# CS331 Tutorial 4

# Java : CDS

A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

# Outline

- Four Way to implement Lock in Java
- Concurrent List
- Concurrent Hash
- Assignment-II
  - Use may use C-List and C-Hash

# Four ways to implement a synchronized counter in Java

- **Suppose there is a  Shared counter and every threads are attempting to manipulate**
  1. **Synchronized Block**
  2. **Atomic Variable**
  3. **Concurrent Lock**
  4. **Semaphore**

# First: Fine-Grained Synchronization

- Instead of using a single lock …
- Split object into
  - Independently-synchronized components
  - **Example: Hash : Modification**
- Methods conflict when they access
  - The same component …
  - At the same time

# Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive

# Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

# Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …

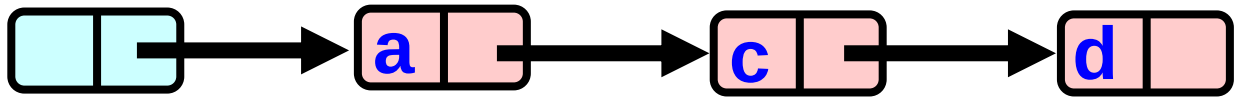
# Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

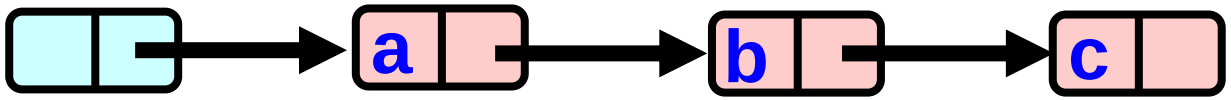
# Concurrent List

# Sequential List Based Set

**Add()**

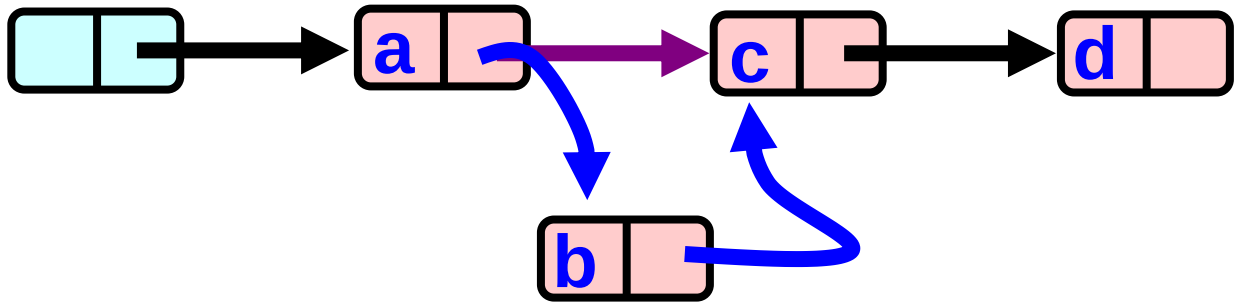



**Remove()**

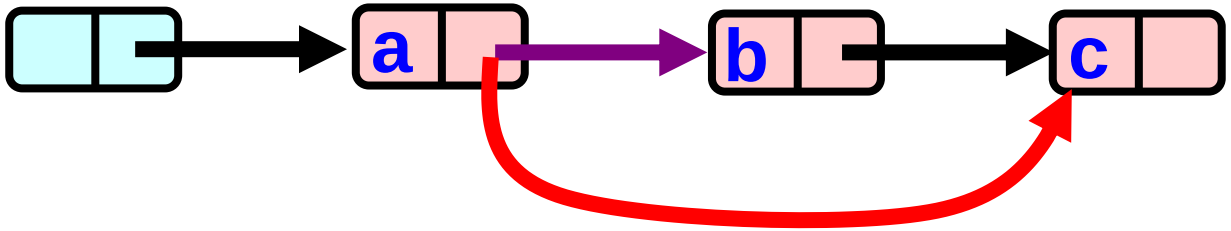


# Sequential List Based Set

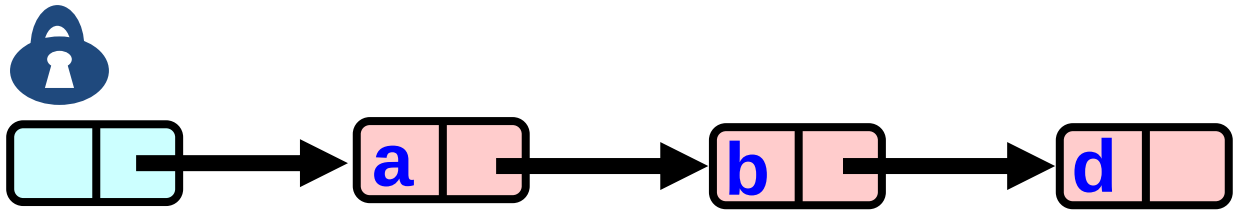
**add()**



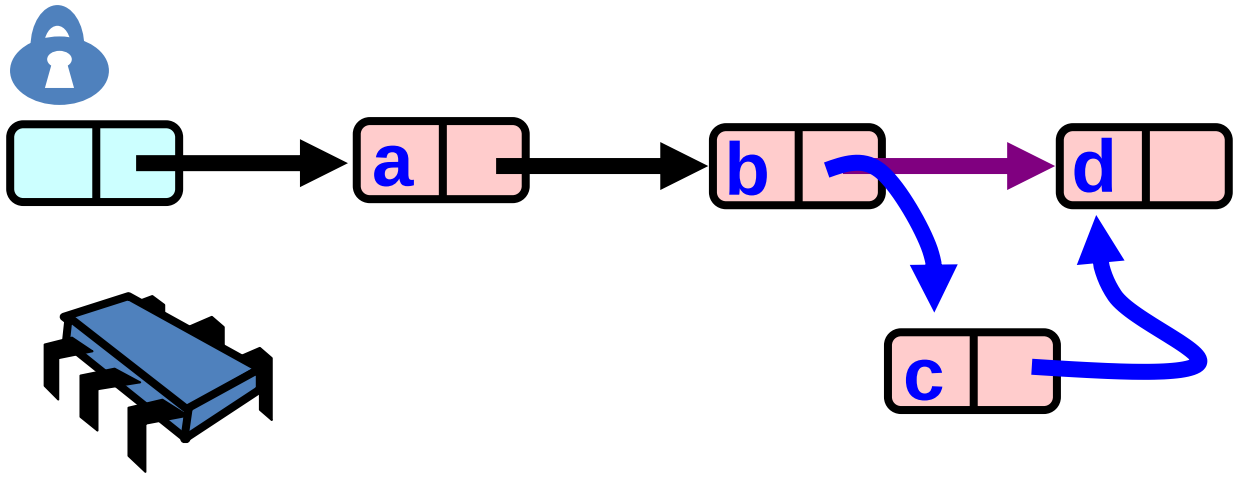

