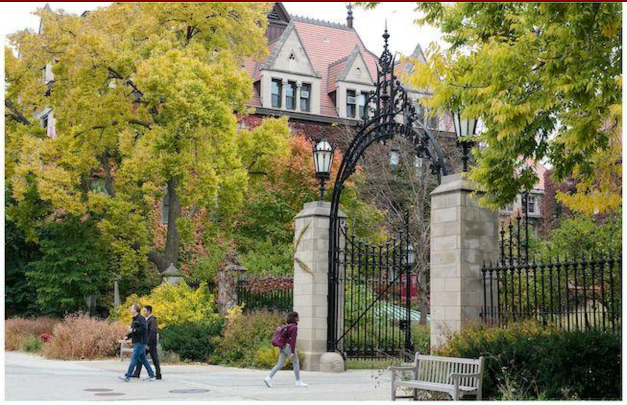
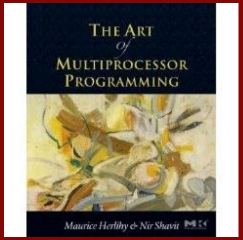
MPCS 52060 - Parallel Programming
M4: Concurrent Data Structures (Part 1)




The Art of Multiprocessor Programming
Maurice Herlihy & Nir Shavit

Original slides from "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit with modifications by Lamont Samuels

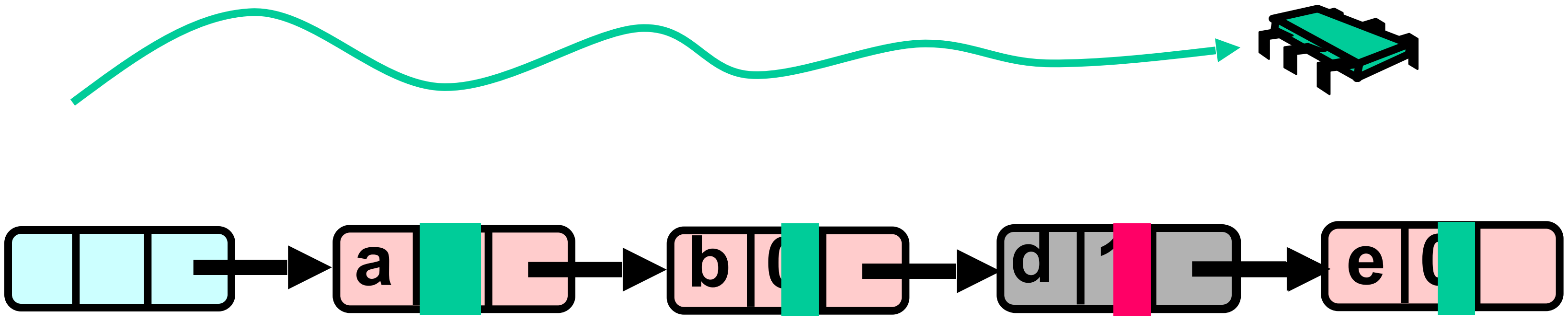
THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

Agenda

- Concurrent Data Structures
  - Lock-free Lists
  - Hash tables (Maps)
  - Queues
  - Stacks

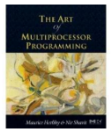
Lock-free Lists

Summary: Wait-free Contains

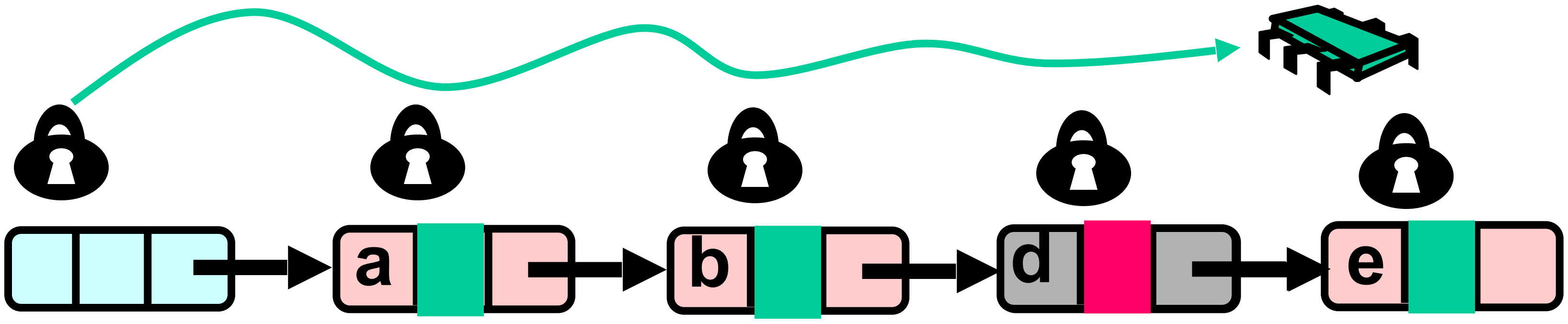


Use Mark bit + list ordering

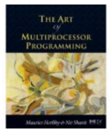
1. Not marked → in the set
2. Marked or missing → not in the set



Lazy List

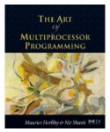


Lazy add() and remove() + Wait-free contains()



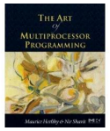
Evaluation

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays



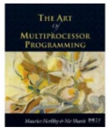
Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….



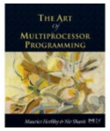
Reminder: Lock-Free Data Structures

- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

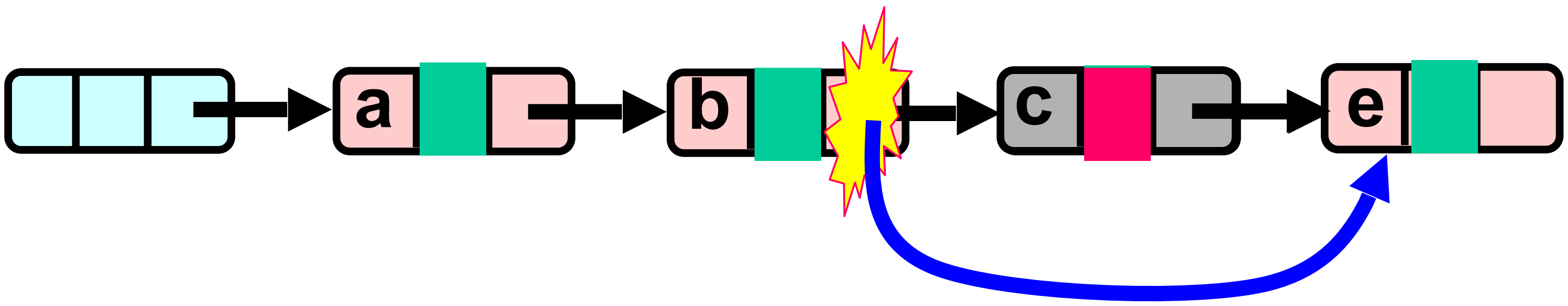


Lock-free Lists

- Next logical step
  - Wait-free contains()
  - lock-free add() and remove()
- Use only compareAndSet()
  - What could go wrong?

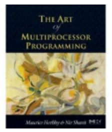


Lock-free Lists

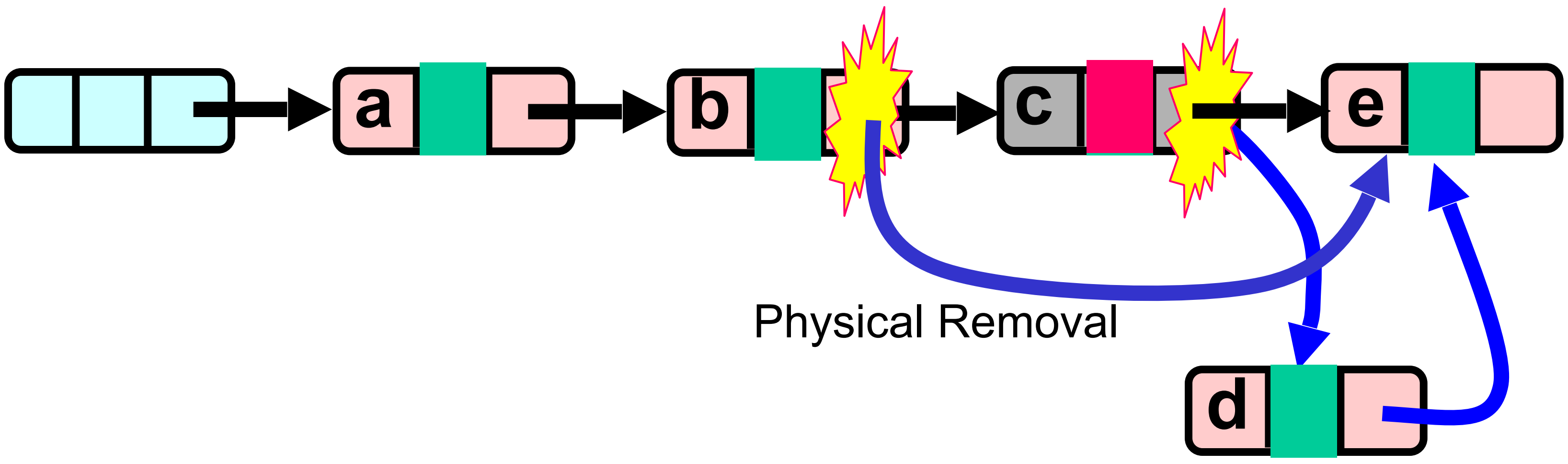


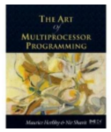
Use CAS to verify pointer is correct

Not enough!



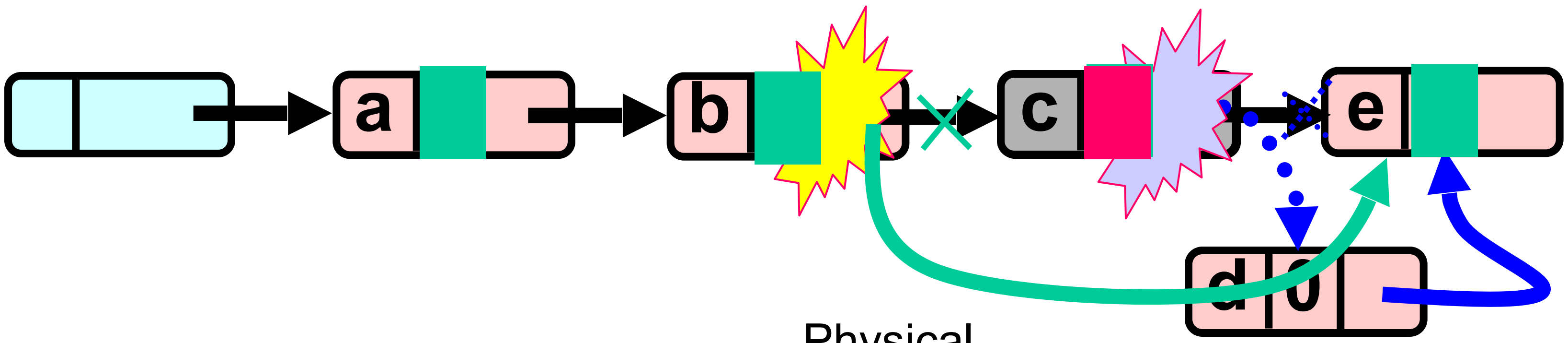
Problem…





The Solution: Combine Bit and Pointer

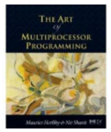
Logical Removal =
Set Mark Bit



Physical
Removal
CAS

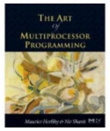
Fail CAS: Node not
added after logical
Removal

Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)



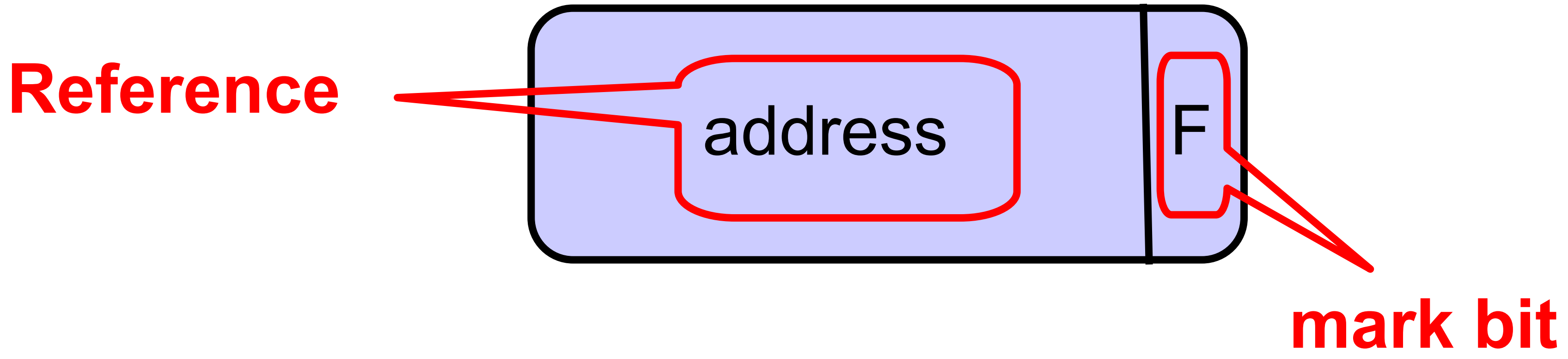
Solution

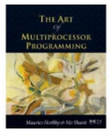
- Use AtomicMarkableReference
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor’s pointer



Marking a Node

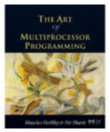
- AtomicMarkableReference class
  - Java.util.concurrent.atomic package





Extracting Reference & Mark

```
Public Object get(boolean[] marked);
```

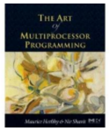


Extracting Reference & Mark

```
Public Object get(boolean[] marked);
```

Returns reference

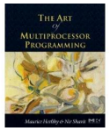
Returns mark at array index 0!



Extracting Mark Only

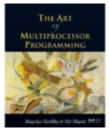
```
public boolean isMarked();
```

Value of mark



Changing State

```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

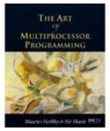


Changing State

If this is the current reference …

```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

And this is the current mark …

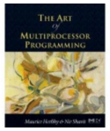


Changing State

…then change to this new reference …

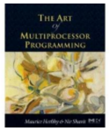
```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

… and this new mark



Changing State

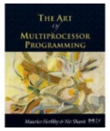
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```



Changing State

```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

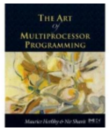
If this is the current reference …



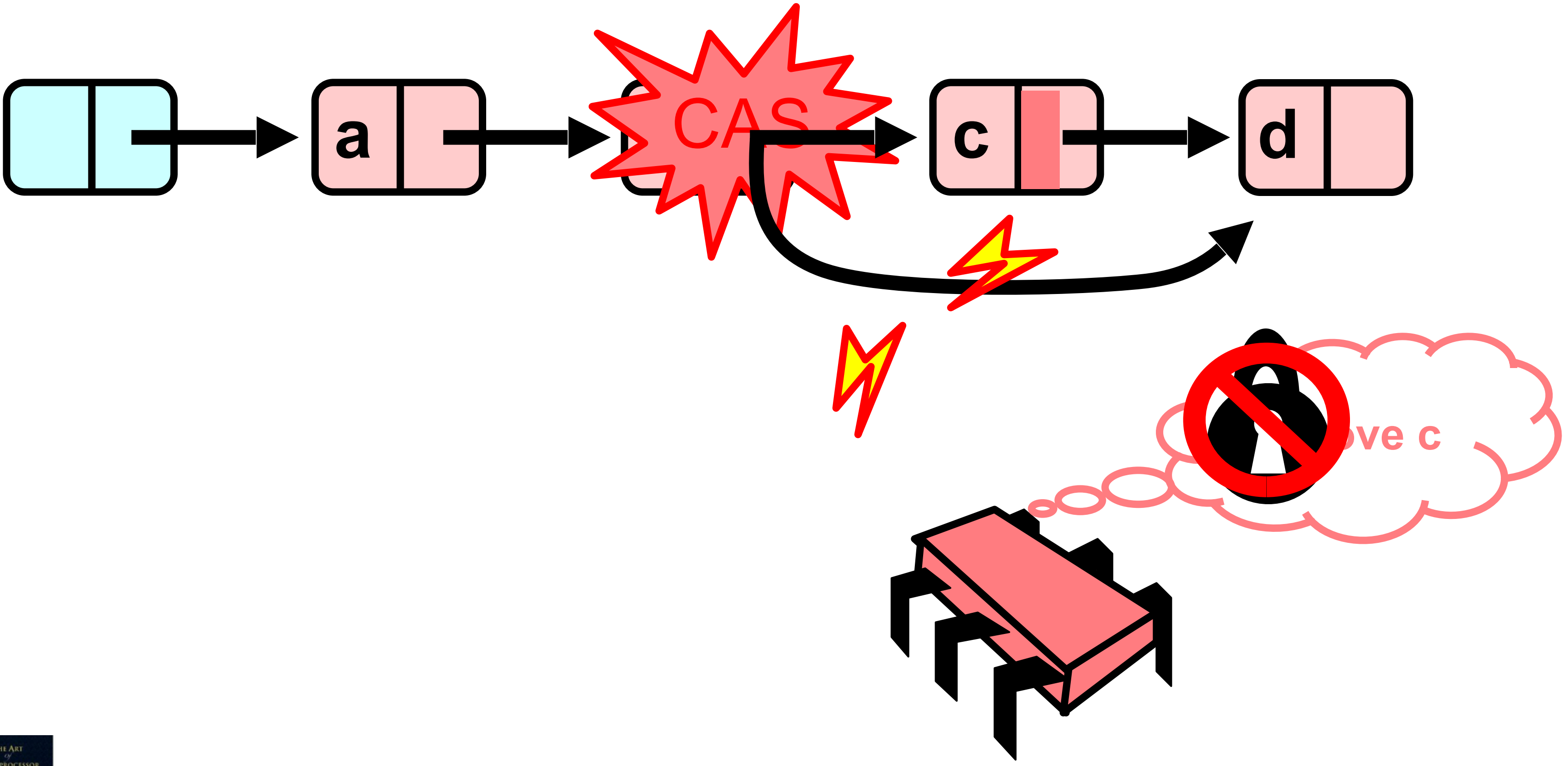
Changing State

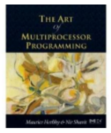
```
public boolean attemptMark(
   Object expectedRef,
   boolean updateMark);
```

.. then change to
this new mark.

