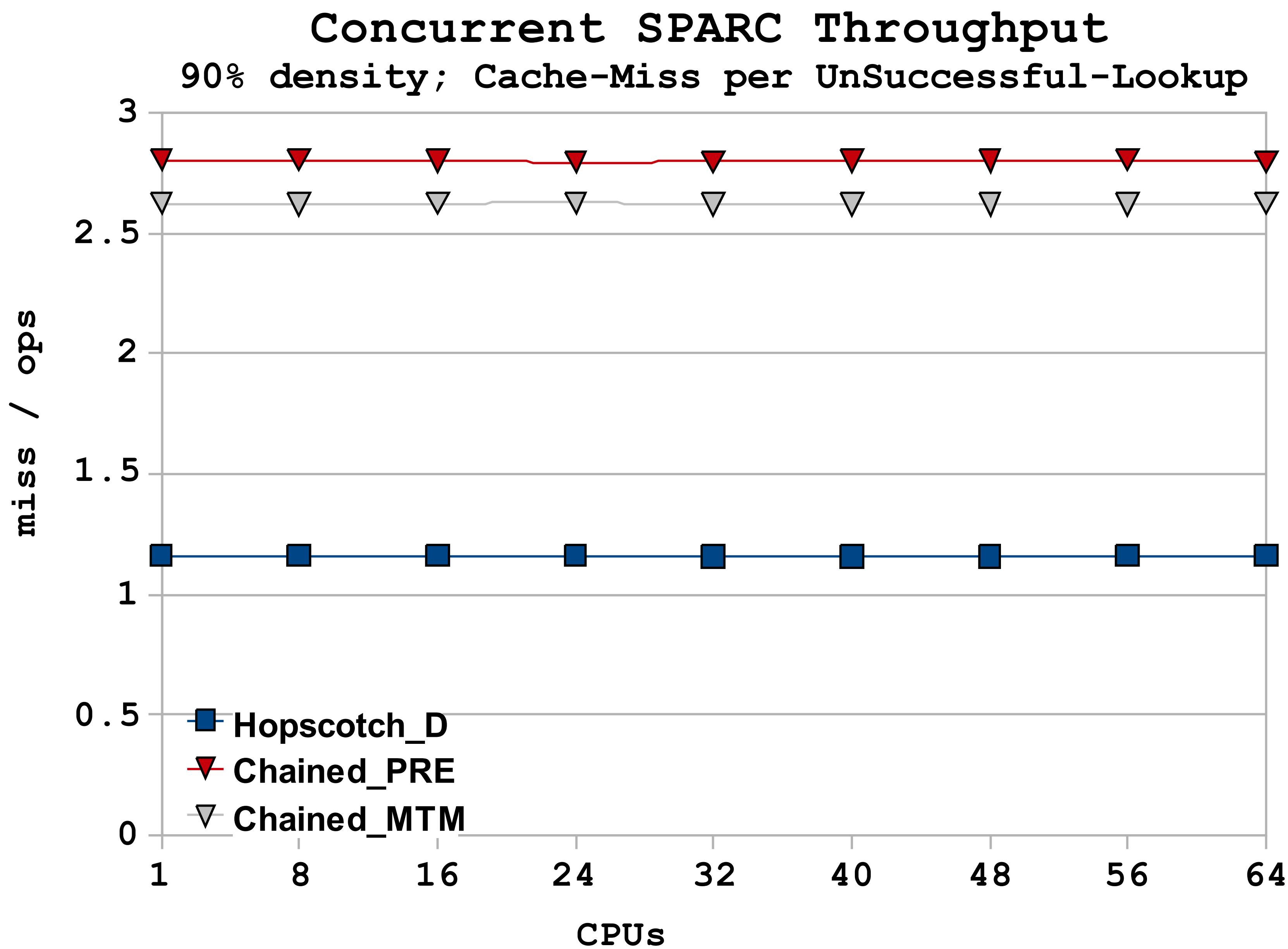


Sequential SPARC High-Density;Throuthput
90% contain, 5% insert,5% remove
4000
3500
3000
2500
2000
1500
1000
500
0
ops /ms
0.9
0.91
0.92
0.93
0.94
0.95
0.96
0.97
0.98
0.99
table density
Hopscotch_D
Hopscotch_ND
LinearProbing
Chained



Sequential CoreDuo; Throughput
90% contain, 5% insert, 5% remove
Cuckoo stops here
ops /ms
table density
14000
12000
10000
8000
6000
4000
2000
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo


Concurrent SPARC Throughput
90% density; 70% contain, 15% insert, 15% remove
Hopscotch_D
Chained_PRE
Chained_MTM
with memory pre-allocated
with allocation
ops /ms
CPUs
0
20000
40000
60000
80000
100000
120000
140000
160000
1
8
16
24
32
40
48
56
64


Concurrent SPARC Throughput
90% density; Cache-Miss per UnSuccessful-Lookup
miss / ops
3
2.5
2
1.5
1
0.5
0
1
8
16
24
32
40
48
56
64
CPUs
Hopscotch_D
Chained_PRE
Chained_MTM

Summary

- Chained hash with striped locking is simple and effective in many cases
- Hopscotch with striped locking great cache behavior
- If incremental resizing needed go for split-ordered