**remove()**



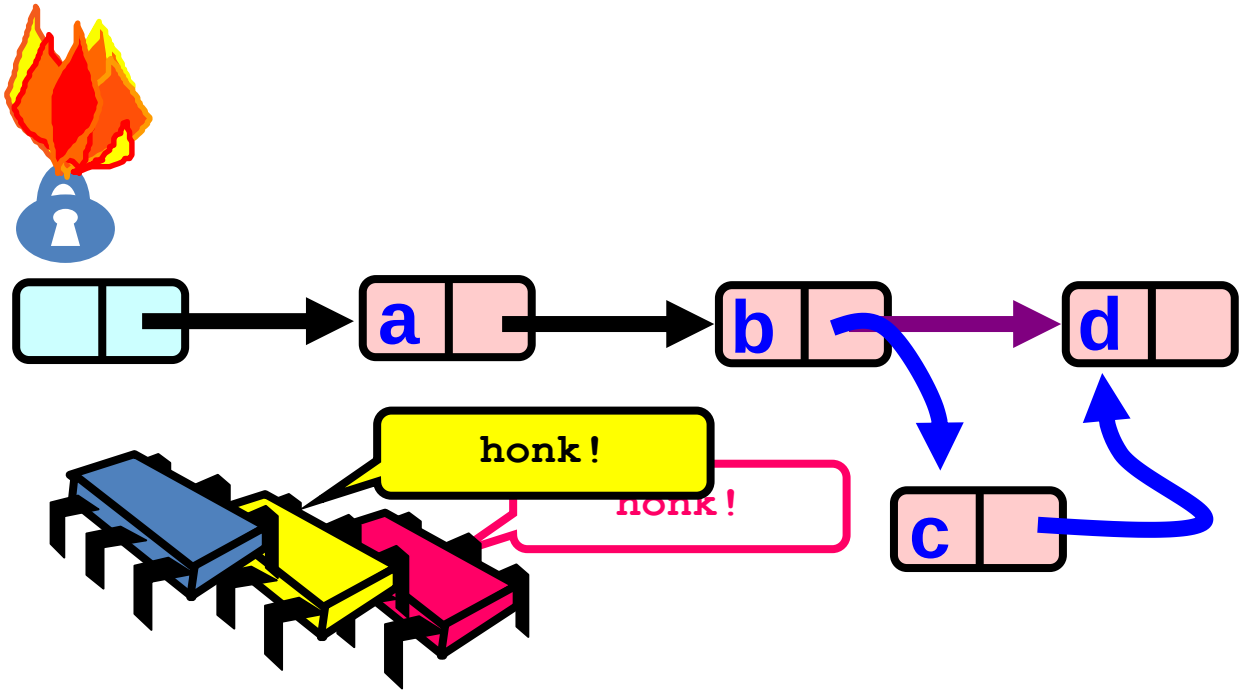
# Coarse-Grained Locking



# Coarse-Grained Locking



# Coarse-Grained Locking




Simple but hotspot + bottleneck

# List

- Suppose a list have Nodes, each node have
  - Key value : Customer Name
  - Item Value : Amount
- Node : key, value, next
- Operations frequencies
  - Balance check : 50%
  - Update amount : 49.8
  - Remove account : 0.1%
  - Add account : 0.1%

# Fine Gran locking

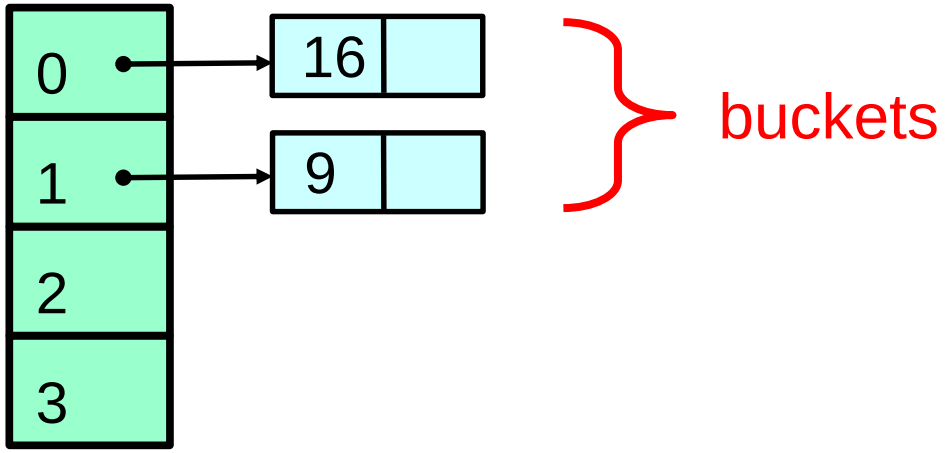
- Suppose a list have Nodes, each node have
  - Key value : Customer Name
  - Item Value : Amount
- Node : key, value, next
- Operations frequencies
  - Balance check : 50% : Lock not required
  - Update amount : 49.8 : lock of the required node
  - Remove account : 0.1% : lock entire list
  - Add account : 0.1% : lock entire list

# Fine Gran locking

- Suppose a list have Nodes, each node have
  - Key value : Customer Name
  - Item Value : Amount
- Node : key, value, next
- Operations frequencies
  - Balance check : 50% : Lock not required
  - Update amount : 49.8 : lock of the required node
  - Remove account : 0.1% : lock entire list
  - Add account : 0.1% **: lock entire list**
- If the Add and remove account contribute: 90% of operation, we need to fine grain add and remove **: Refer Book Art of Multiprocessor programming**