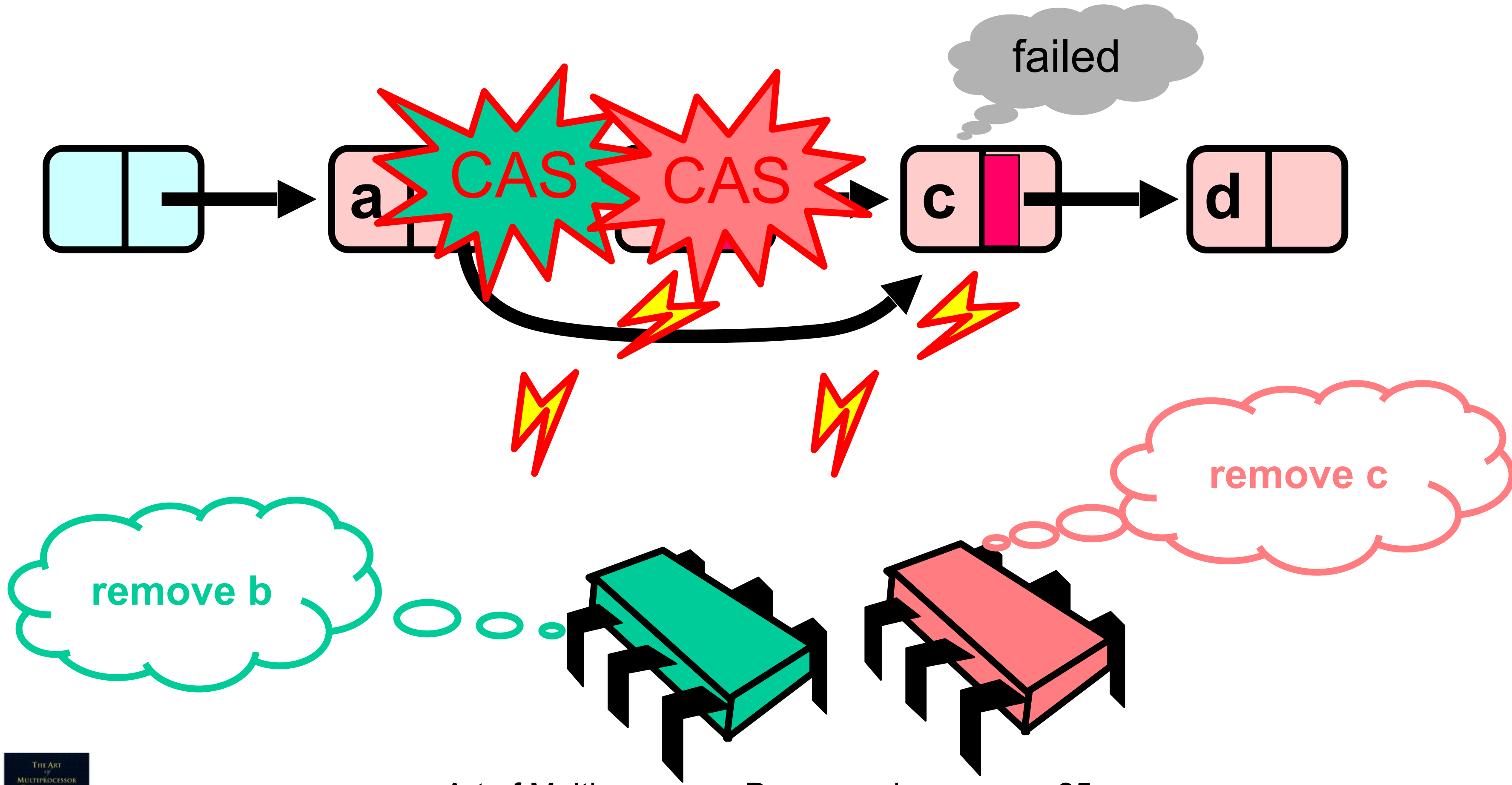


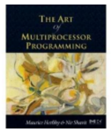
Removing a Node



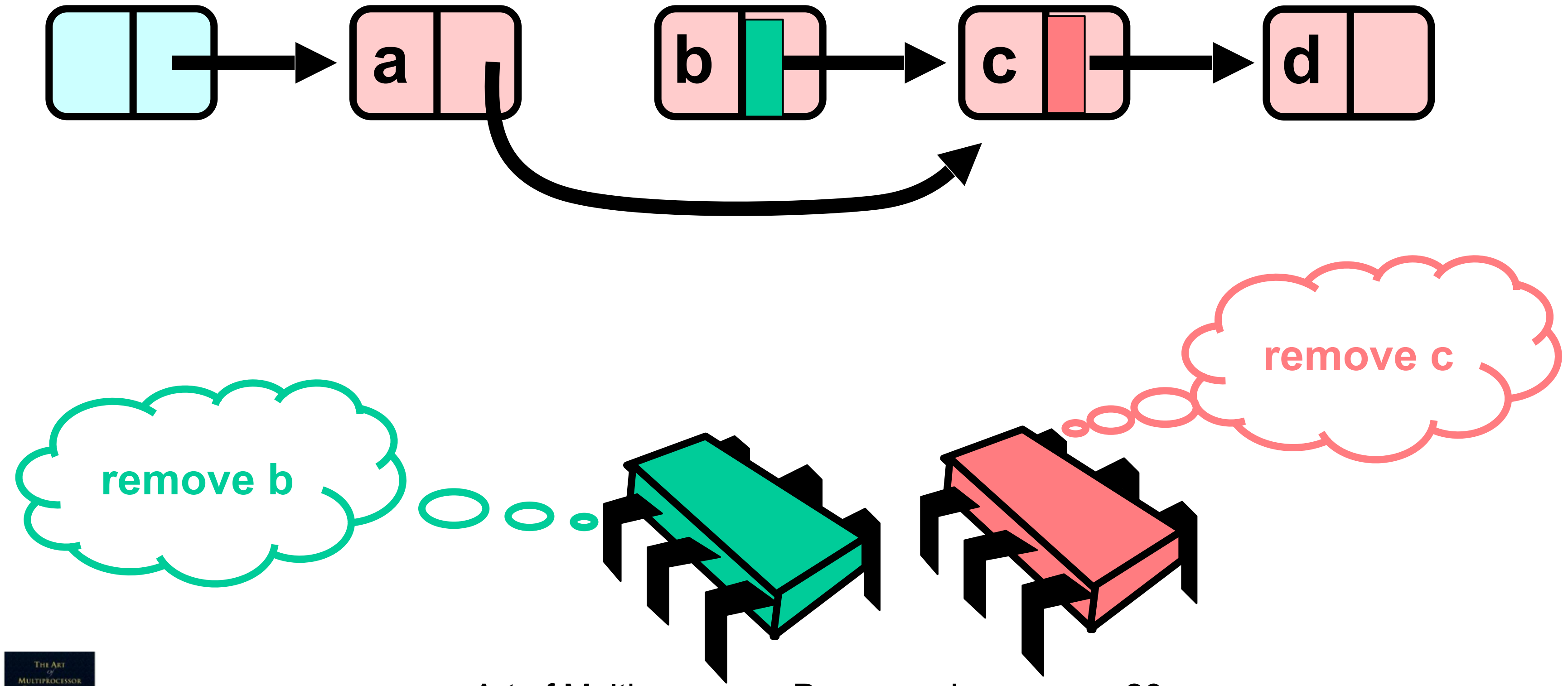


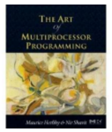
Removing a Node



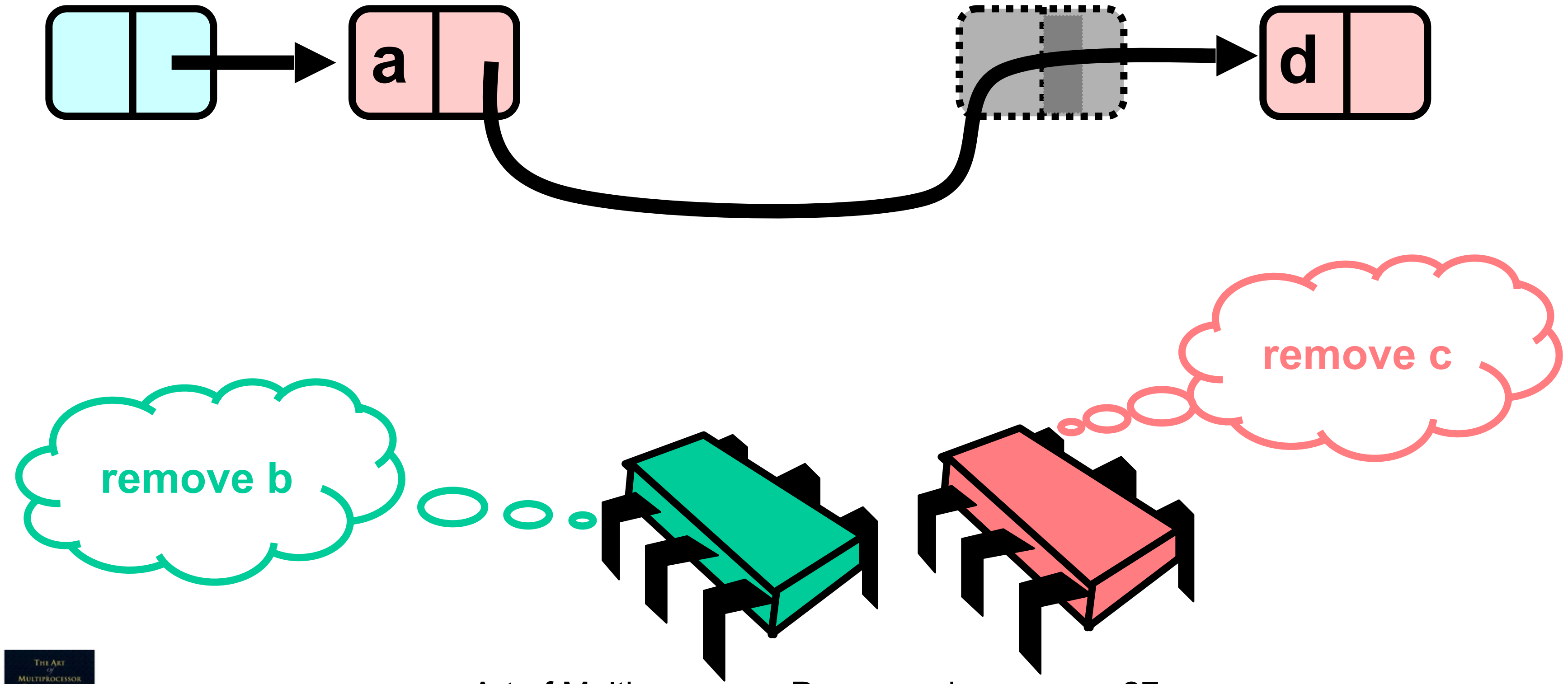


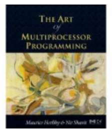
Removing a Node





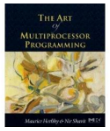
Removing a Node



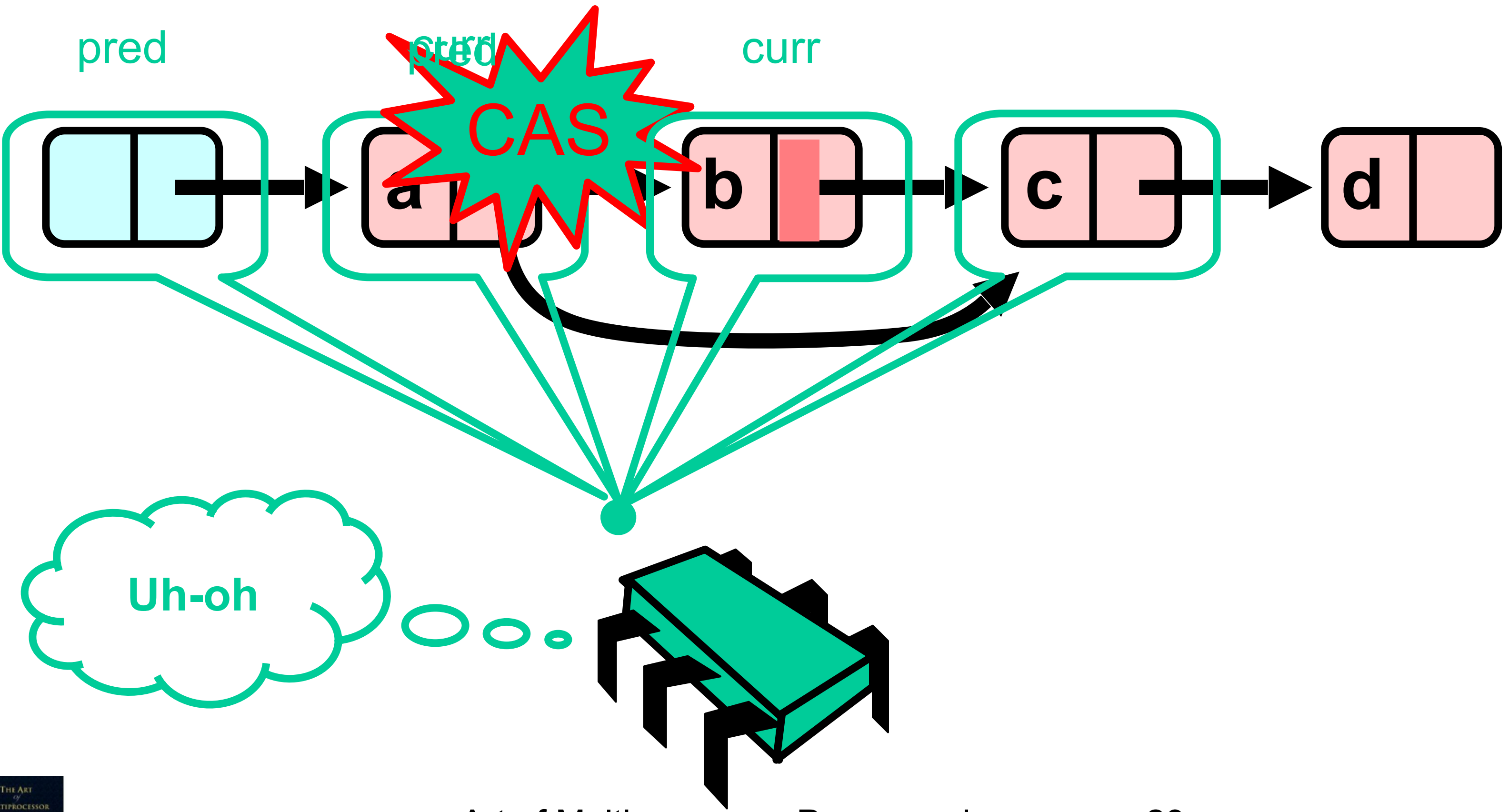


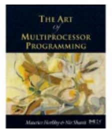
Traversing the List

- Q: what do you do when you find a “logically” deleted node in your path?
- A: finish the job.
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)



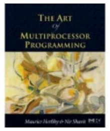
Lock-Free Traversal (only Add and Remove)





The Window Class

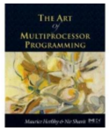
```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```



The Window Class

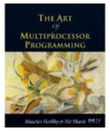
```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```

A container for pred and current values



Using the Find Method

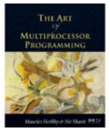
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```



Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

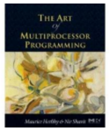
Find returns window



Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

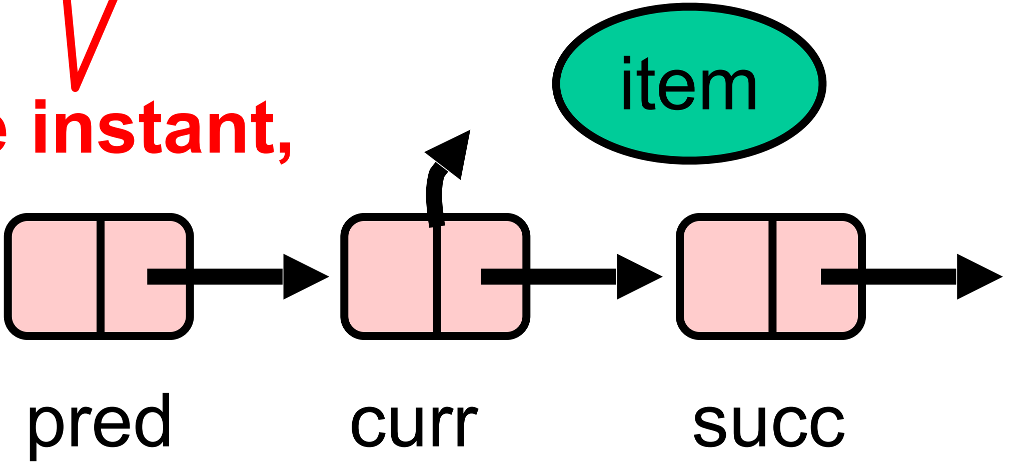
Extract pred and curr



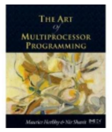
The Find Method

```
Window window = find(item);
```

At some instant,



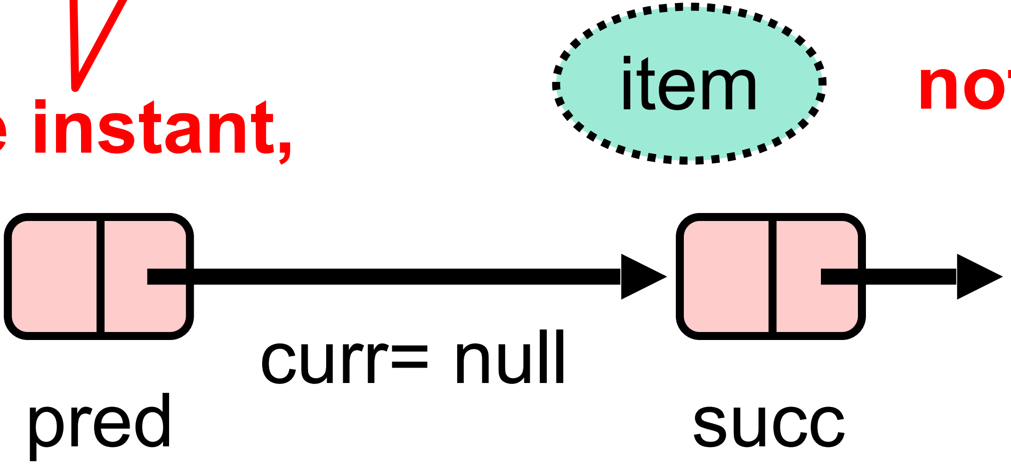
or …

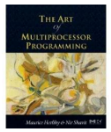


The Find Method

```
Window window = find(item);
```

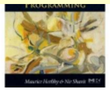
At some instant,





Remove

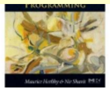
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet (succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```

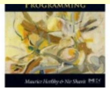
Keep trying



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
   if (curr.key != key) {
      return false;
   } else {
   Node succ = curr.next.getReference();
   snip = curr.next.compareAndSet (succ, succ, false,
true);
   if (!snip) continue;
    pred.next.compareAndSet(curr, succ, false, false);
      return true;
}}}
```

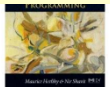
Find neighbors



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```

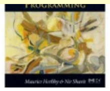
She's not there …



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```

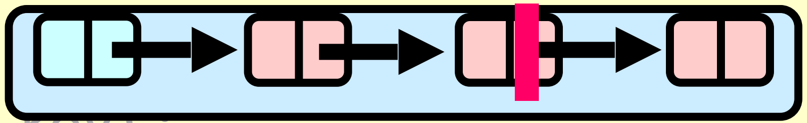
Try to mark node as deleted



Remove

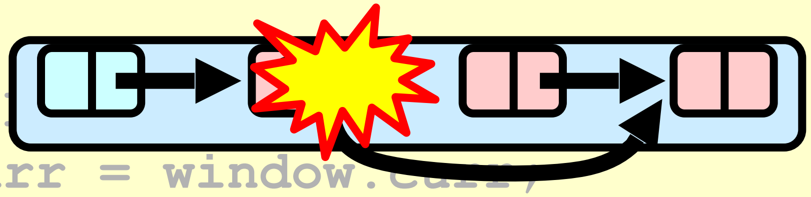
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
 if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```

If it doesn’t work, just retry, if it does, job essentially done

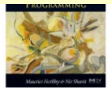


Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head,
 Node pred = window.pred, curr = window.curr);
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}
```

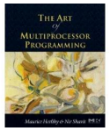


Try to advance reference
(if we don't succeed, someone else did or will).