# Concurrent Hash

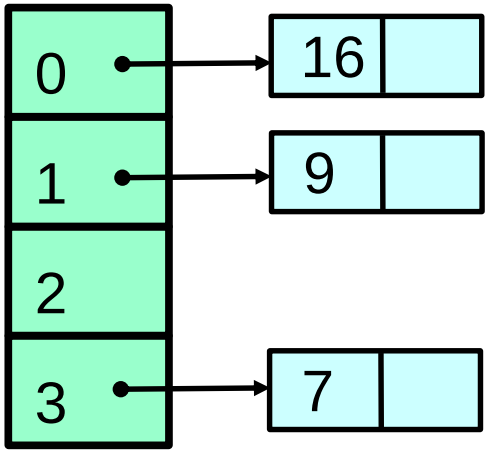
# Sequential Closed Hash Map




2 Items

h(k) = k mod 4

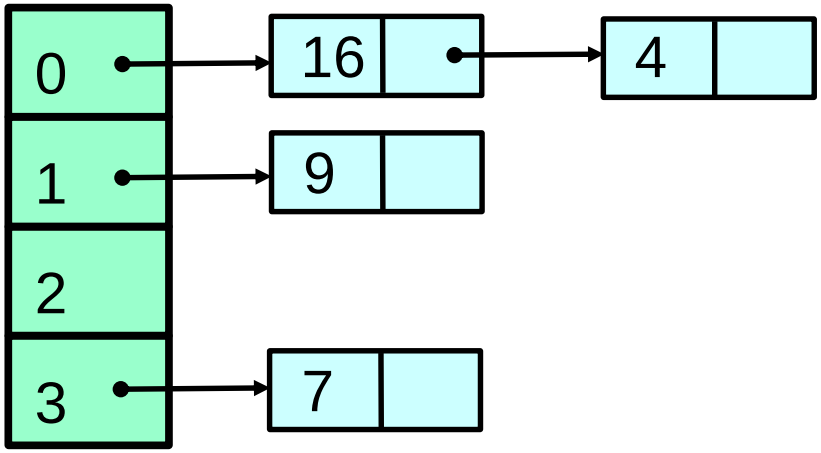
# Add an Item




3 Items

h(k) = k mod 4

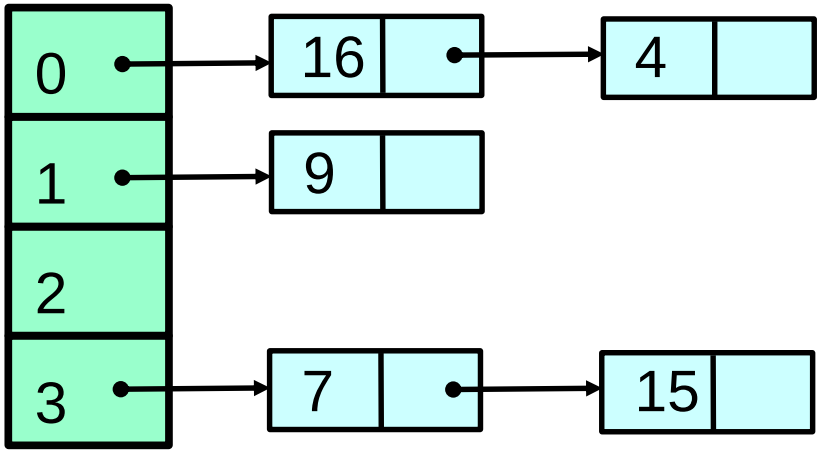
# Add Another: Collision




4 Items

h(k) = k mod 4

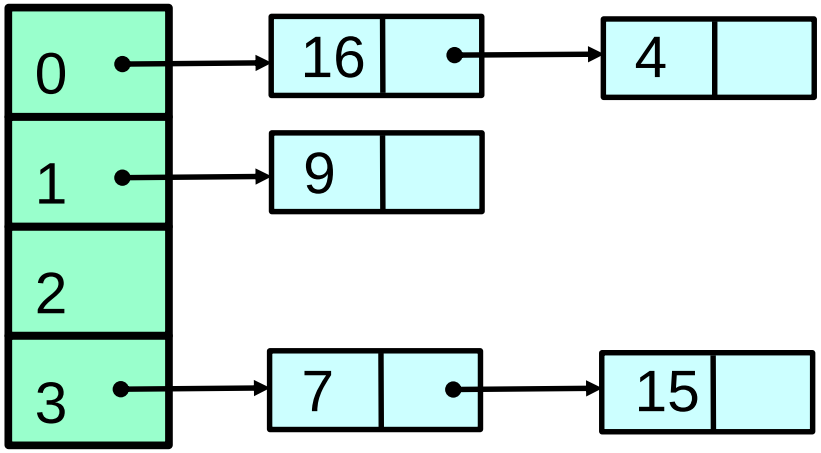
# More Collisions




5 Items

h(k) = k mod 4

# More Collisions

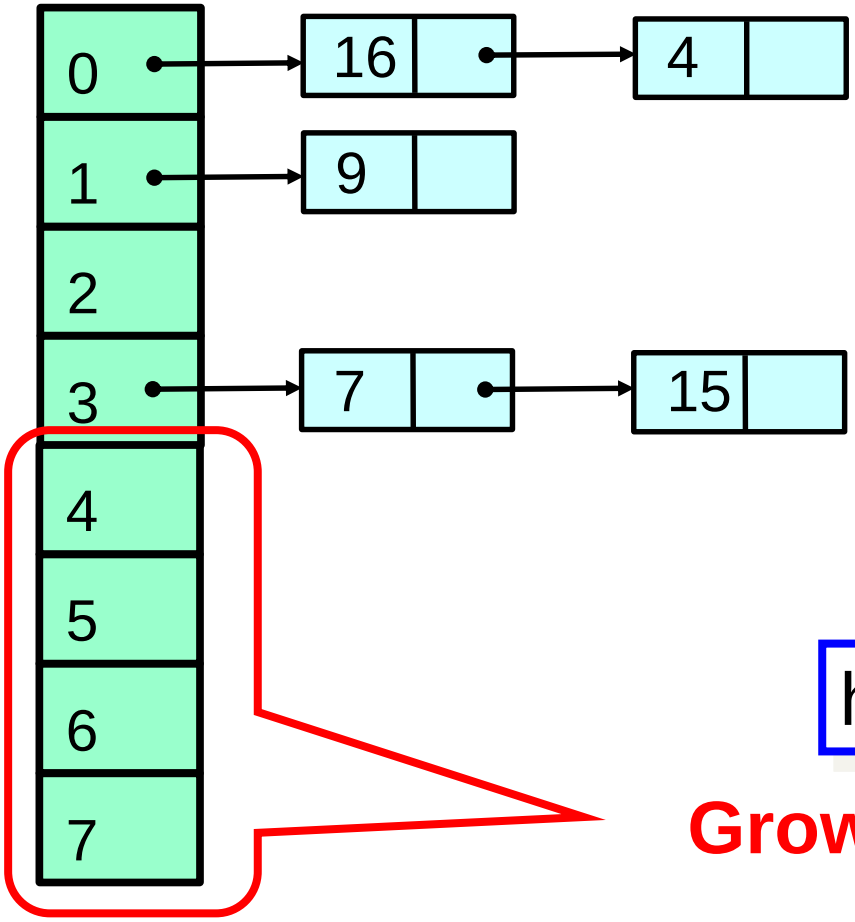



5 Items

h(k) = k mod 4

**Problem:**
**buckets getting too long**

# Resizing

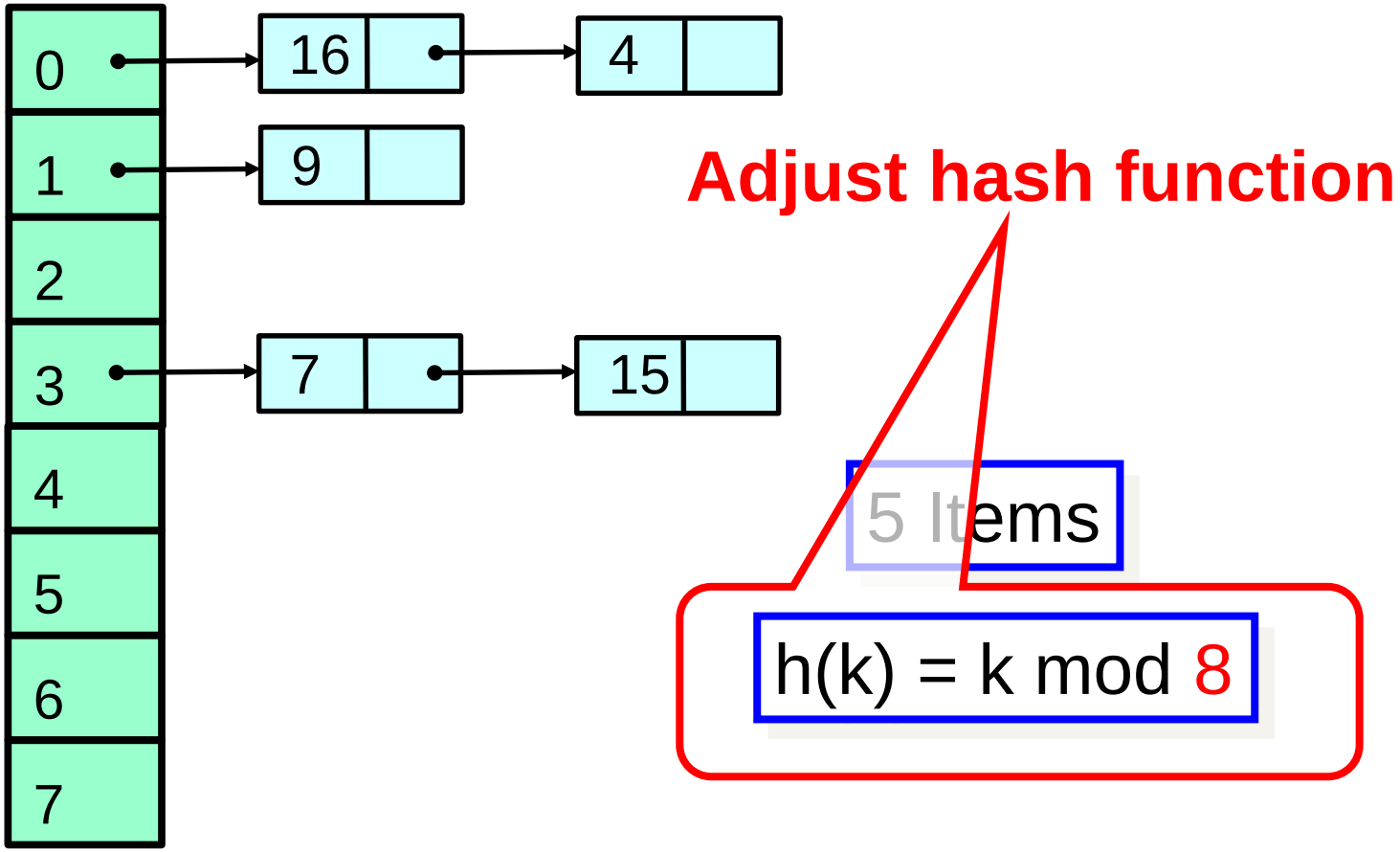



5 Items

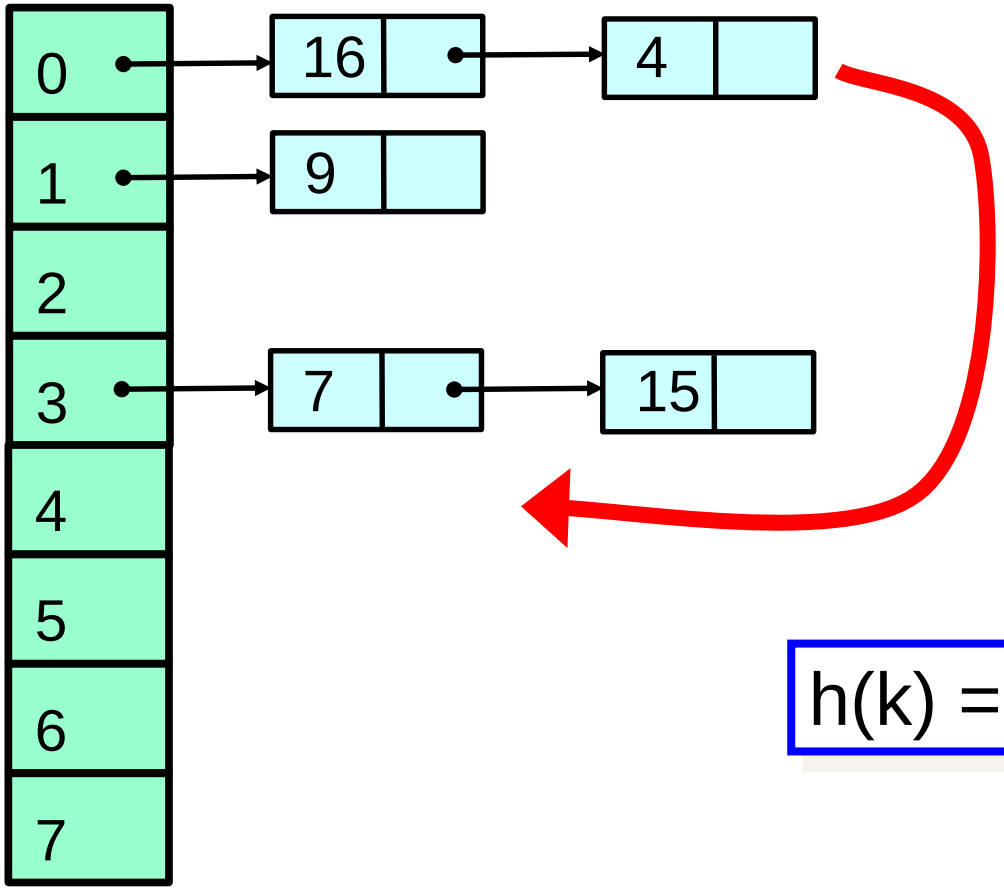