Add

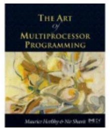
```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```



Add

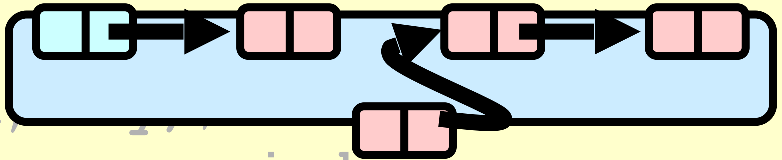
```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Item already there.

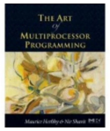


Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```



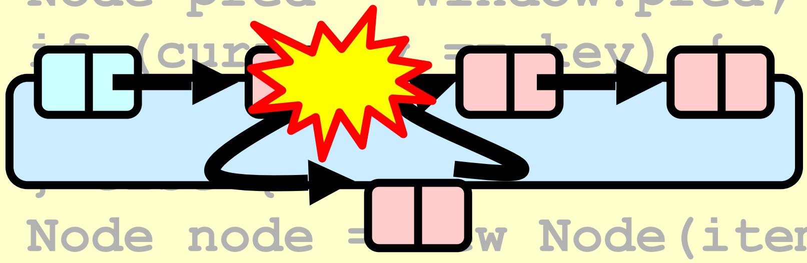
create new node

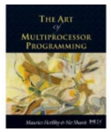


Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

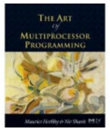
Install new node, else retry loop





Wait-free Contains

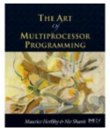
```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```



Wait-free Contains

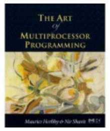
```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Only diff is that we get and check marked



Lock-free Find

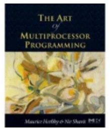
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
         return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

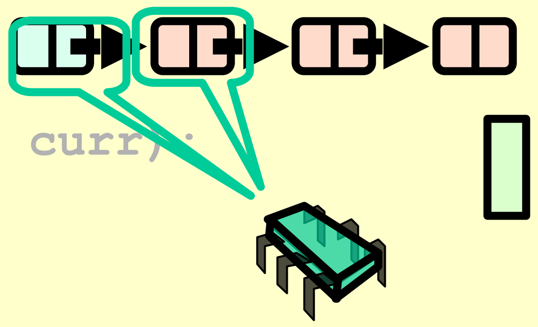
If list changes while traversed, start over



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null,
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

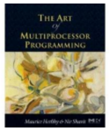
Start looking from head



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

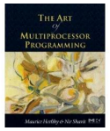
Move down the list



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

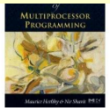
Get ref to successor and current deleted bit



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
```

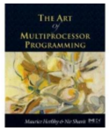
Try to remove deleted nodes in path…code details soon



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

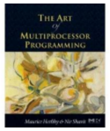
If curr key that is greater or equal, return pred and curr



Lock-free Find

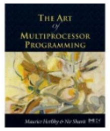
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

Otherwise advance window and loop again



Lock-free Find

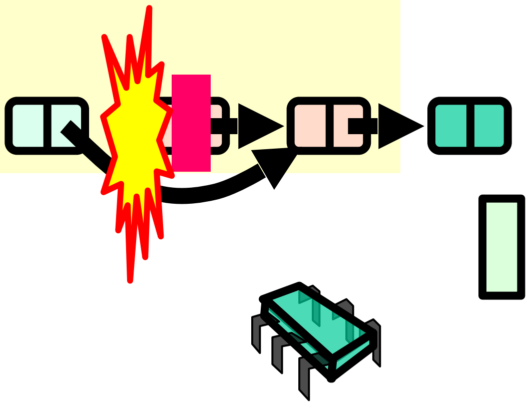
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                               succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

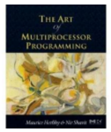


Lock-free Find

Try to snip out node

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                    succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

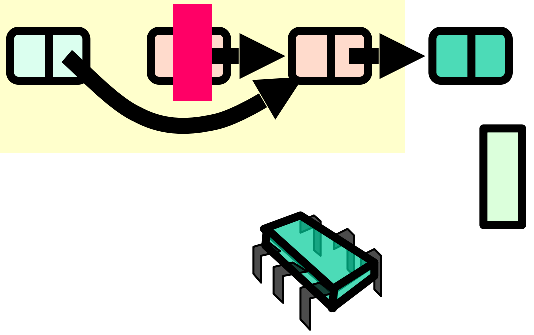


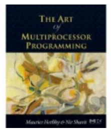


Lock-free Find

if predecessor's next field changed, retry whole traversal

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                    succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

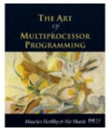




Lock-free Find

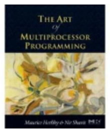
Otherwise move on to check if next node deleted

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```



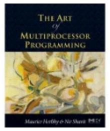
Summary

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization



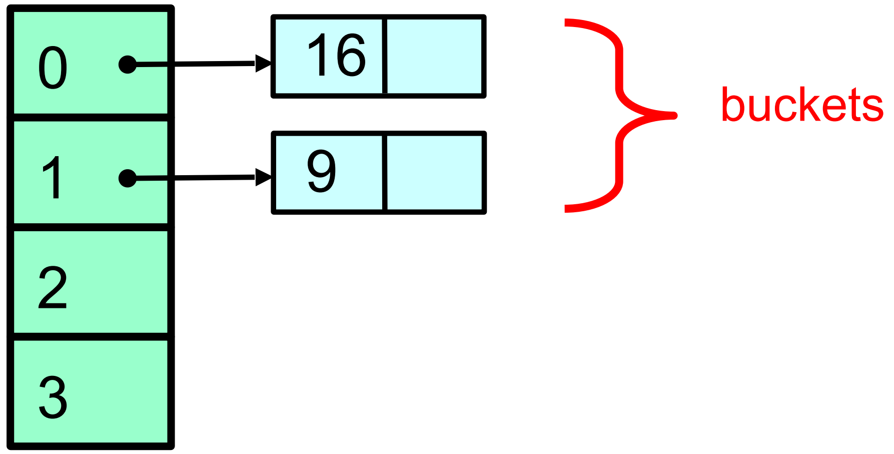
"To Lock or Not to Lock"

- Locking vs. Non-blocking:
  - Extremist views on both sides
- The answer: nobler to compromise
  - Example: Lazy list combines blocking **`add()`** and **`remove()`** and a wait-free **`contains()`**
  - Remember: Blocking/non-blocking is a property of a method



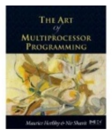
Concurrent Hash tables

Sequential Closed Hash Map

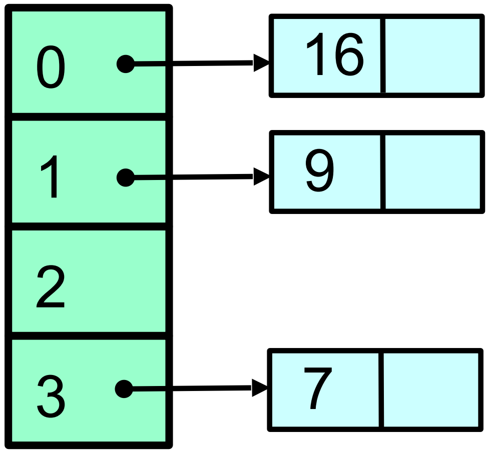


2 Items

h(k) = k mod 4

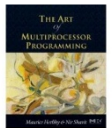


Add an Item

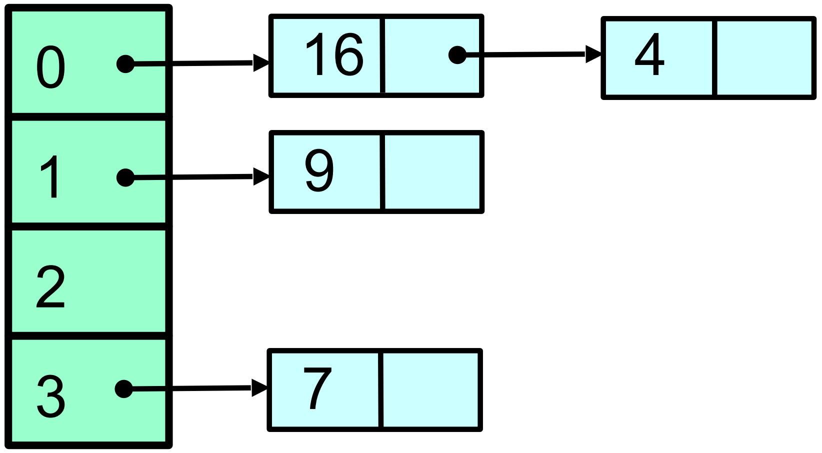


3 Items

h(k) = k mod 4

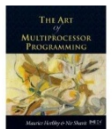


Add Another: Collision

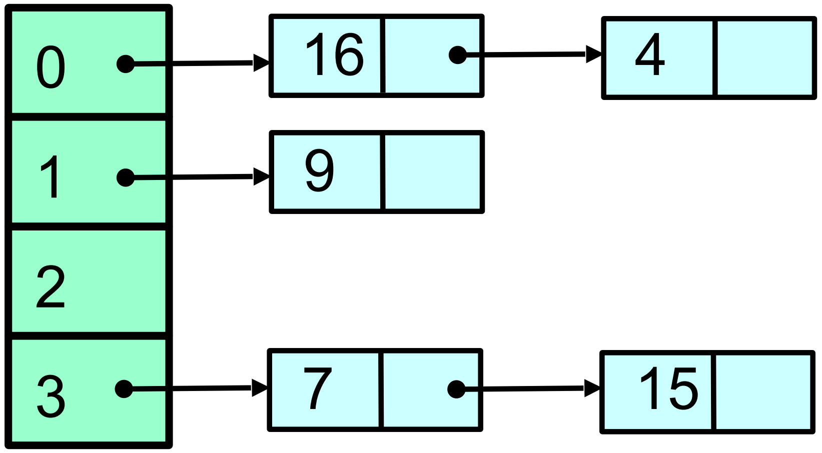


4 Items

h(k) = k mod 4

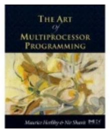


More Collisions

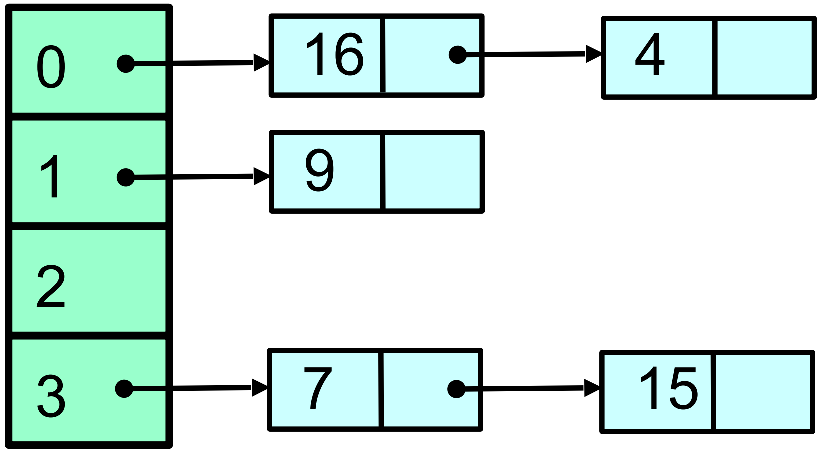


5 Items

h(k) = k mod 4



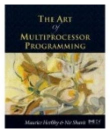
More Collisions



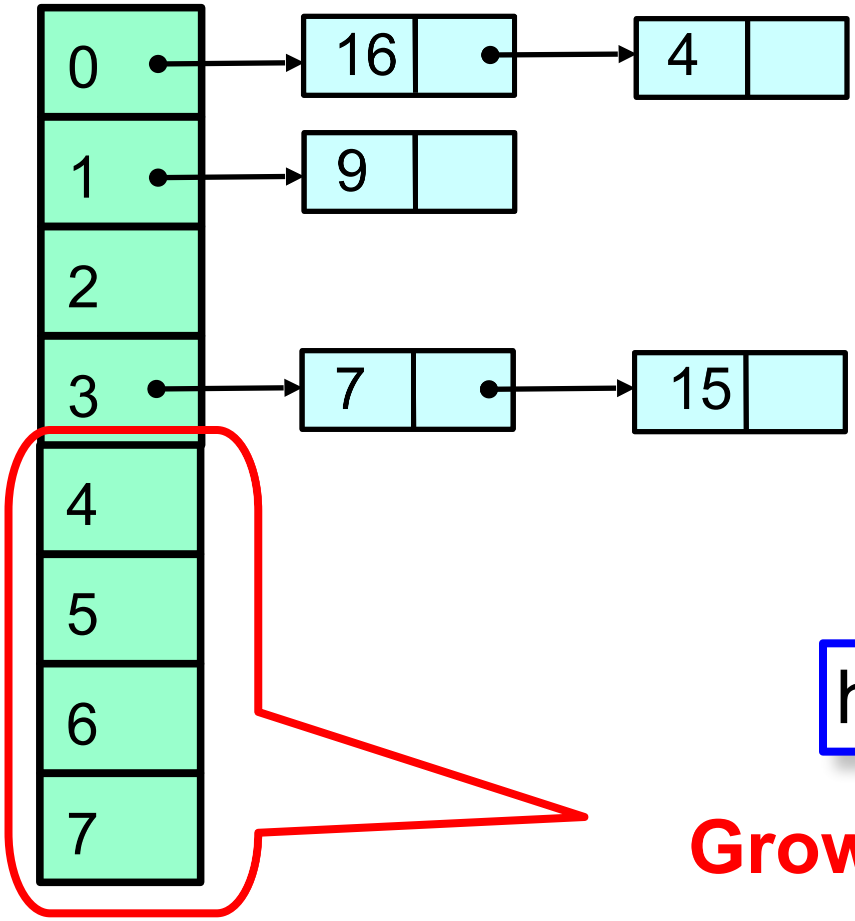
5 Items

h(k) = k mod 4

Problem:
buckets getting too long



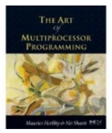
Resizing



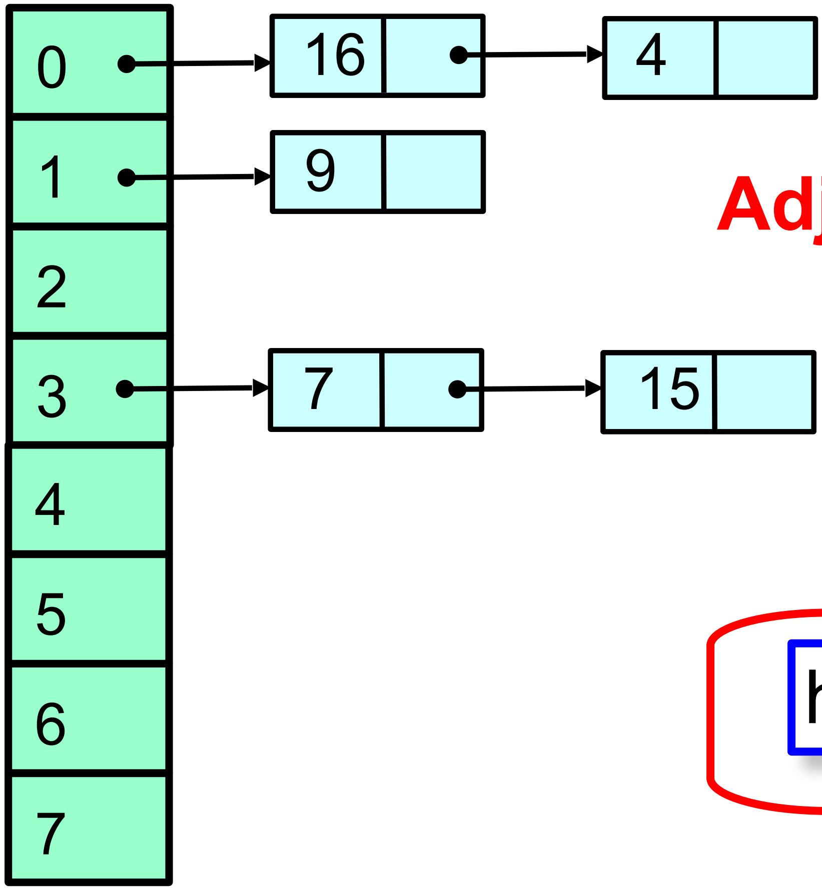
5 Items

h(k) = k mod 8

Grow the array



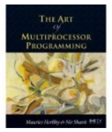
Resizing



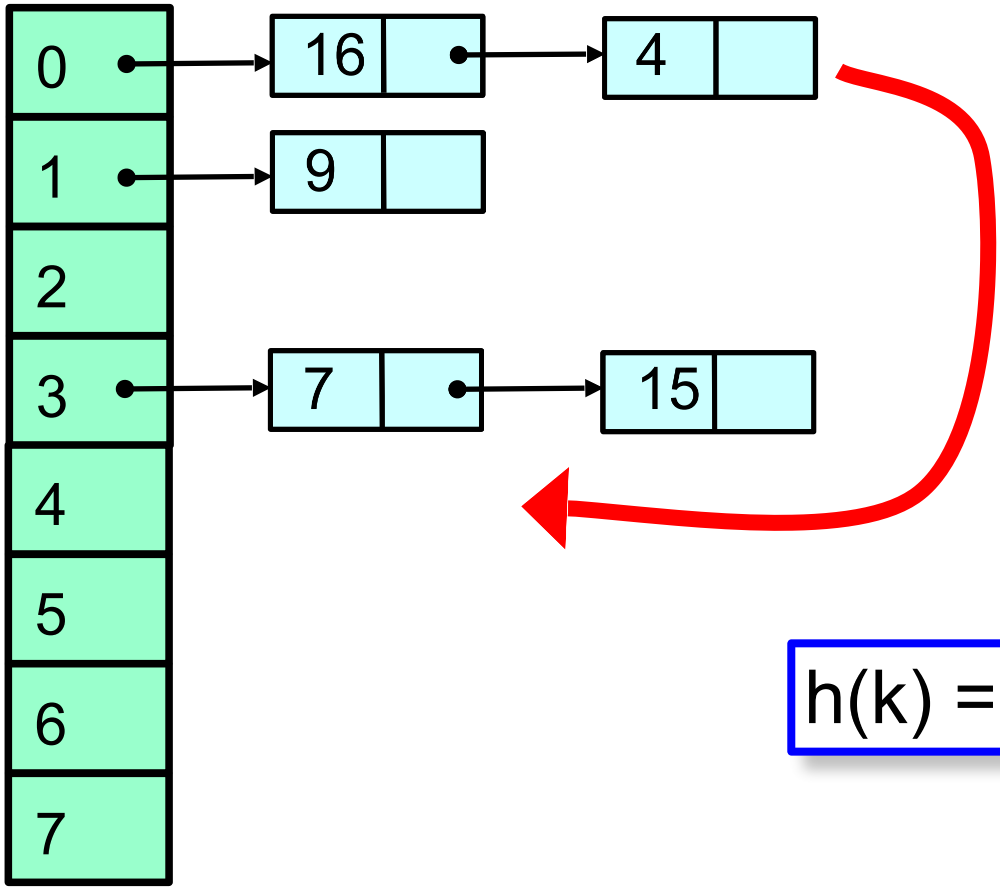
Adjust hash function

5 Items

$h(k) = k \bmod 8$

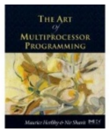


Resizing

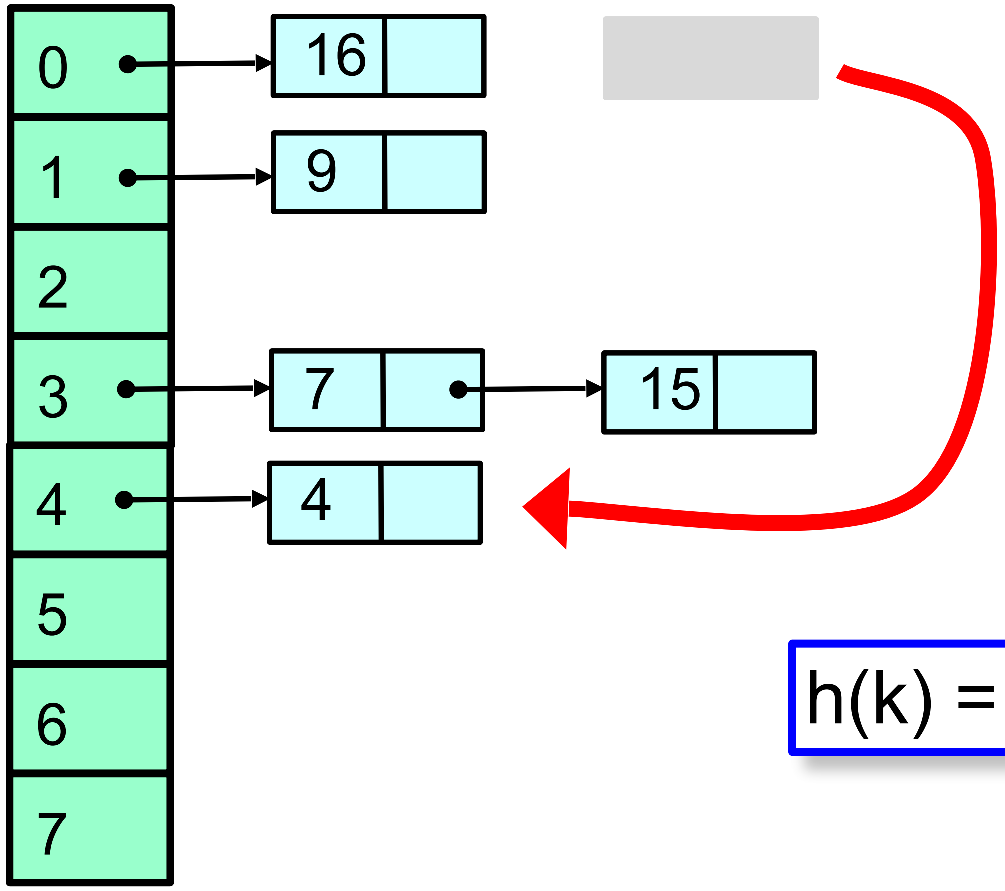


h(4) = 0 mod 8

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

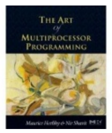


Resizing

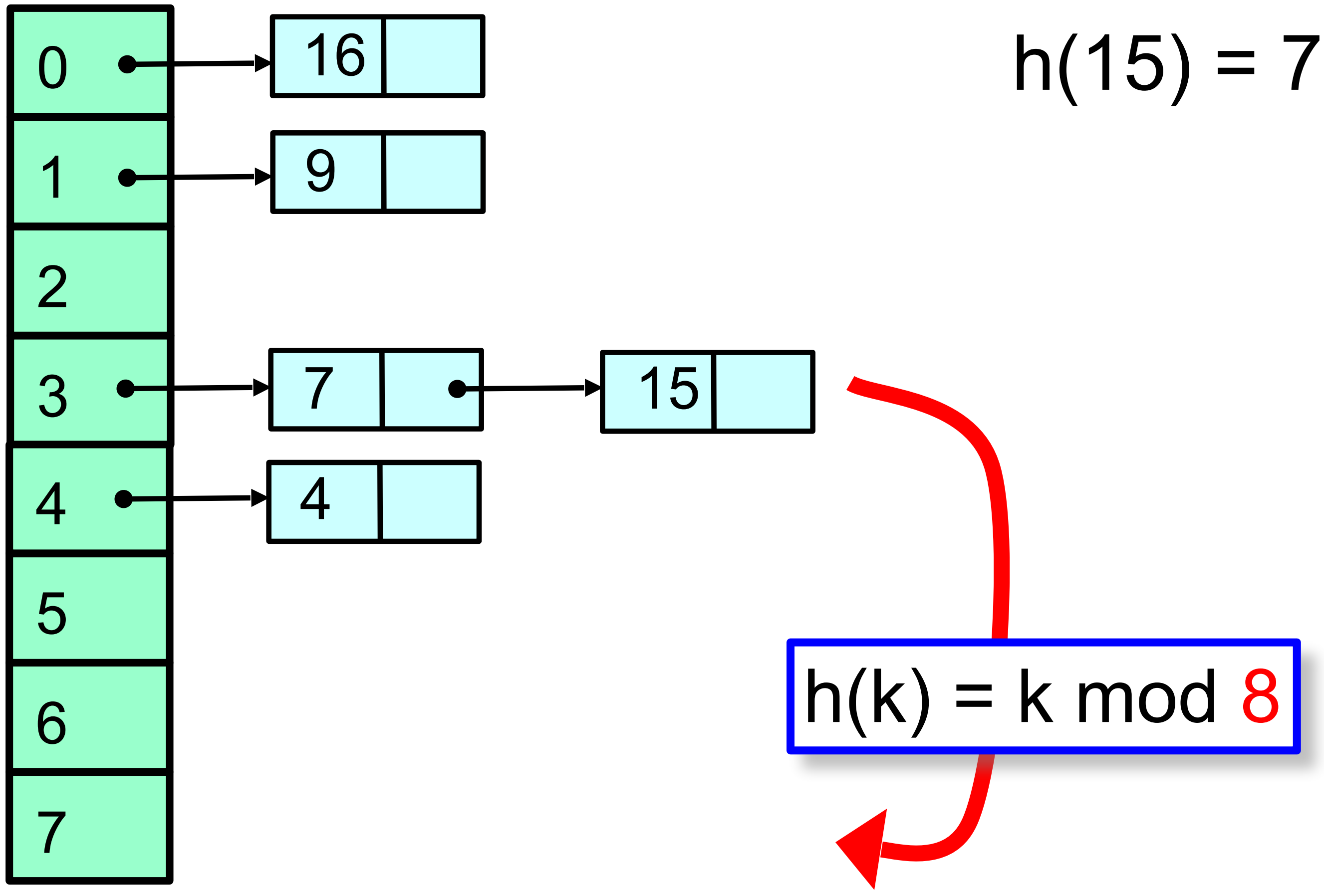


h(4) = 4 mod 8

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

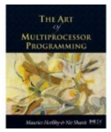


Resizing



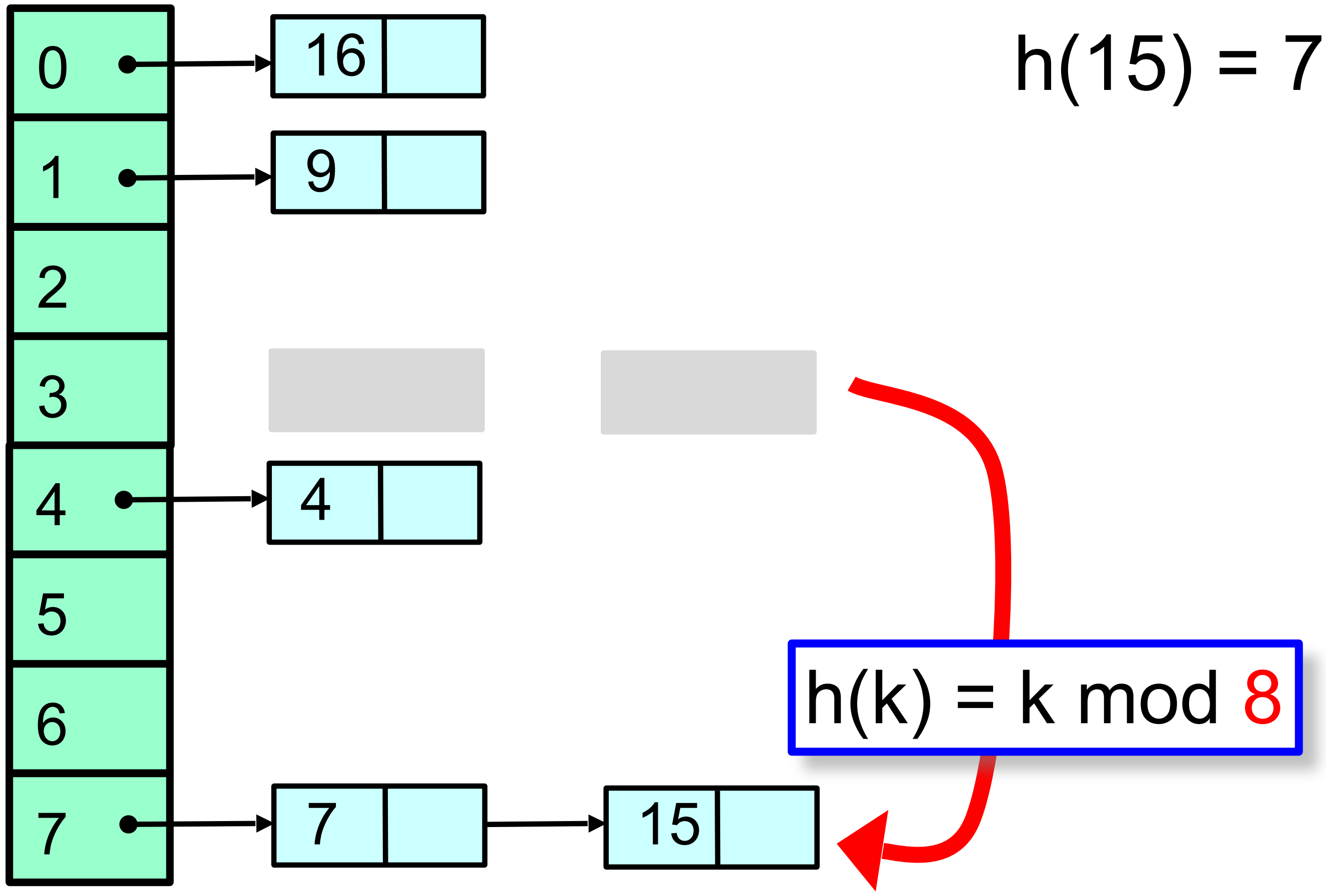
h(15) = 7 mod 8

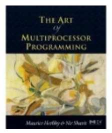
h(k) = k mod 8



Resizing

h(15) = 7 mod 8

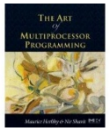




Fields