h(k) = k mod 8

**Grow the array**

# Resizing



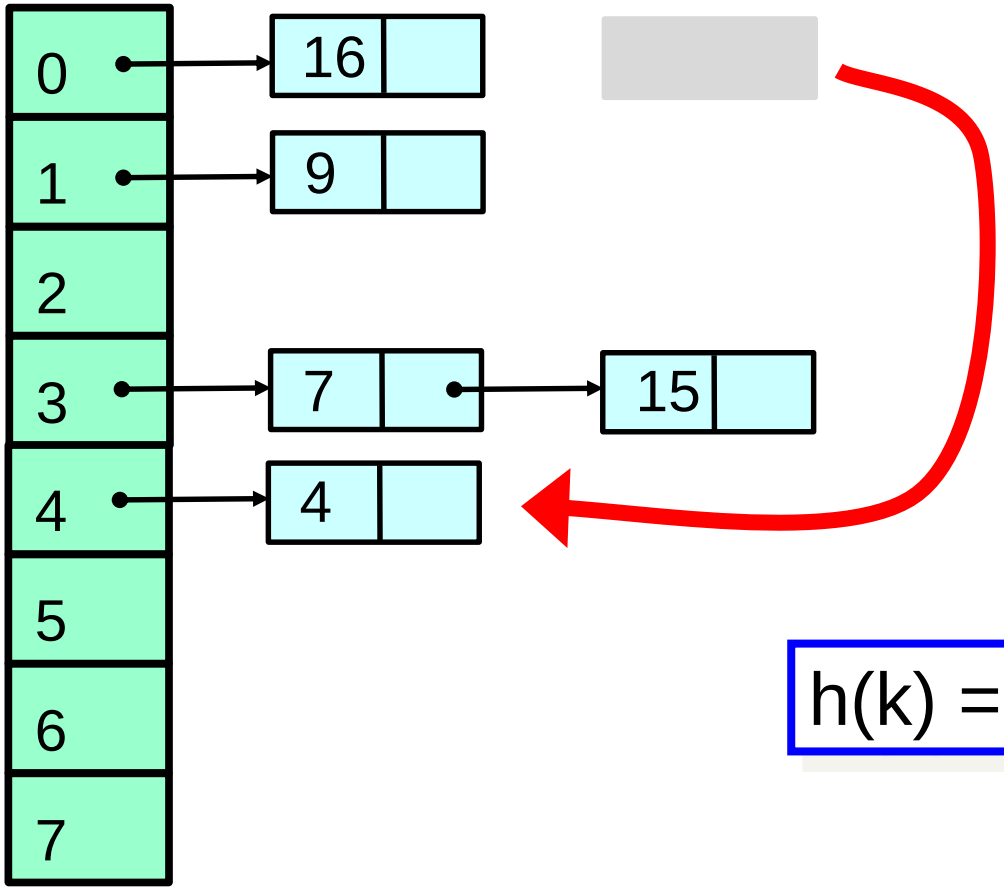
# Resizing




h(4) = 0 mod 8

h(k) = k mod 8

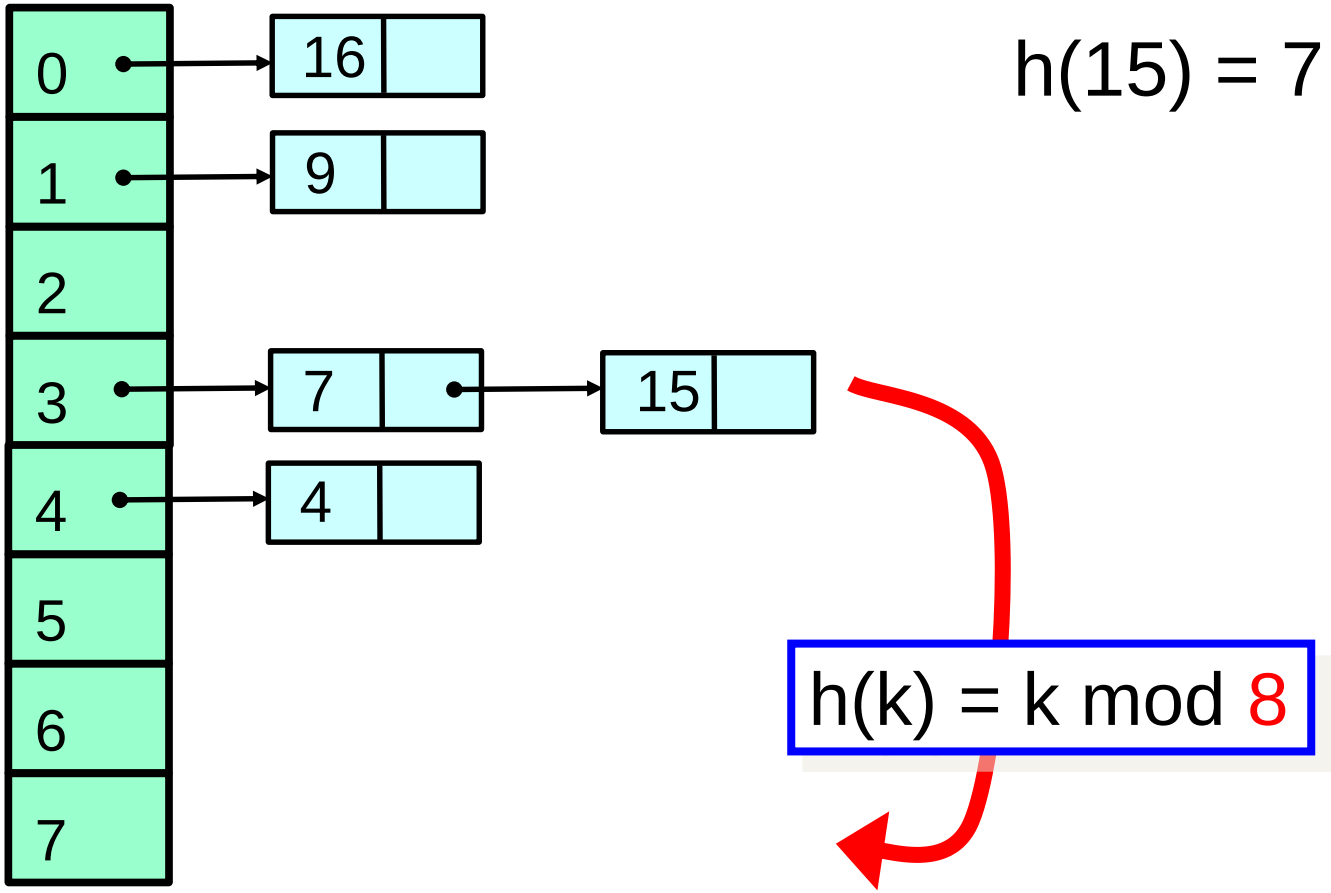
# Resizing




h(4) = 4 mod 8

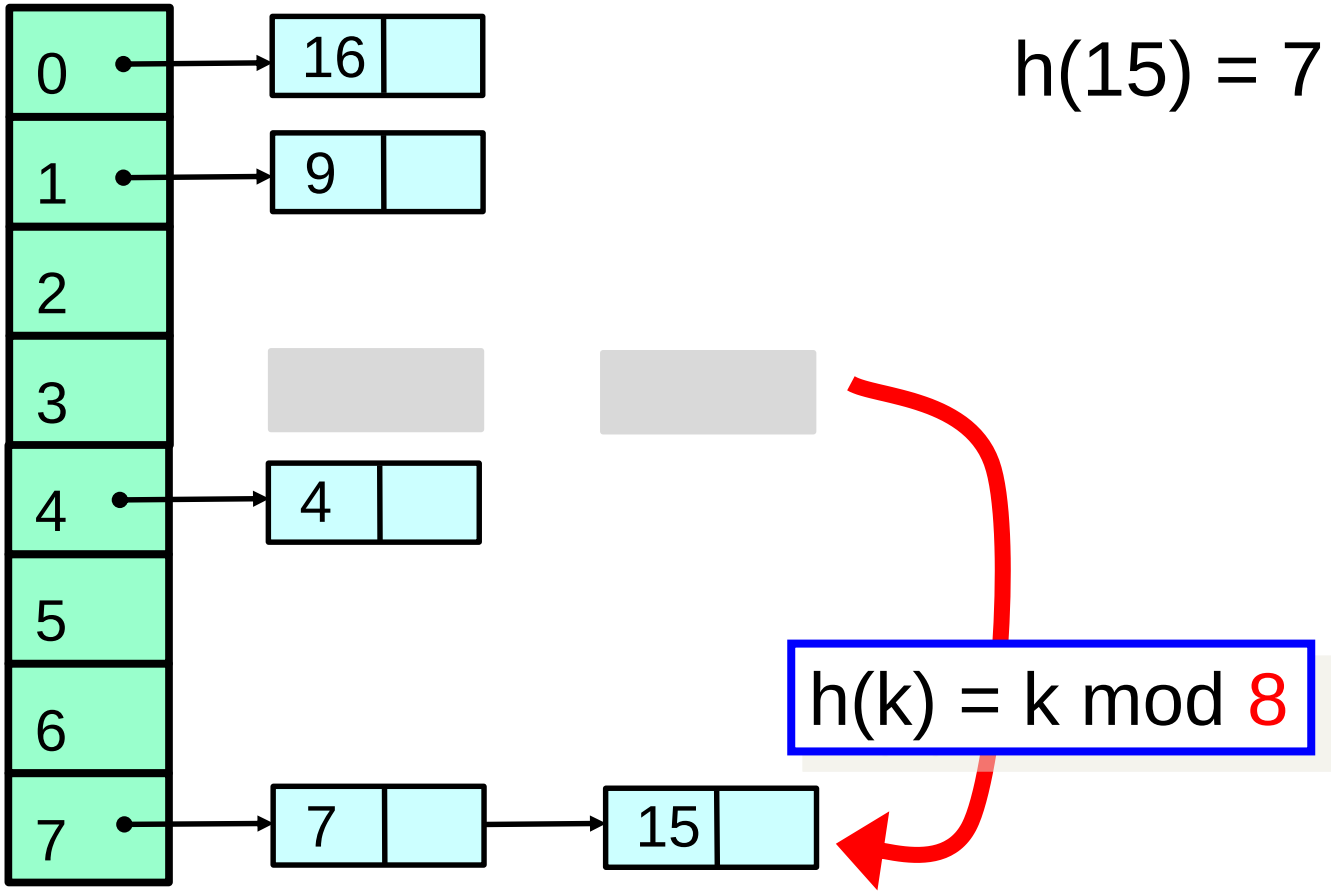
h(k) = k mod 8

# Resizing




h(15) = 7 mod 8

# Resizing



# 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**

# 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() //Read operation
  - **9%** add ()
  - **1%** remove()
- Growing is important
- Shrinking not so much

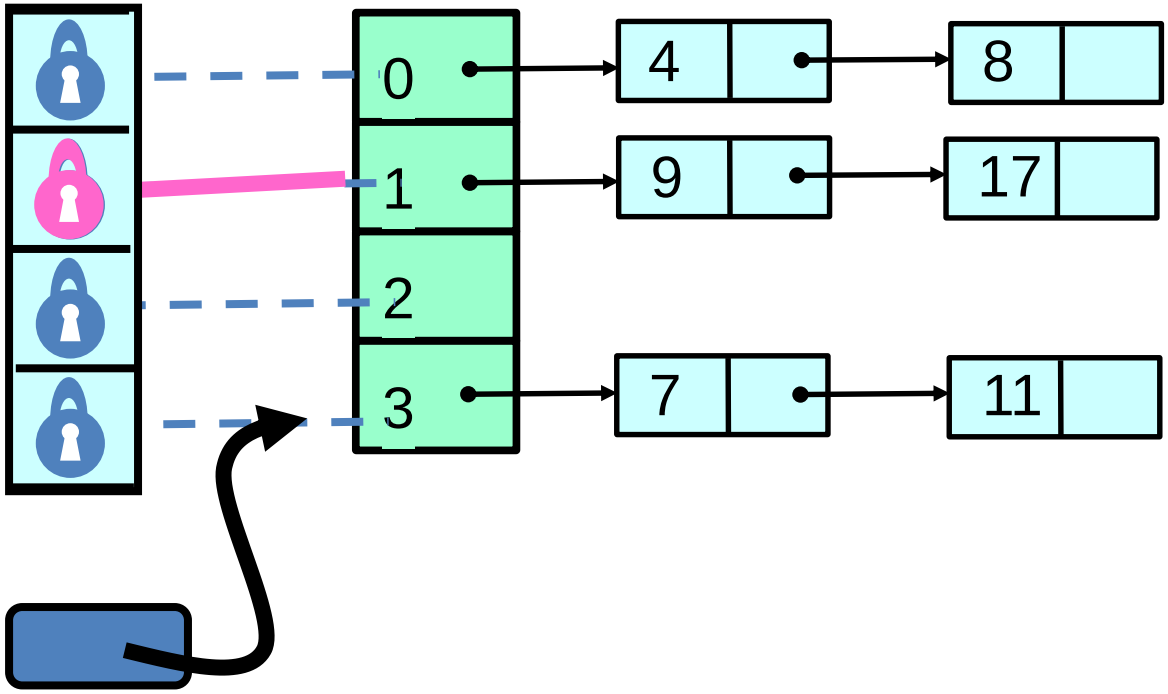
# When to Resize?