```
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();
  }
…
```

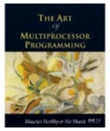
Array of lock-free lists



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();
  }
…
```

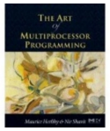
Initial size



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();
  }
…
```

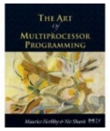
Allocate memory



Constructor

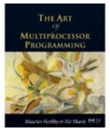
```
public class SimpleHashSet {
  protected LockFreeList[] table;

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

Initialization



Add Method

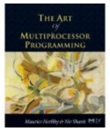
```
public boolean add(Object key) {
 int hash =
  key.hashCode() % table.length;
 return table[hash].add(key);
}
```



Add Method

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

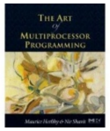
Use object hash code to pick a bucket



Add Method

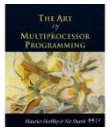
```
public boolean add(Object key) {
 int hash =
  key.hashCode() % table.length;
 return table[hash].add(key);
}
```

Call bucket's add() method



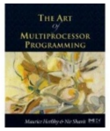
No Brainer?

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



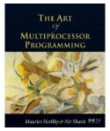
No Brainer?

- We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation
- What’s not to like?
- We don’t know how to resize …



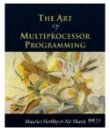
Is Resizing Necessary?

- Constant-time method calls require
  - Constant-length buckets
  - Table size proportional to set size
  - As set grows, must be able to resize



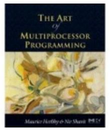
Set Method Mix

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



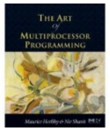
When to Resize?

- 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

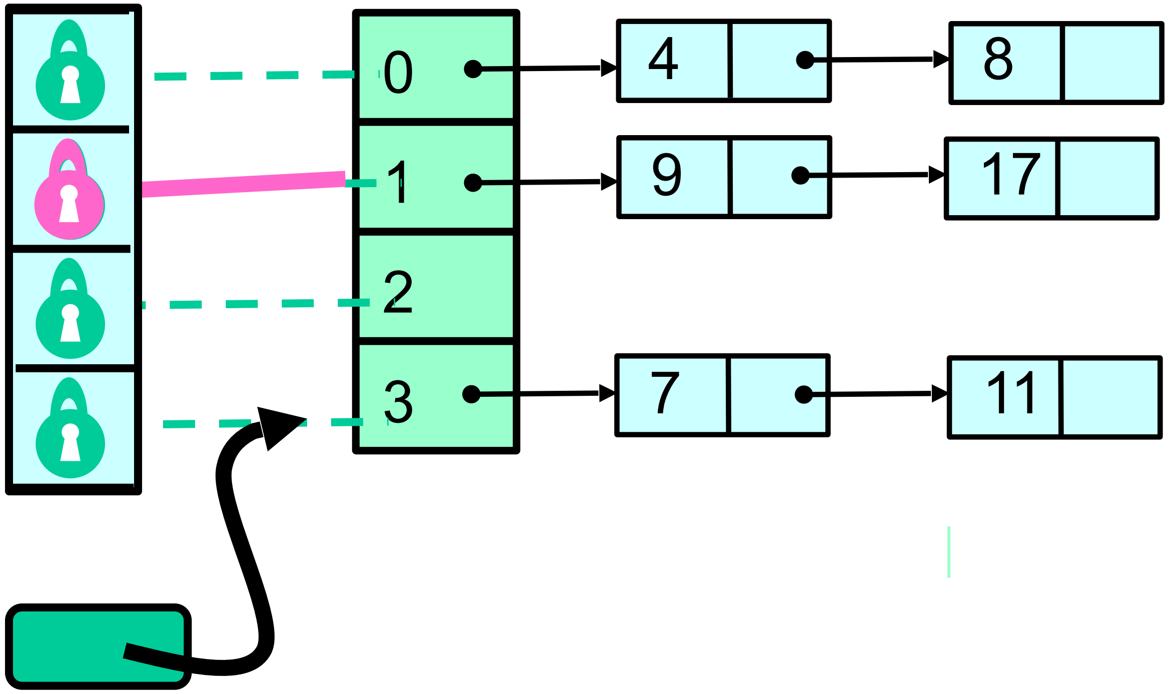


Coarse-Grained Locking

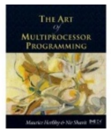
- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck



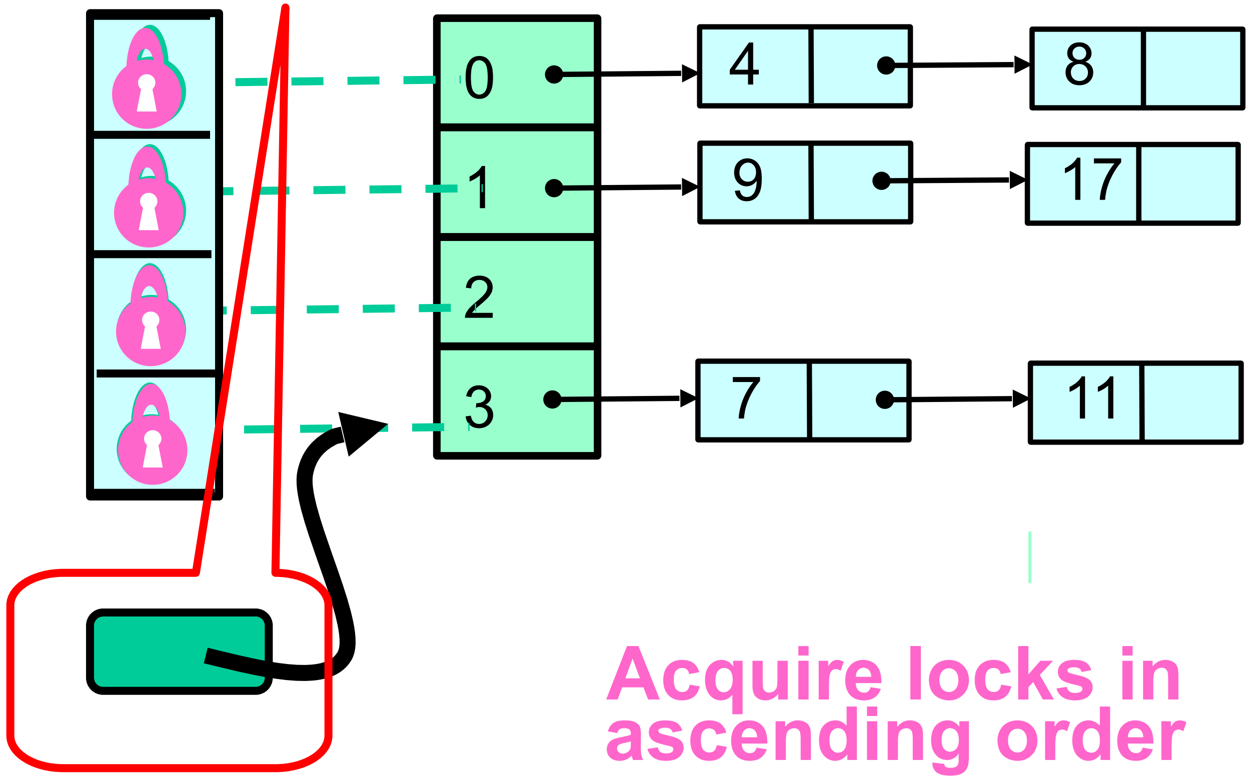
Fine-grained Locking



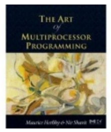
Each lock associated with one bucket



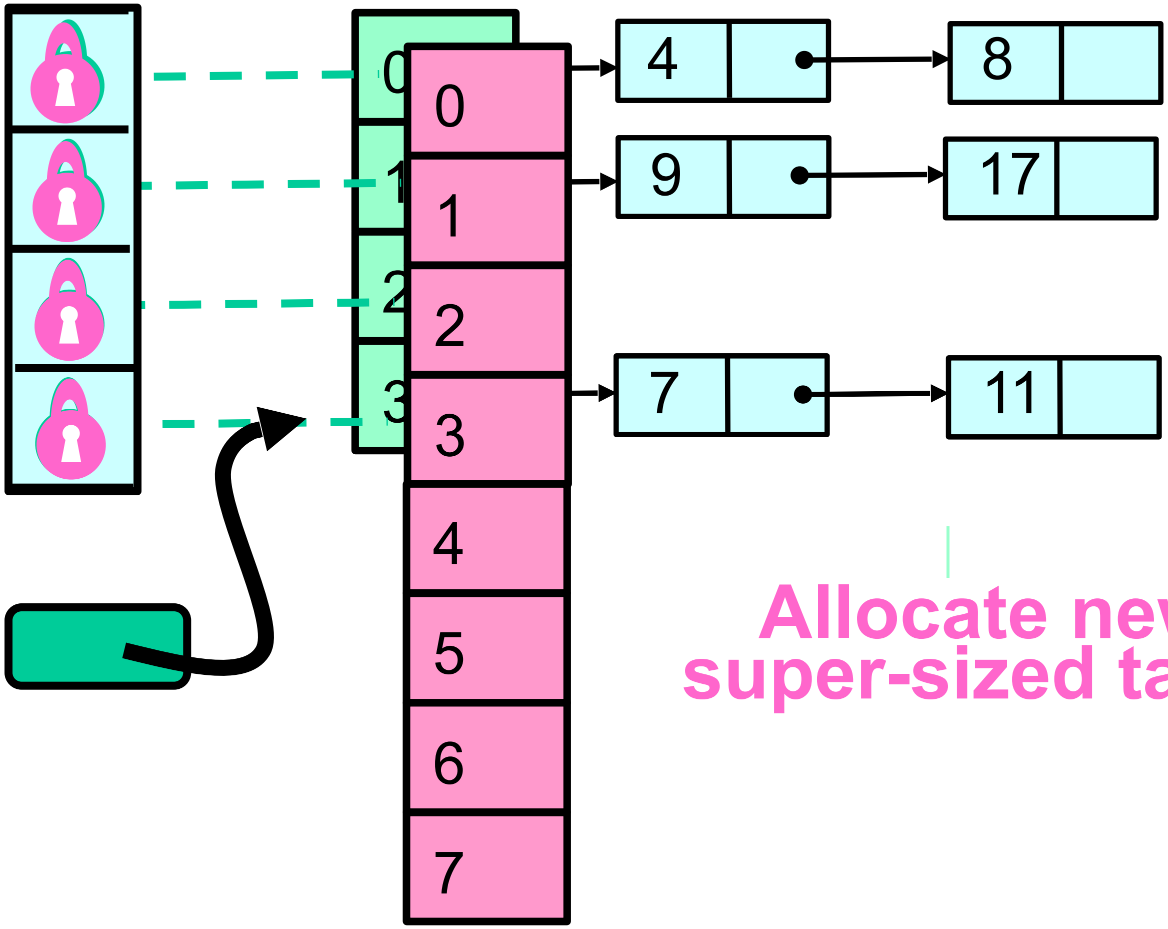
Make sure table reference didn't change between resize decision and lock acquisition



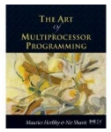
Acquire locks in ascending order



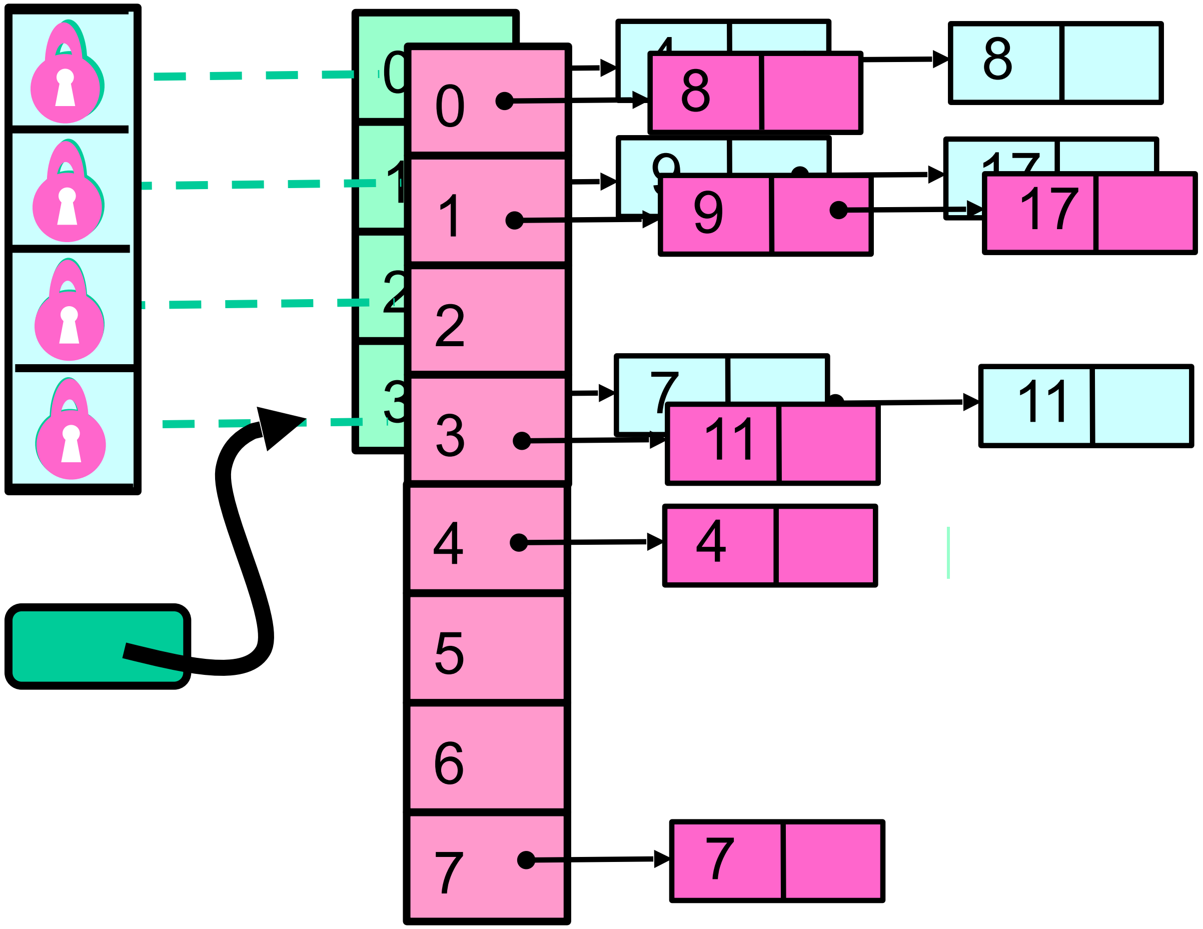
Resize This

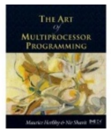


Allocate new super-sized table



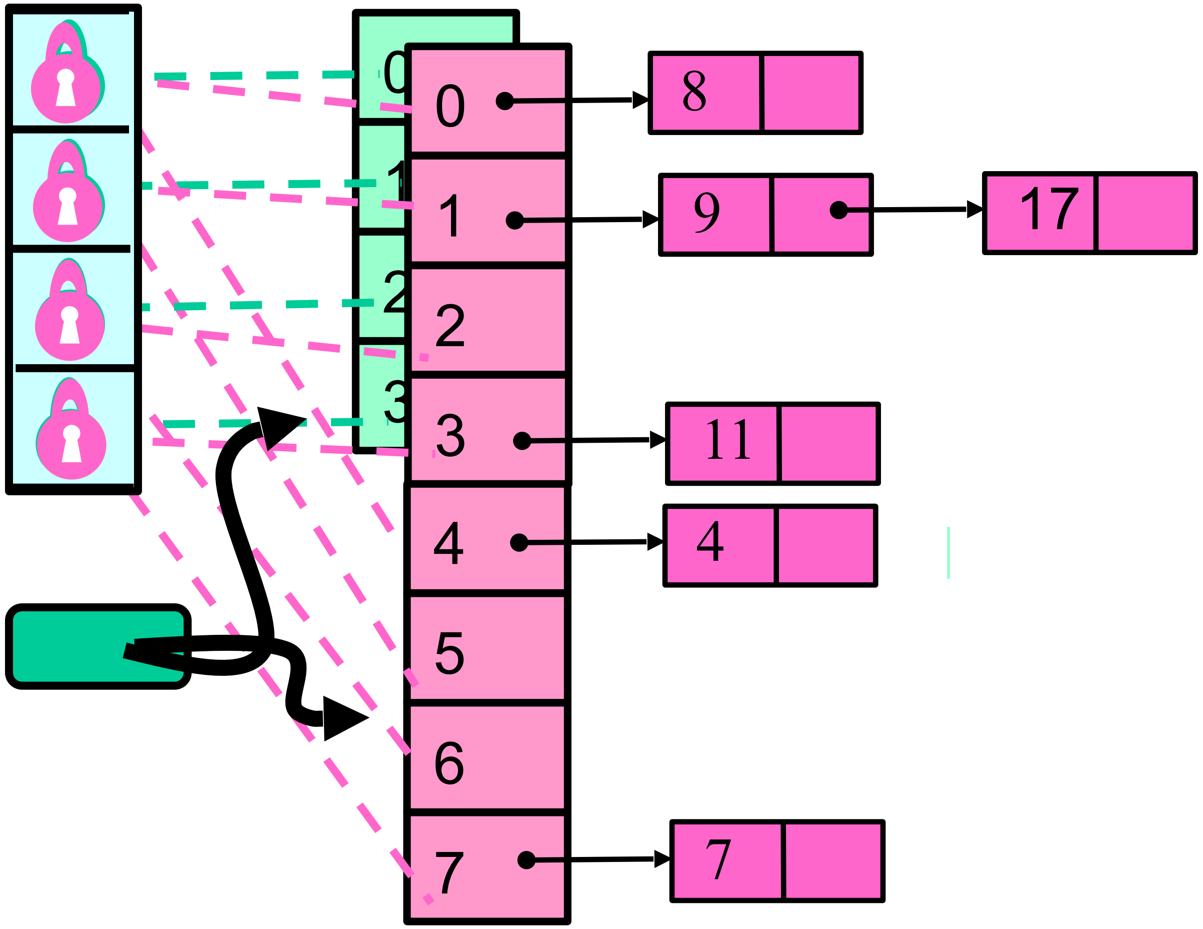
Resize This

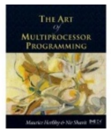




Striped Locks: each lock now associated with two buckets

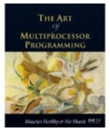
Resize This





Fine-Grained Hash Set

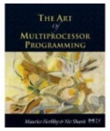
```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 public FGHashSet(int capacity) {
  table = new List[capacity];
  lock = new RangeLock[capacity];
  for (int i = 0; i < capacity; i++) {
   lock[i] = new RangeLock();
   table[i] = new LinkedList();
  }} …
```



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();
  }} …
```

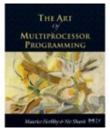
Array of locks



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();
  }} …
```

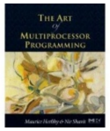
Array of buckets



Fine-Grained Hash Set

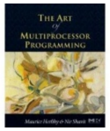
```
public class FGHashSet {
 protected RangeLock[] lock;
 protected List[] table;
 public FGHashSet(int capacity) {
  table = new List[capacity];
  lock = new RangeLock[capacity];
  for (int i = 0; i < capacity; i++) {
   lock[i] = new RangeLock();
   table[i] = new LinkedList();
  }} …
```

Initially *same* number of locks and buckets



The add() method

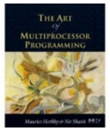
```
public boolean add(Object key) {
 int keyHash
  = key.hashCode() % lock.length;
 synchronized (lock[keyHash]) {
  int tabHash = key.hashCode() %
                  table.length;
  return table[tabHash].add(key);
 }
}
```



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);
 }
}
```

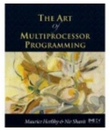
Which lock?



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);
 }
}
```

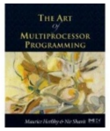
Acquire the lock



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);
 }
}
```

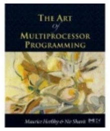
Which bucket?



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);
 }
}
```

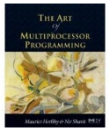
Call that bucket's add() method



Fine-Grained Locking

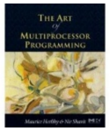
```
private void resize(int depth,
                    List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table){
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();
 }}}
```

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



Resizing

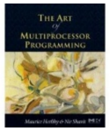
```
private void resize(int depth,
                    List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table){
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();
 }}}
```



Resizing

```
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();
 }}}
```

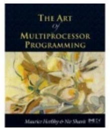
Acquire next lock



Resizing

```
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();
 }}}
```

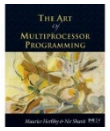
Check that no one else has resized



Resizing

```
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();
 }}}
```

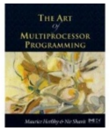
Recursively acquire next lock



Resizing

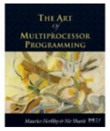
```
private void resize(int depth,
        List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table){
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();
 }}}
```

Locks acquired, do the work



Stop The World Resizing

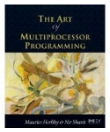
- Resizing stops all concurrent operations
- What about an incremental resize?
- Must avoid locking the table
- A lock-free table + incremental resizing? (See textbook)



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)

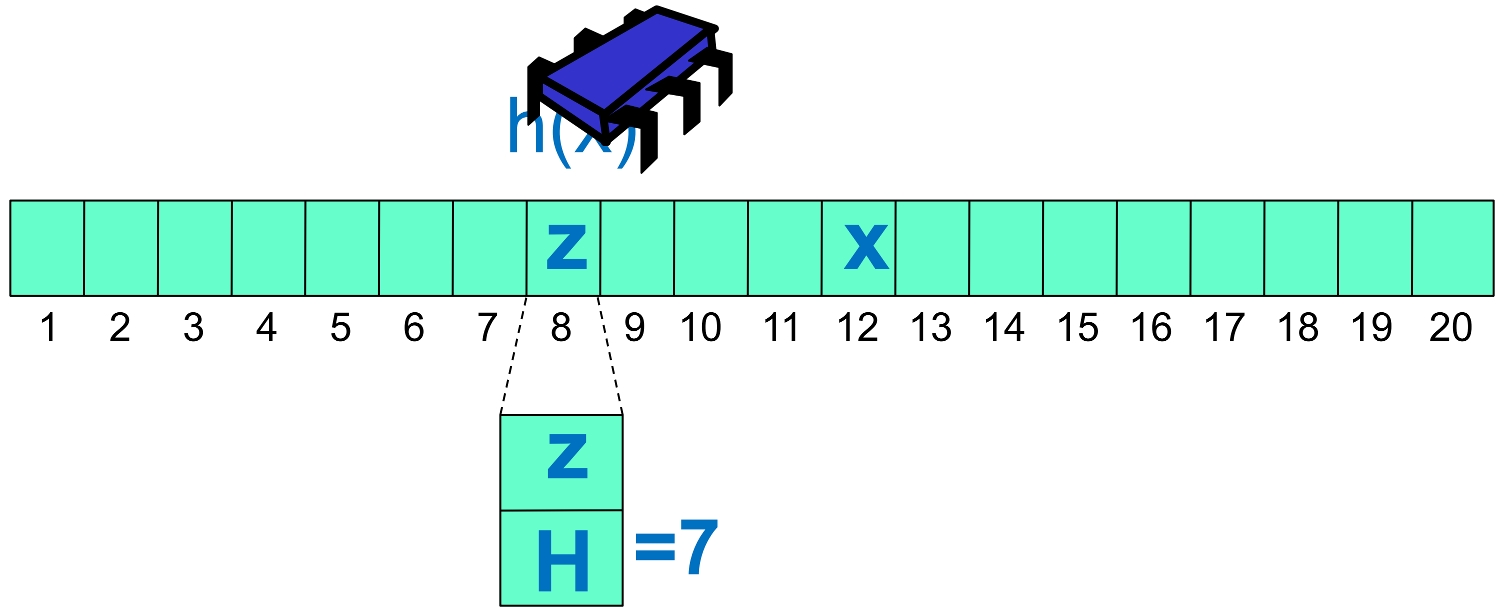
Oh, did we mention that cache behavior matters on a multicore?



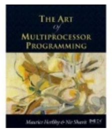
Open Addressed Hashing

- Keep all items in an array
- One per bucket
- If you have collisions, find an empty bucket and use it
- Must know how to find items if they are outside their bucket

Linear Probing*

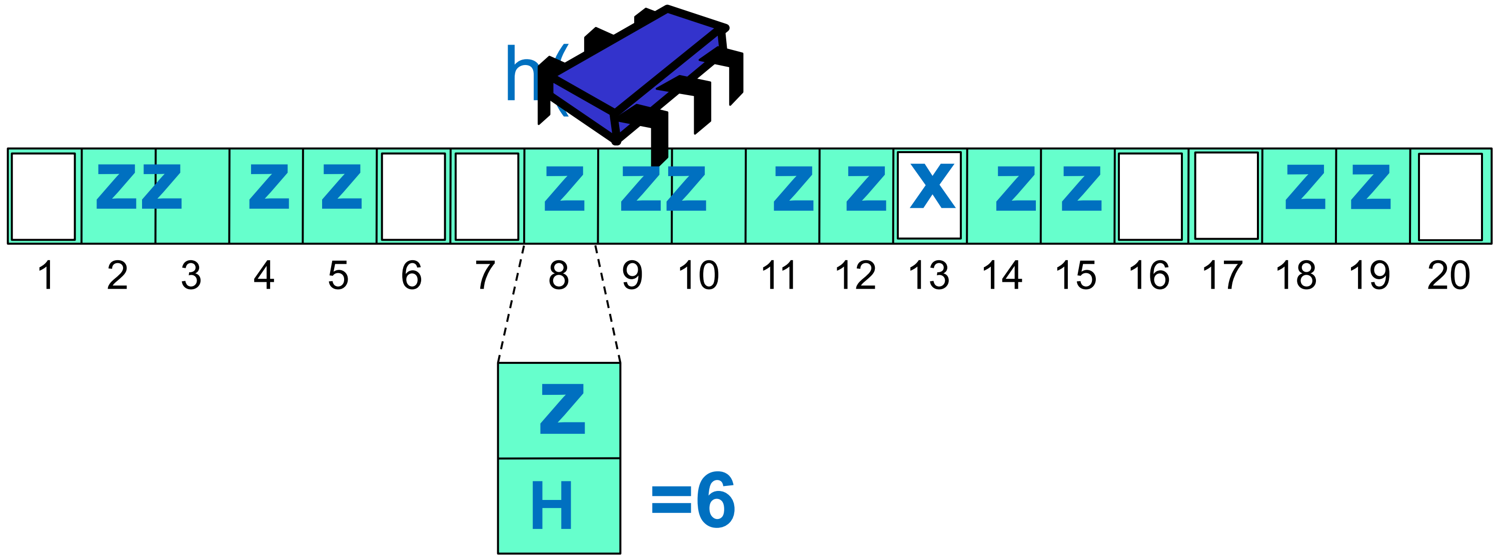


contains(x) – search linearly from h(x) to h(x) + H recorded in bucket.

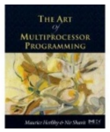


*Attributed to Amdahl…

Linear Probing



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



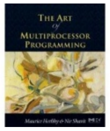
Linear Probing

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

$$½(1+(1/(1-M/N))\ ^{2}$$

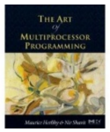
$M/N = 0.5$ ➔ search 2.5 buckets

$M/N = 0.9$ ➔ 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



Concurrent Open Address Hashing

- Need to either lock whole chain of displacements (see book)
- or have extra space to keep items as they are displaced step by step (Cuckoo hashing, see book).