- 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

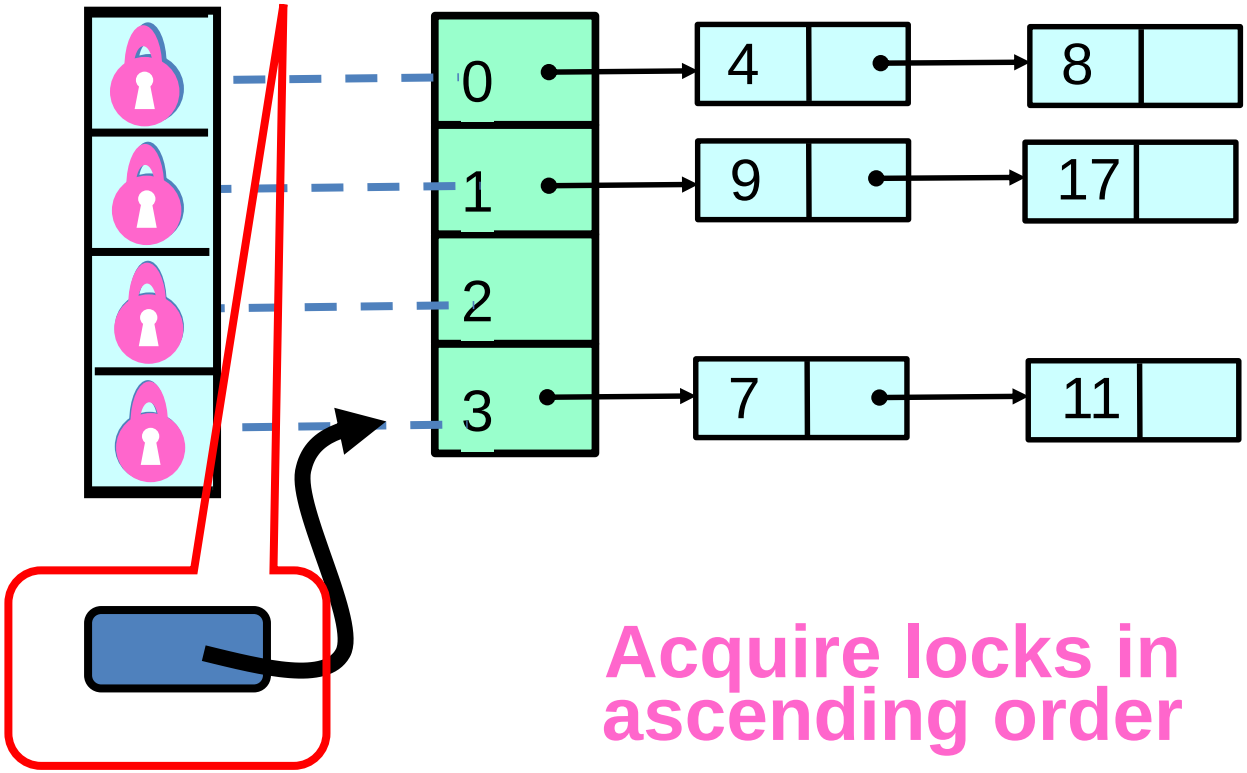
# Fine-grained Locking




**Each lock associated with one bucket**

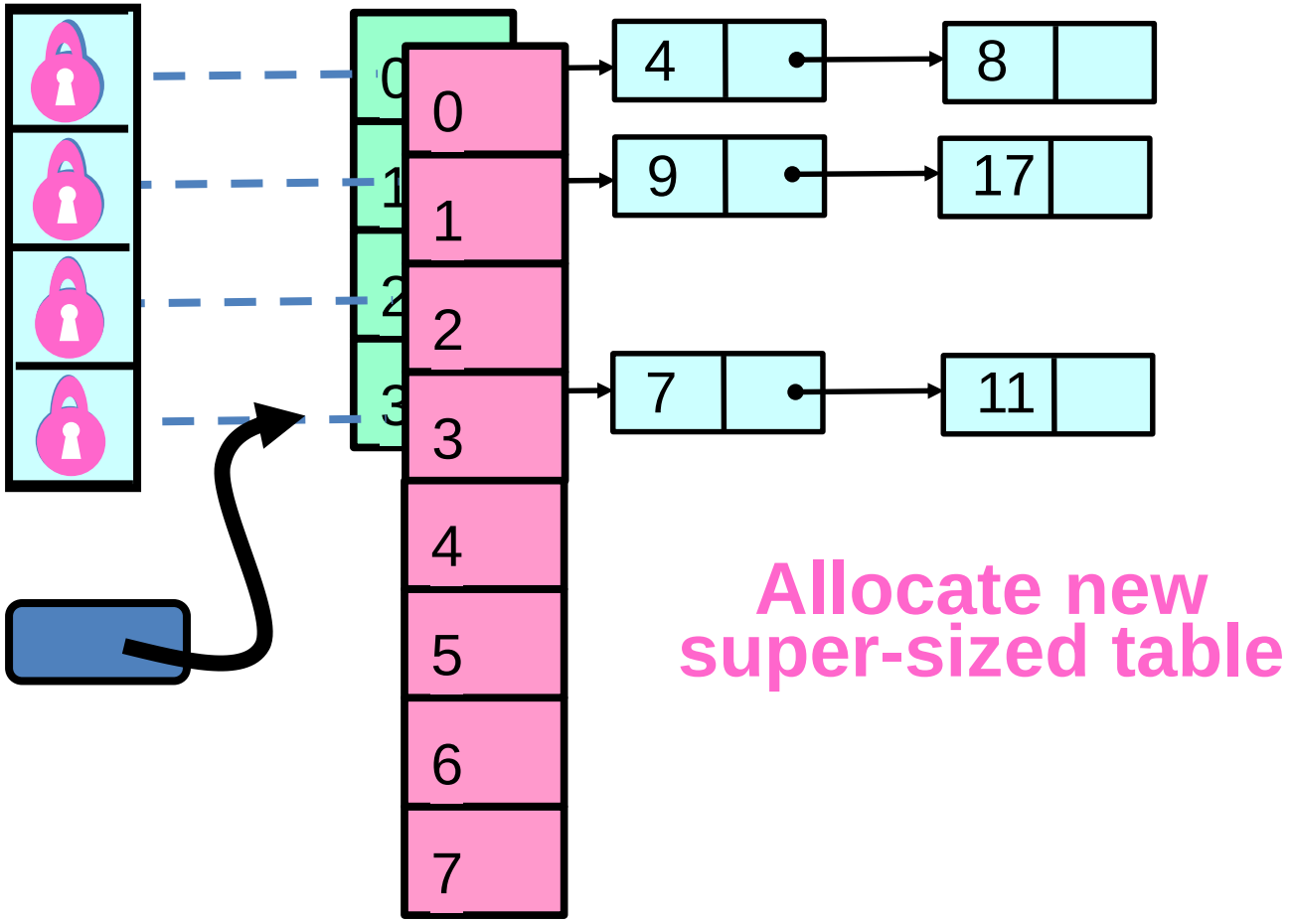
# Resize This

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

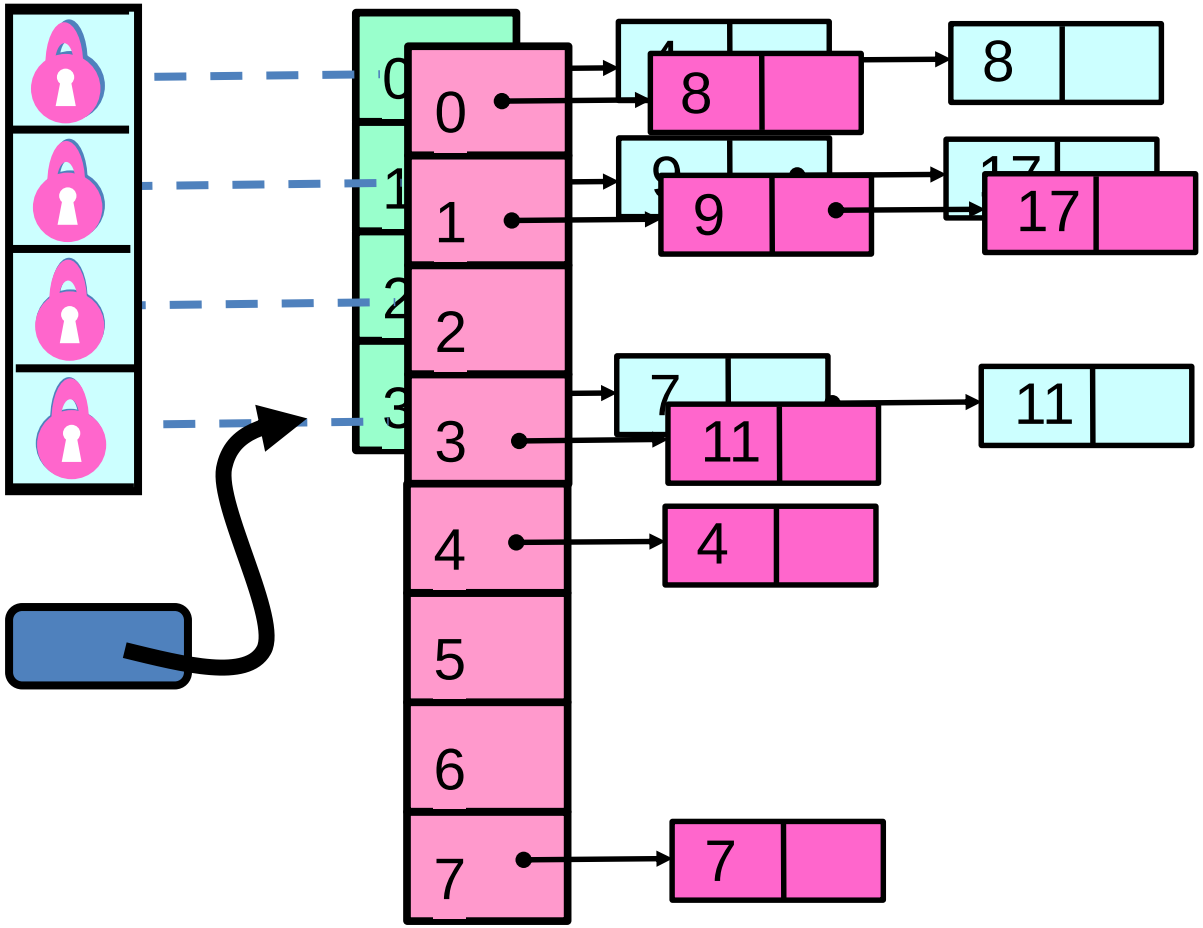



**Acquire locks in ascending order**

# Resize This

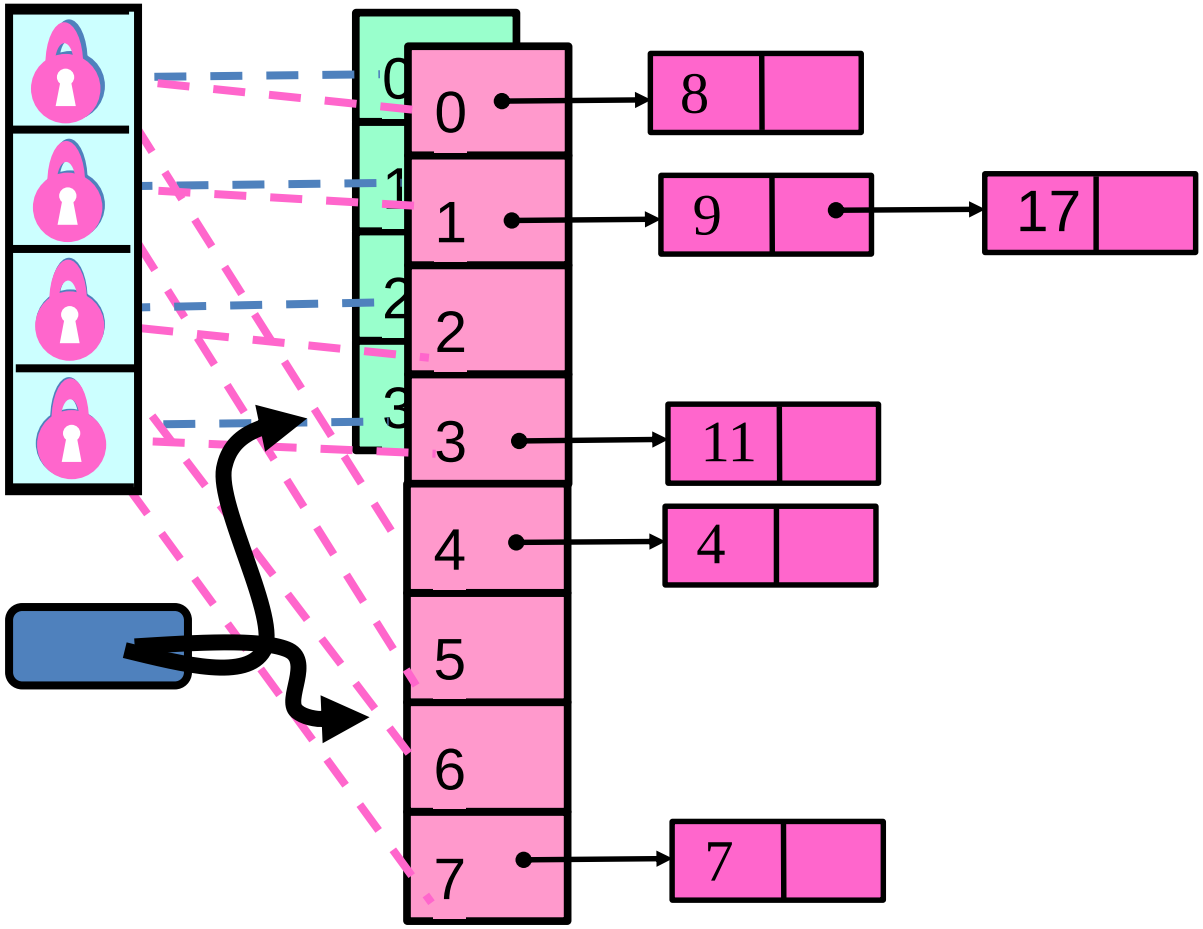


# Resize This



## 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
  - If we're locking anyway, why pay?

# Assignment II

- Design and implement a Java multithreaded program
- To simulate a bank money transaction system for up to $10^6$ users.
- Name of the bank is Guwahati National Bank (GNB)
  - the bank have 10 branches at different location of India and each branch have 10 updaters.
- Initially, $10^4$ users will be there for each branch with a random amount of money in their accounts**.**
- **Every updater of the GNB can be modeled as a separate thread.**

# Assignment II

- The updater gets a request
  - To Cash Deposit, Cash withdrawal (withdrawal amount should less than amount money in the account).
  - To Transfer Money from one customer account to another customer account. The source account and destination account may be in different branches of GNB.
  - to add a customer with some initial money in that account, added customer will be in the **updater branch** of GNB.
  - To delete a customer from the system (or close the account of a user of the GNBs from **any branch**).
  - To transfer customer account from **one branch to another branch** of GNB.

# Assignment II

- Assume the probability of getting cash deposit, cash withdrawal, money transfer, add a customer, delete a customer, transfer a customer to updater are 0.33, 0.33, 0.33, 0.003, 0.003, and 0.004 respectively.
- Simulate up to $10^6$ transactions per updater to test
  - the correctness of your implementation.
  - Report the execution time of your simulation program.

# Assignment II

- Suppose information about all the customer accounts of the GNB
  - is maintained by **an array of linked lists**.
  - Each linked list represent (or hold data of a) branch of GNB, so there are 10 linked list and the array of linked list is maintained by a hash data structure.
  - Every customer account number is represented by 10 digits and the first digit of the customer account number identify the branches of GNB.
- You to be used linked list and hash should be
  - thread-safe, and throughput should be high.
  - You are allowed to use any inbuilt data structure, locking protocol, synchronized functions for the same.

# Thanks