Summary

- *Chained hash* with striped locking is simple and effective in many cases
- See Textbook: *Hopscotch (Concurrent Cuckoo Hashing)* with striped locking great cache behavior
- See Textbook: If incremental resizing needed go for *split-ordered*

Concurrent Pools

pool

- Data Structure similar to Set
  - Does not necessarily provide contains() method
  - Allows the same item to appear more than once
  - get() and set()

```
public interface Pool<T> {
  void put(T item);
  T get();
}
```

Queues & Stacks

- Both: pool of items
- Queue
  - enq() & deq()
  - First-in-first-out (FIFO) order
- Stack
  - push() & pop()
  - Last-in-first-out (LIFO) order

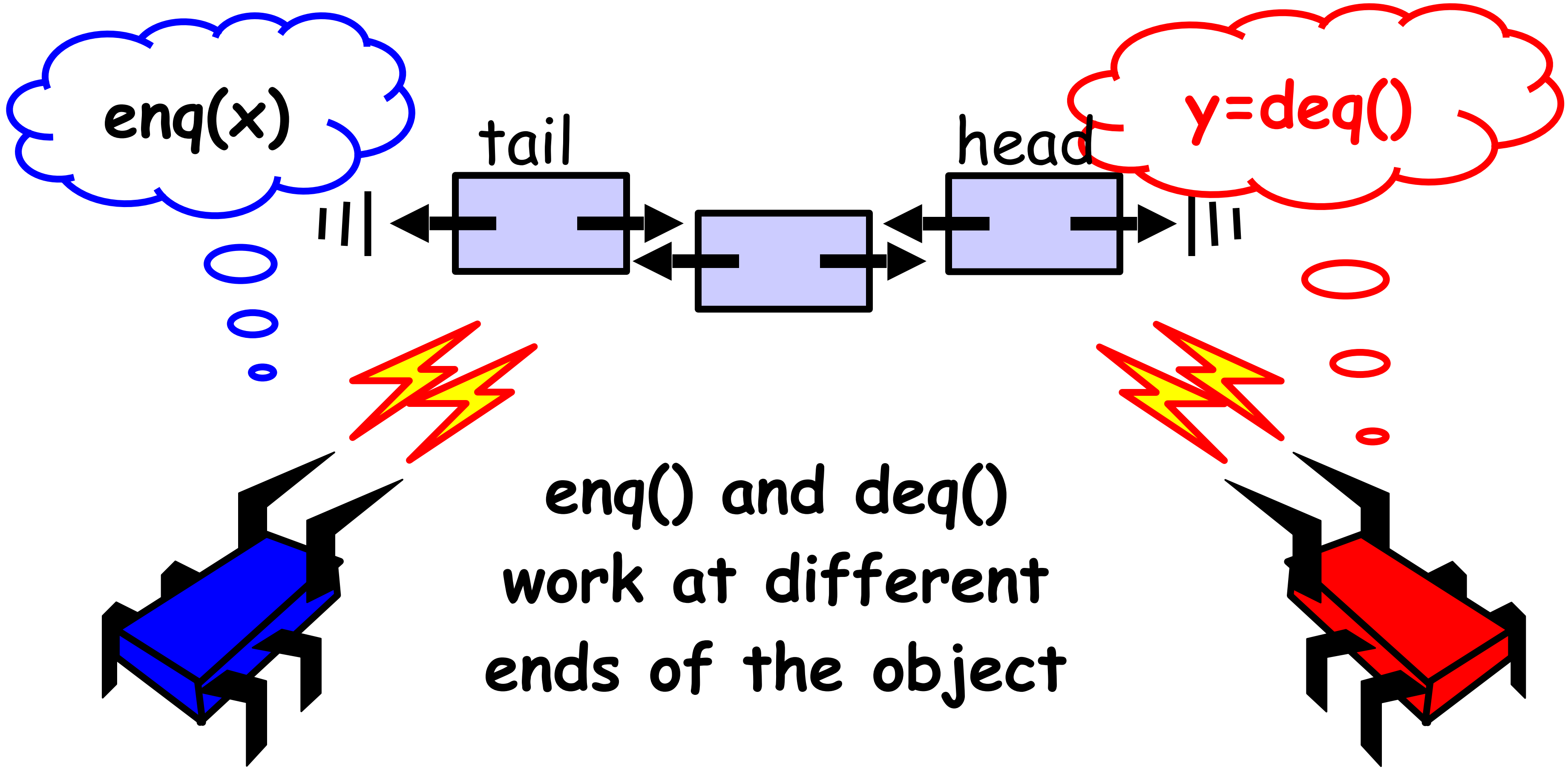
Bounded vs Unbounded

- Bounded
  - Fixed capacity
  - Good when resources an issue
- Unbounded
  - Holds any number of objects

Blocking vs Non-Blocking

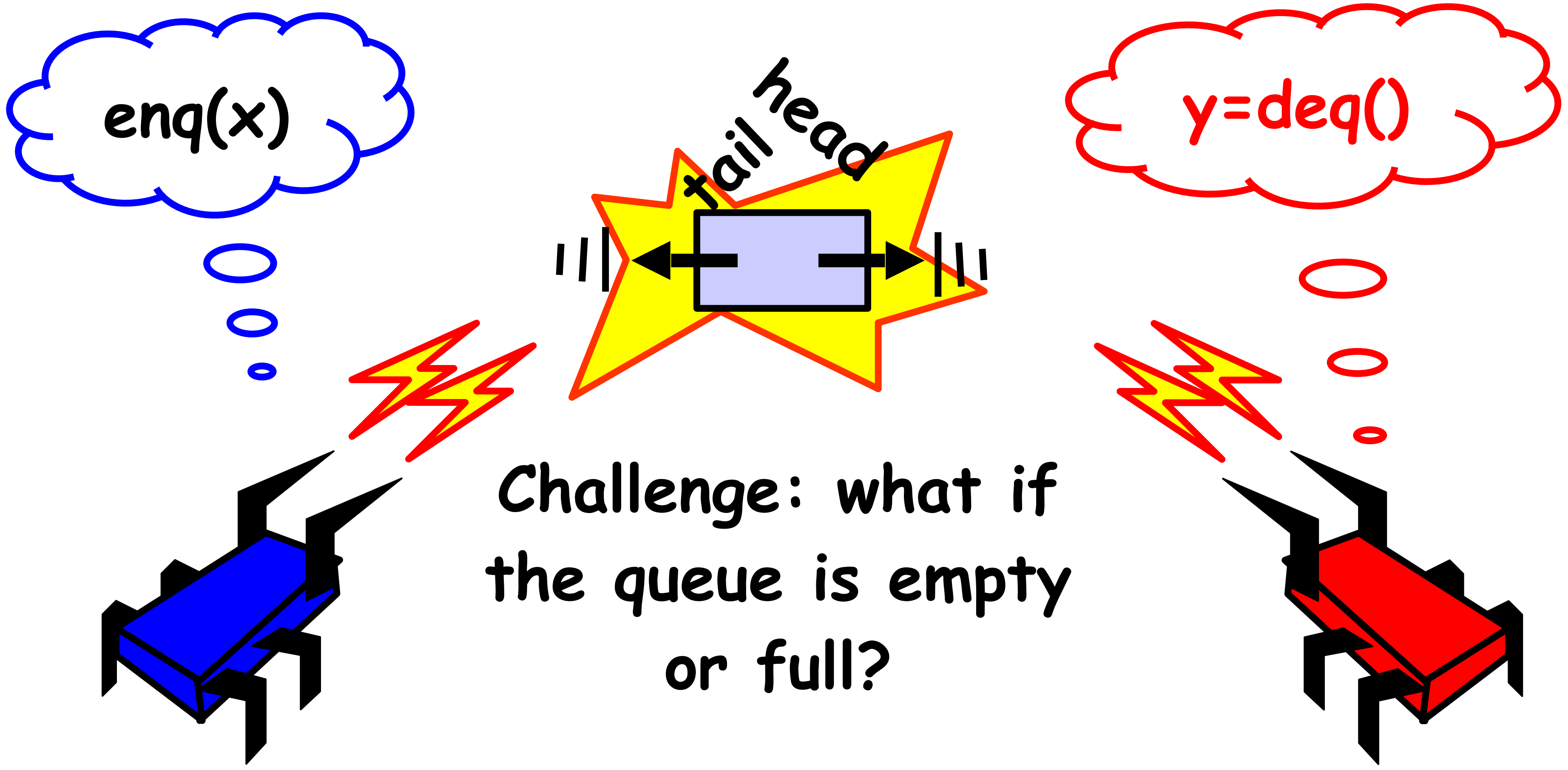
- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception or error

Queue: Concurrency



enq() and deq() work at different ends of the object

Concurrency



Challenge: what if the queue is empty or full?

lock

- enqLock/deqLock
  - At most one enqueuer/dequeuer at a time can manipulate the queue’s fields
- Two locks
  - Enqueuer does not lock out dequeuer
  - vice versa
- Association
  - enqLock associated with notFullCondition
  - deqLock associated with notEmptyCondition

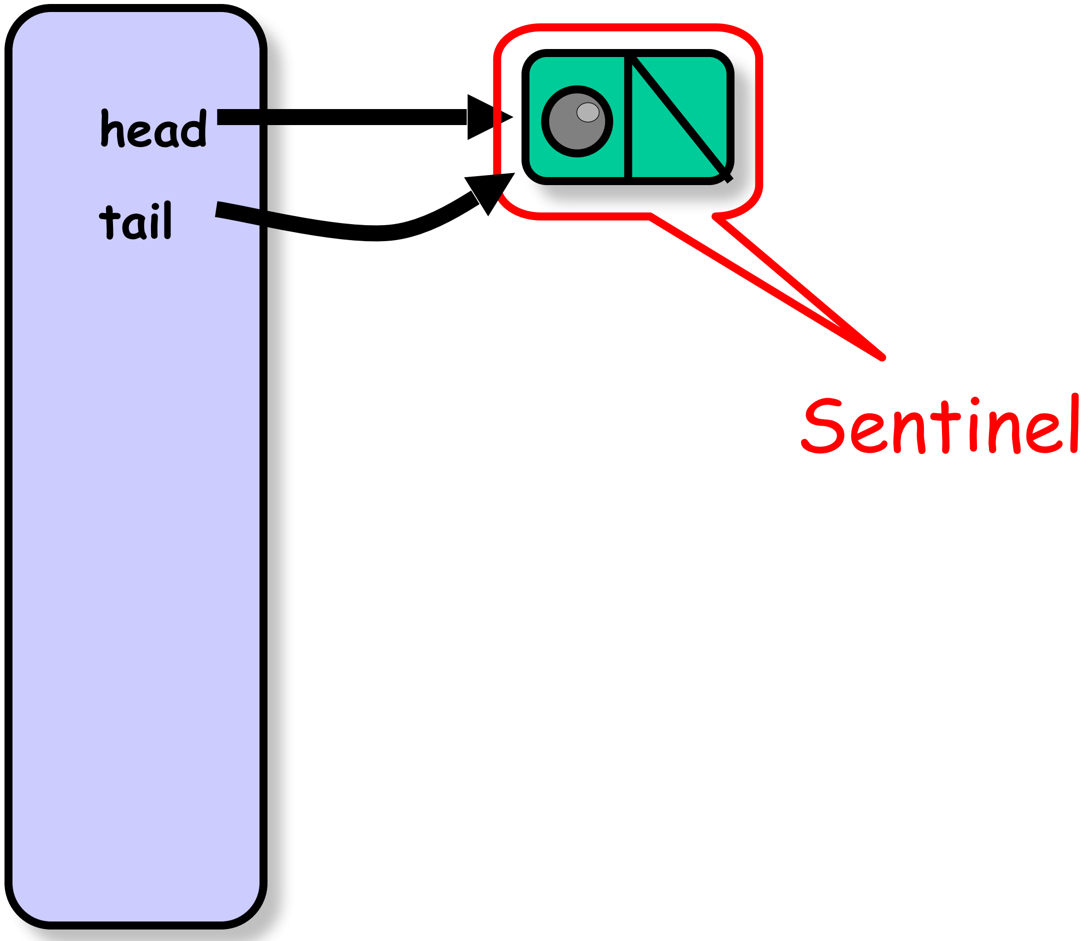
enqueue

1. Acquires enqLock
2. Reads the size field
3. If full, enqueuer must wait until dequeuer makes room
4. enqueuer waits on notFullCondition field, releasing enqLock temporarily, and blocking until that condition is signaled.
5. Each time the thread awakens, it checks whether there is a room, and if not, goes back to sleep
6. Insert new item into tail
7. Release enqLock
8. If queue was empty, notify/signal waiting dequeuers

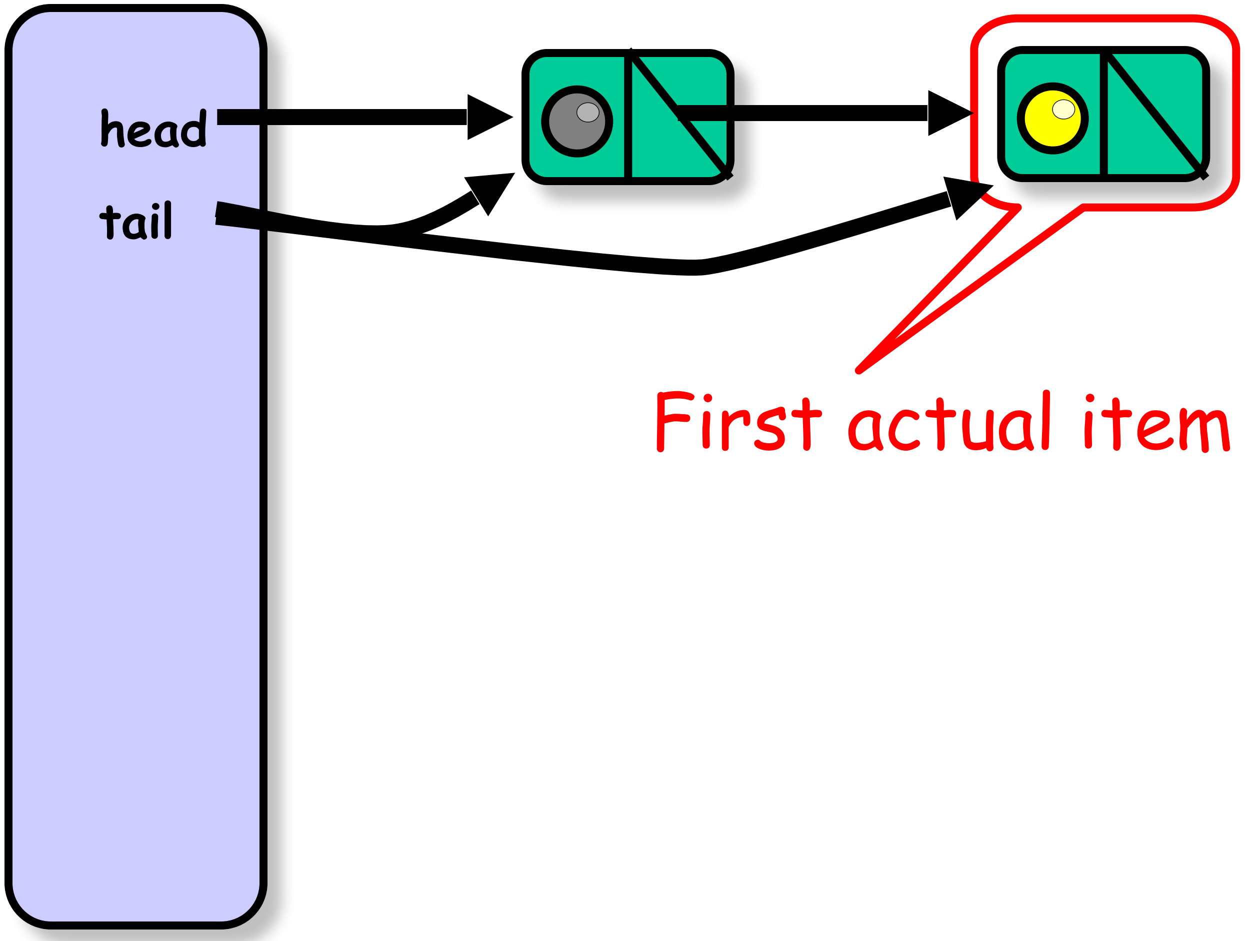
dequeue

1. Acquires deqLock
2. Reads the size field
3. If empty, dequeuer must wait until item is enqueued
4. dequeuer waits on notEmptyCondition field, releasing deqLock temporarily, and blocking until that condition is signaled.
5. Each time the thread awakens, it checks whether item was enqueued, and if not, goes back to sleep
6. Assign the value of head's next node to "result" and reset head to head's next node
7. Release deqLock
8. If queue was full, notify/signal waiting enqueuers
9. Return "result"

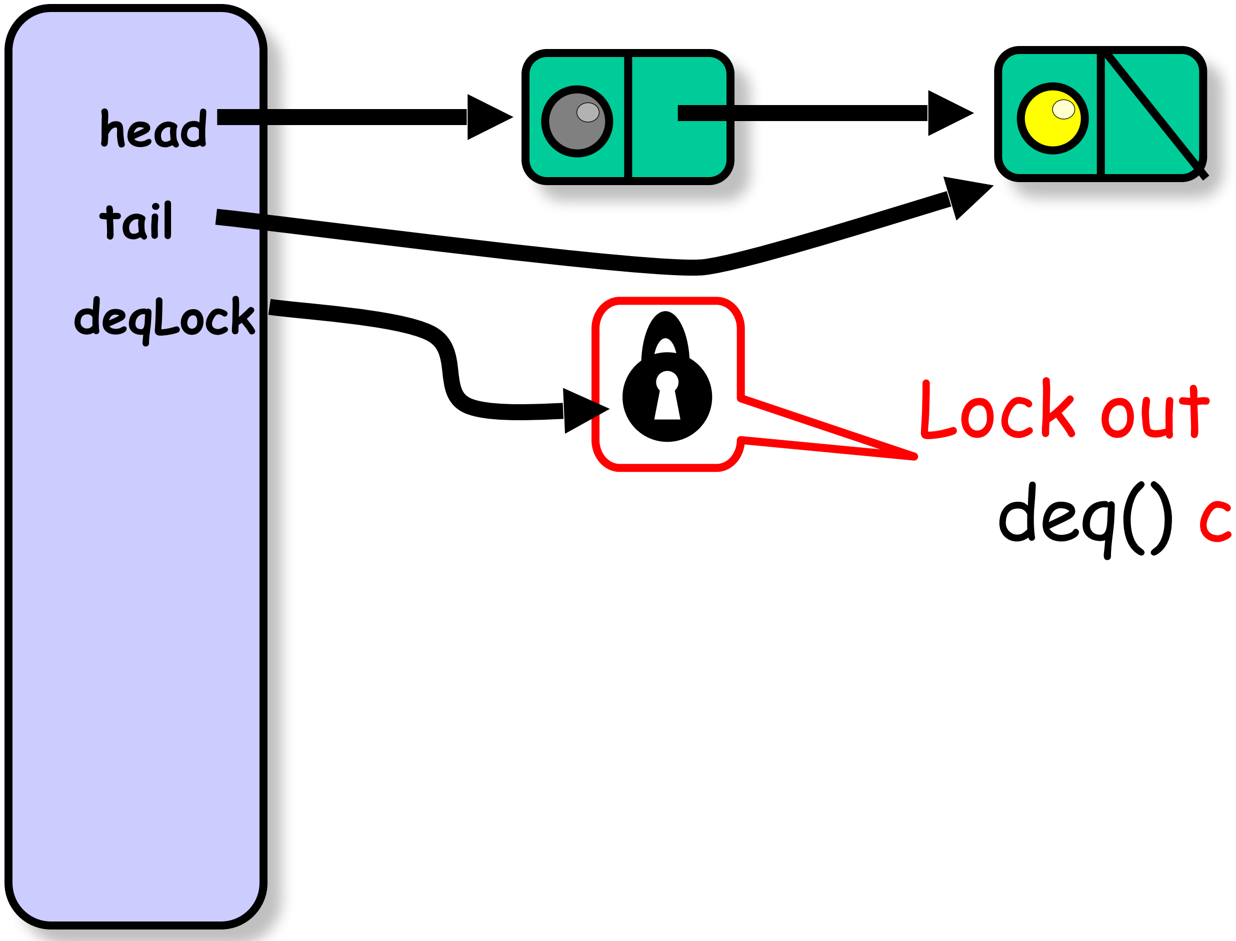
Bounded Queue



Bounded Queue

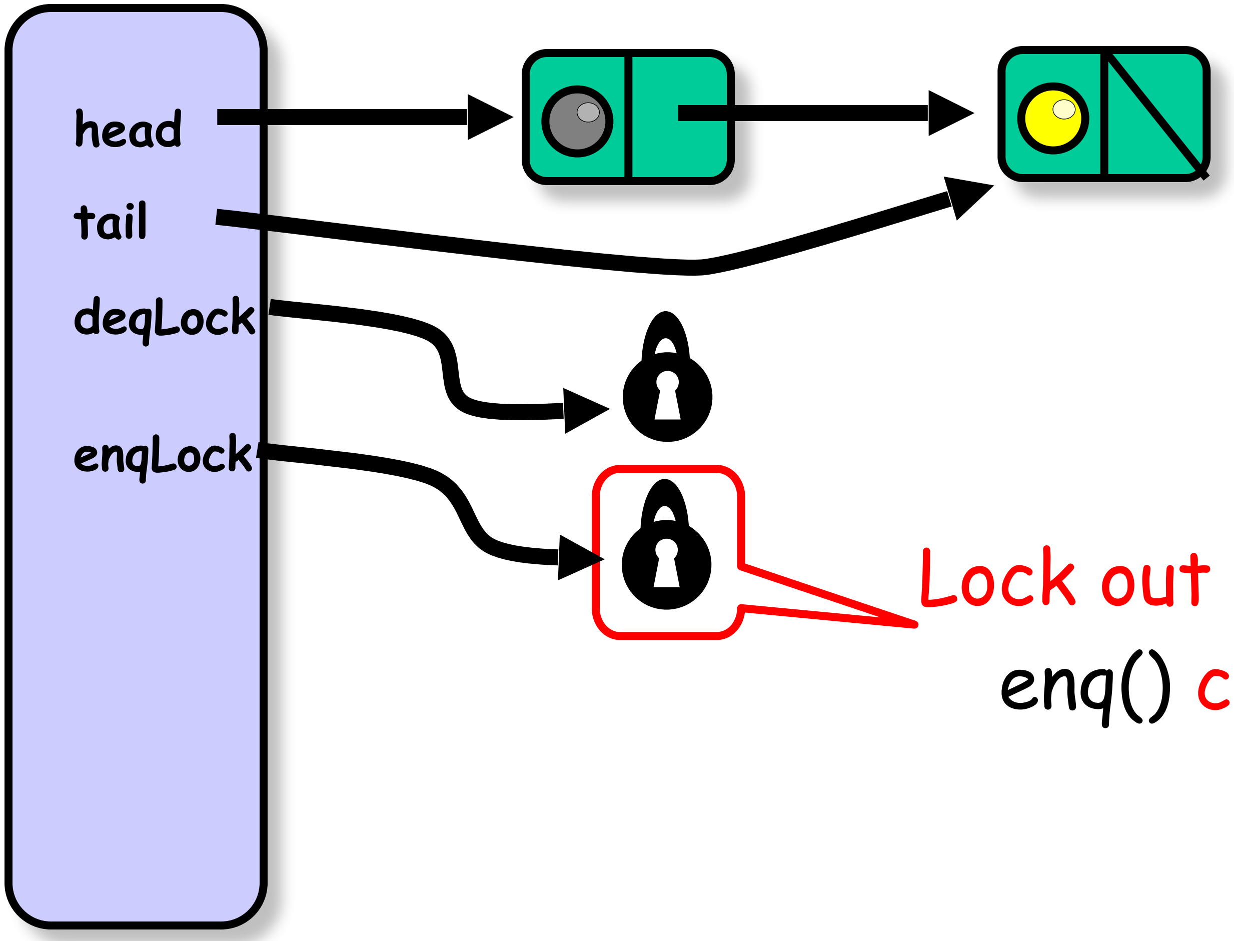


Bounded Queue



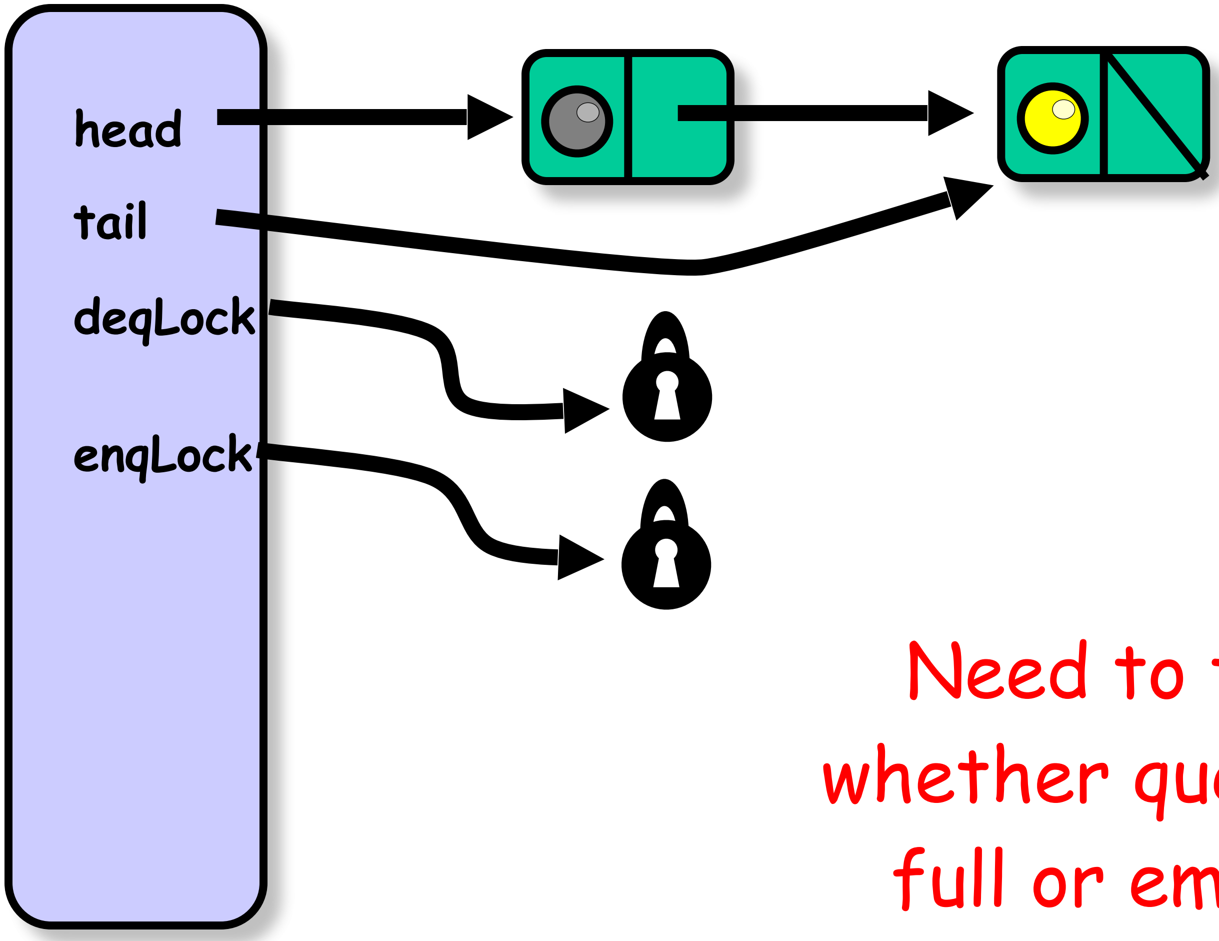
Lock out other deq() calls

Bounded Queue



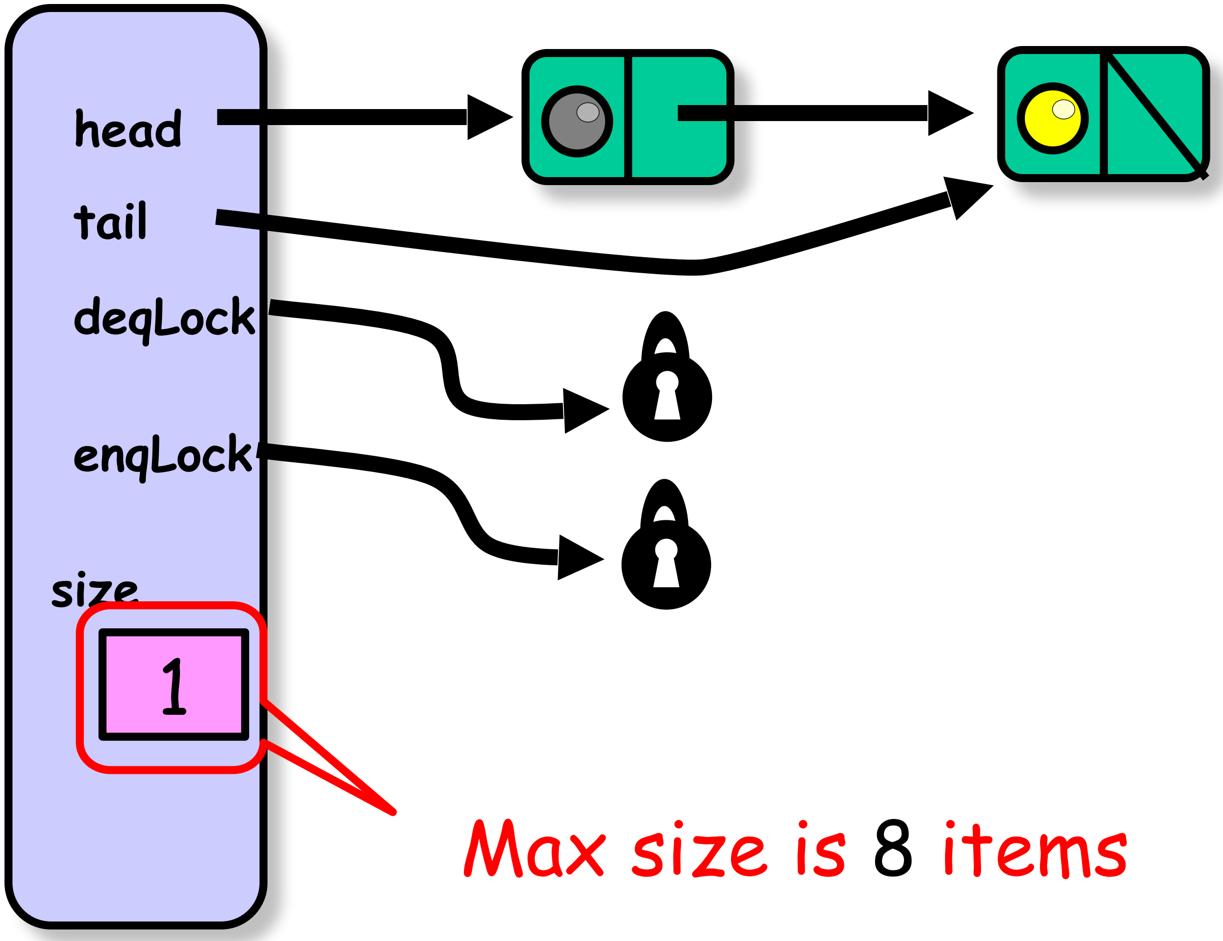
Lock out other enq() calls

Not Done Yet



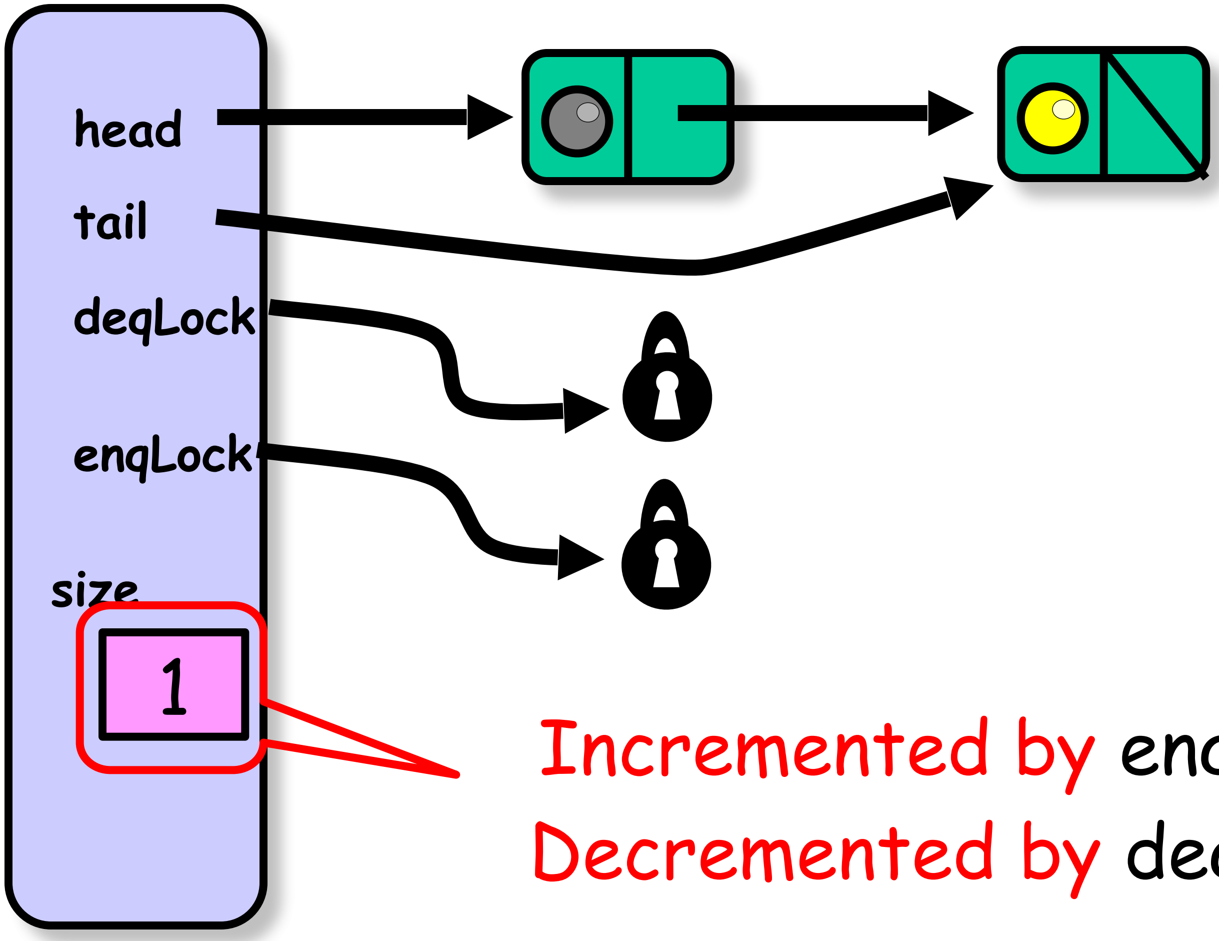
Need to tell whether queue is full or empty

Not Done Yet



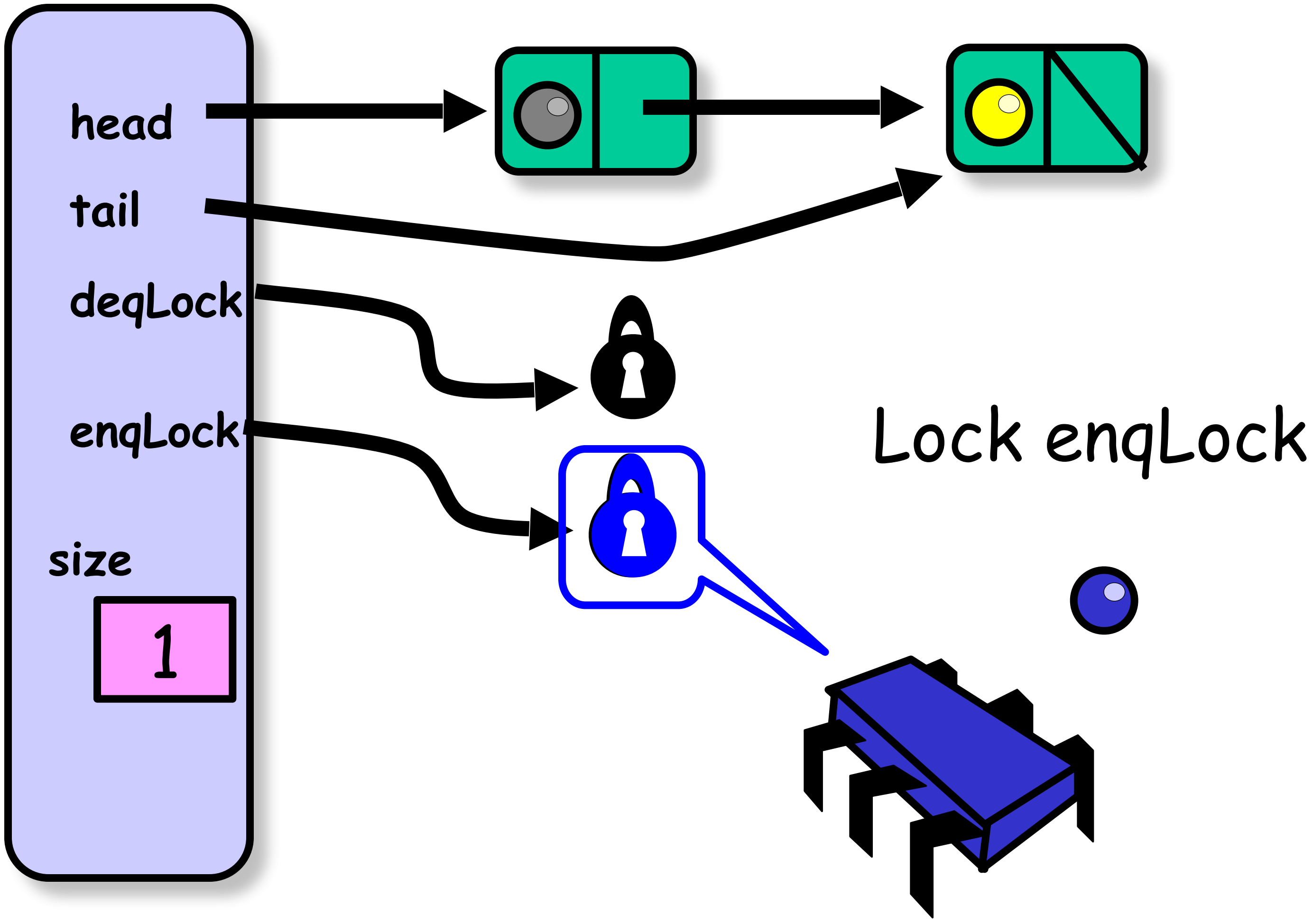
Max size is 8 items

Not Done Yet

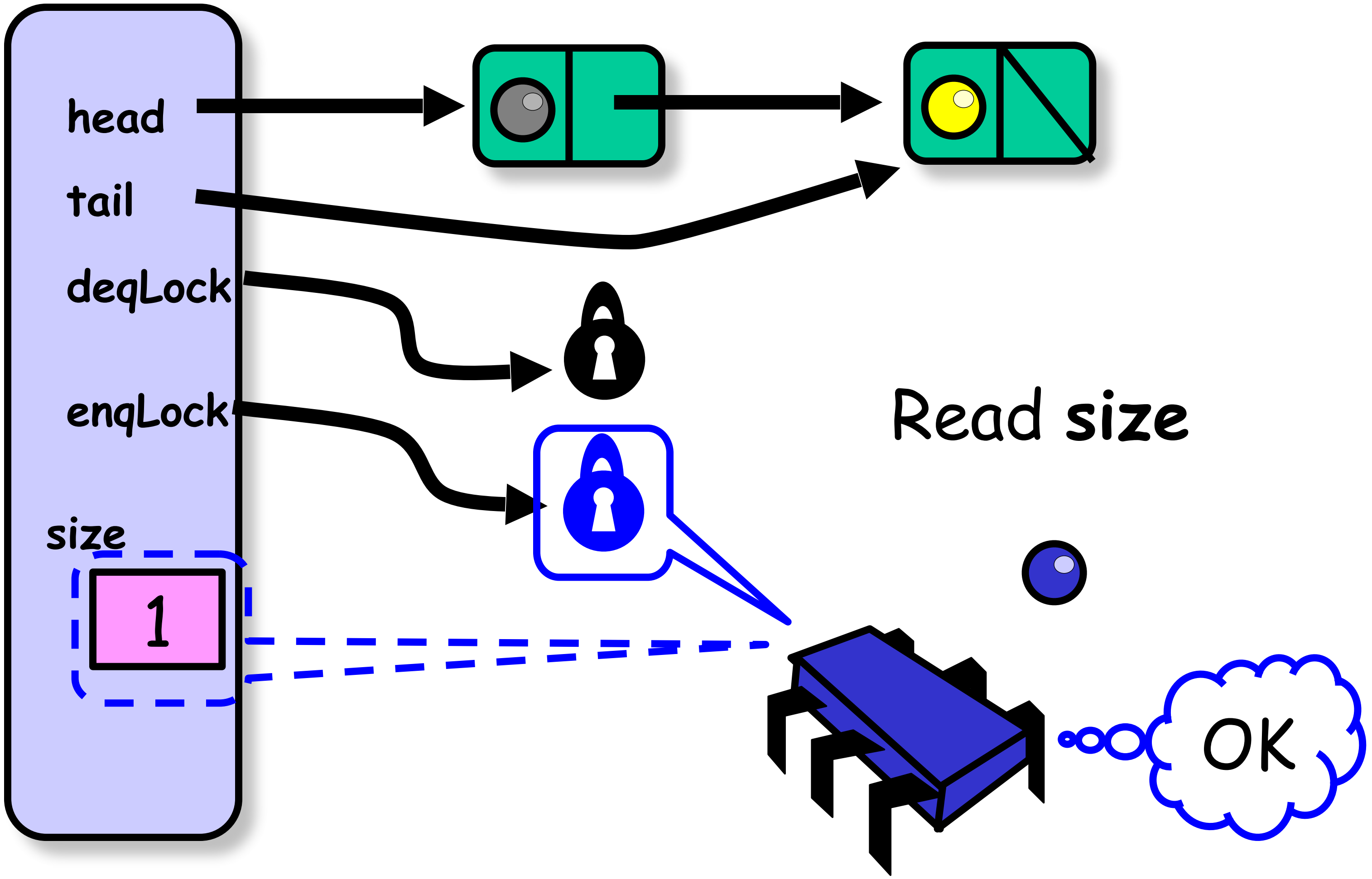


Incremented by enq()
Decremented by deq()

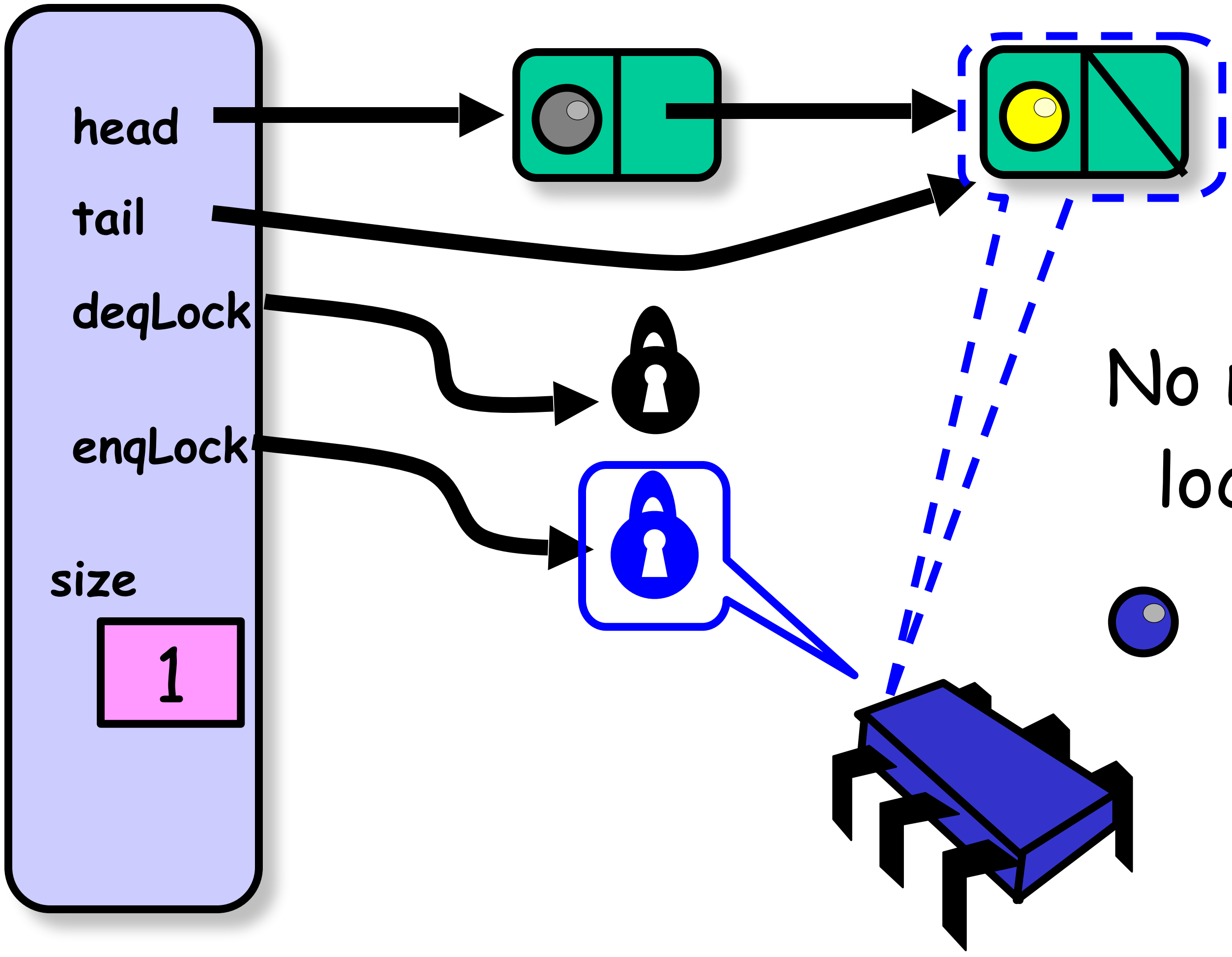
Enqueuer



Enqueuer

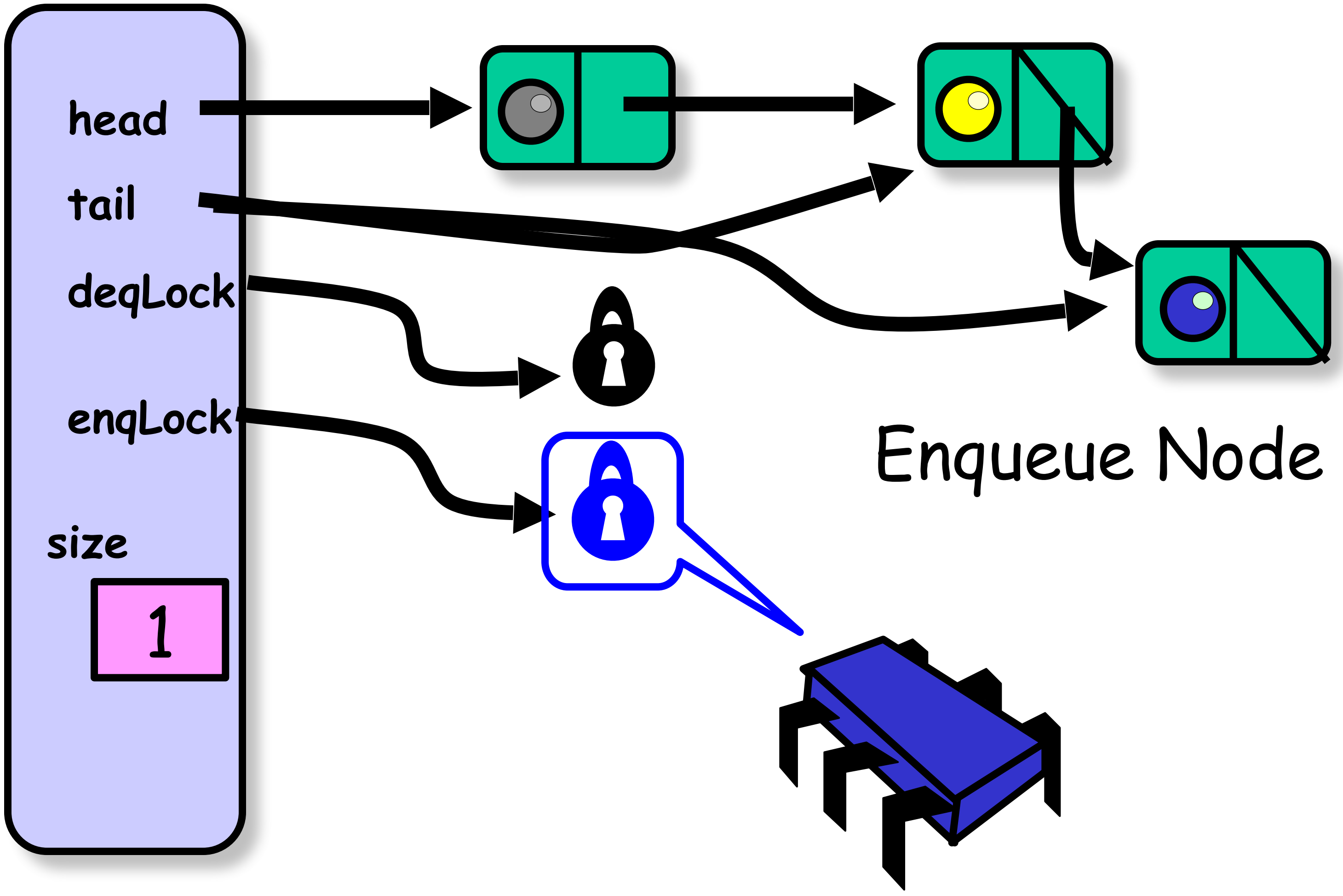


Enqueuer

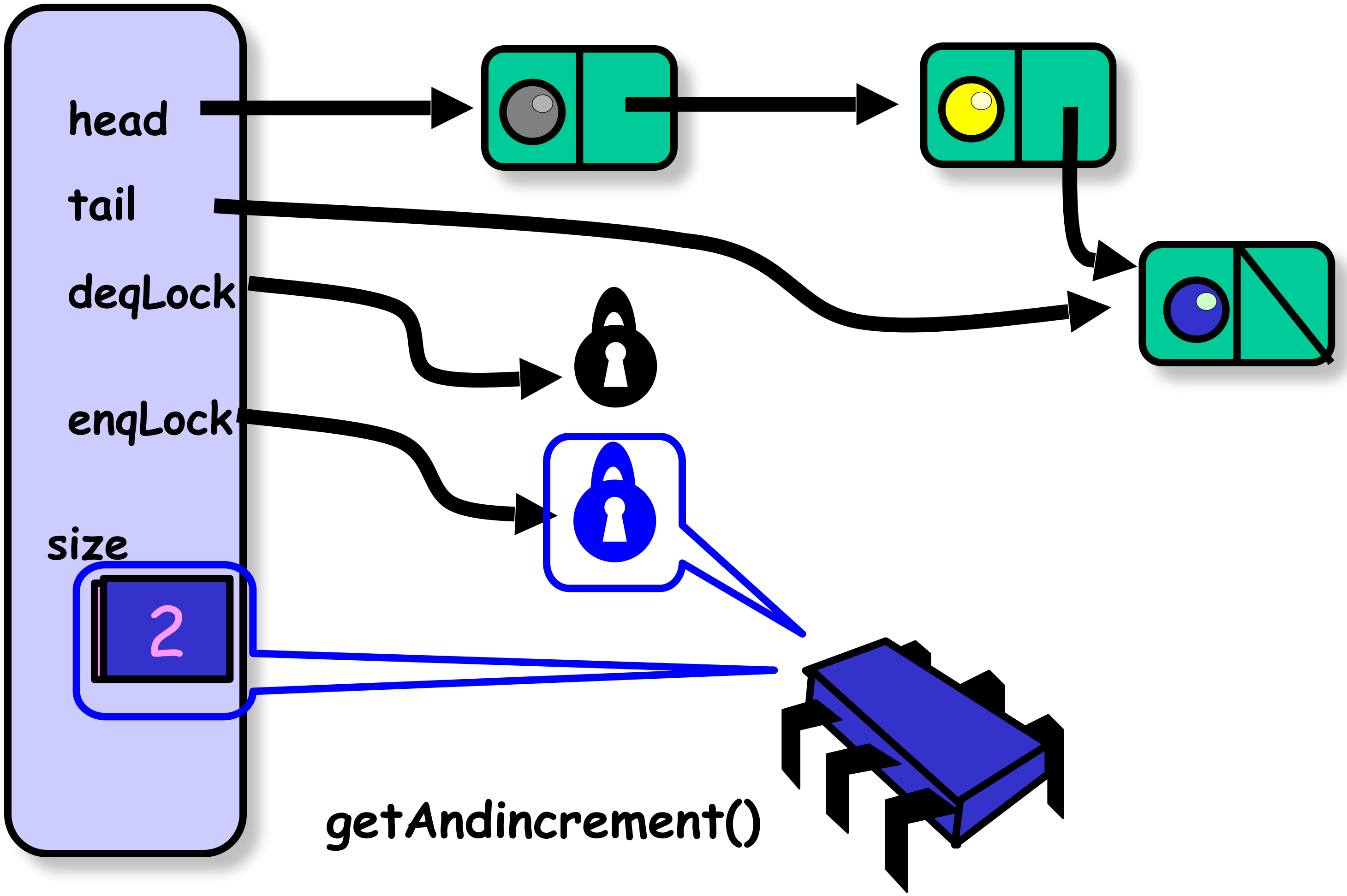


No need to lock **tail**

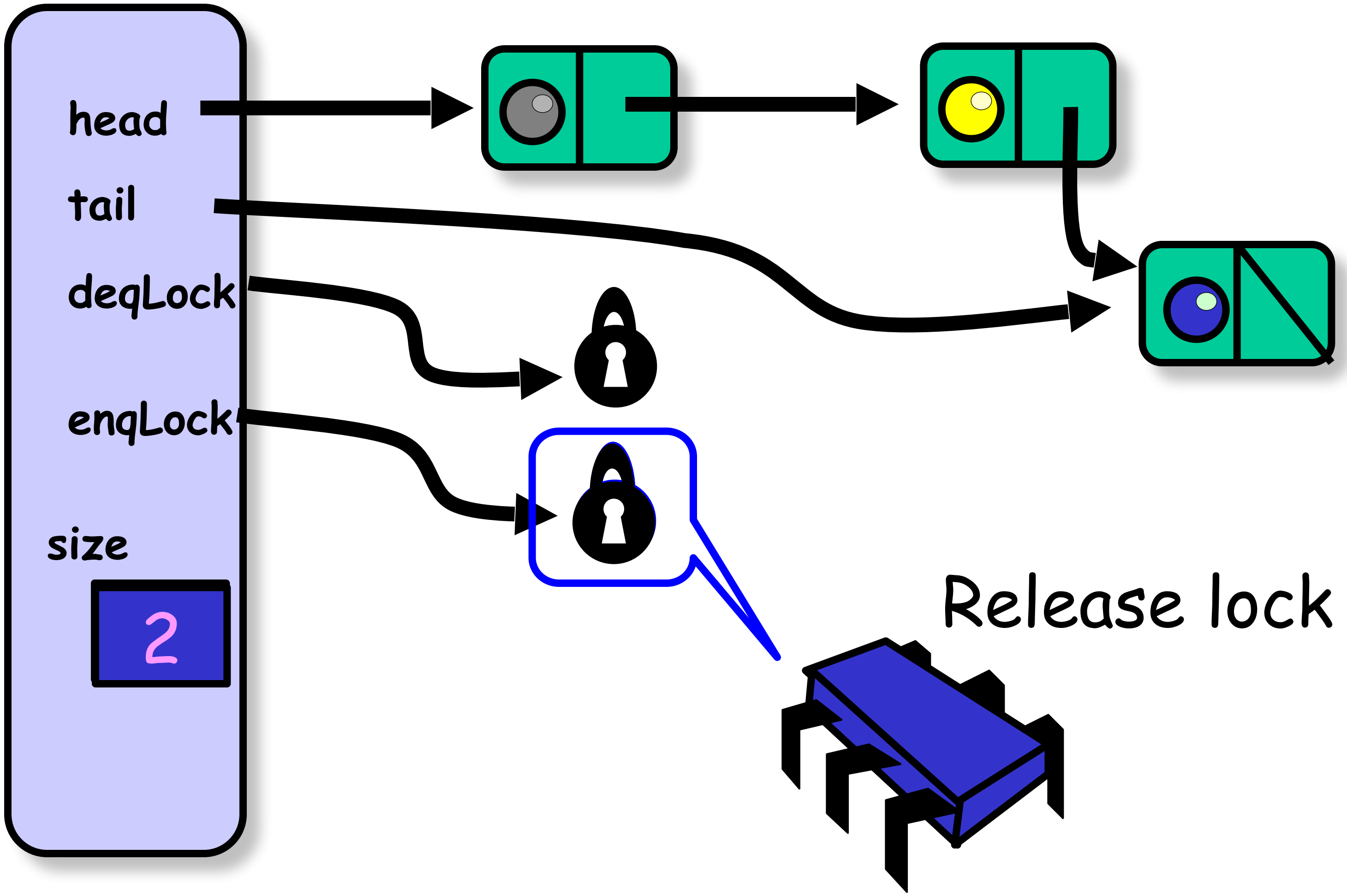
Enqueuer



Enqueuer

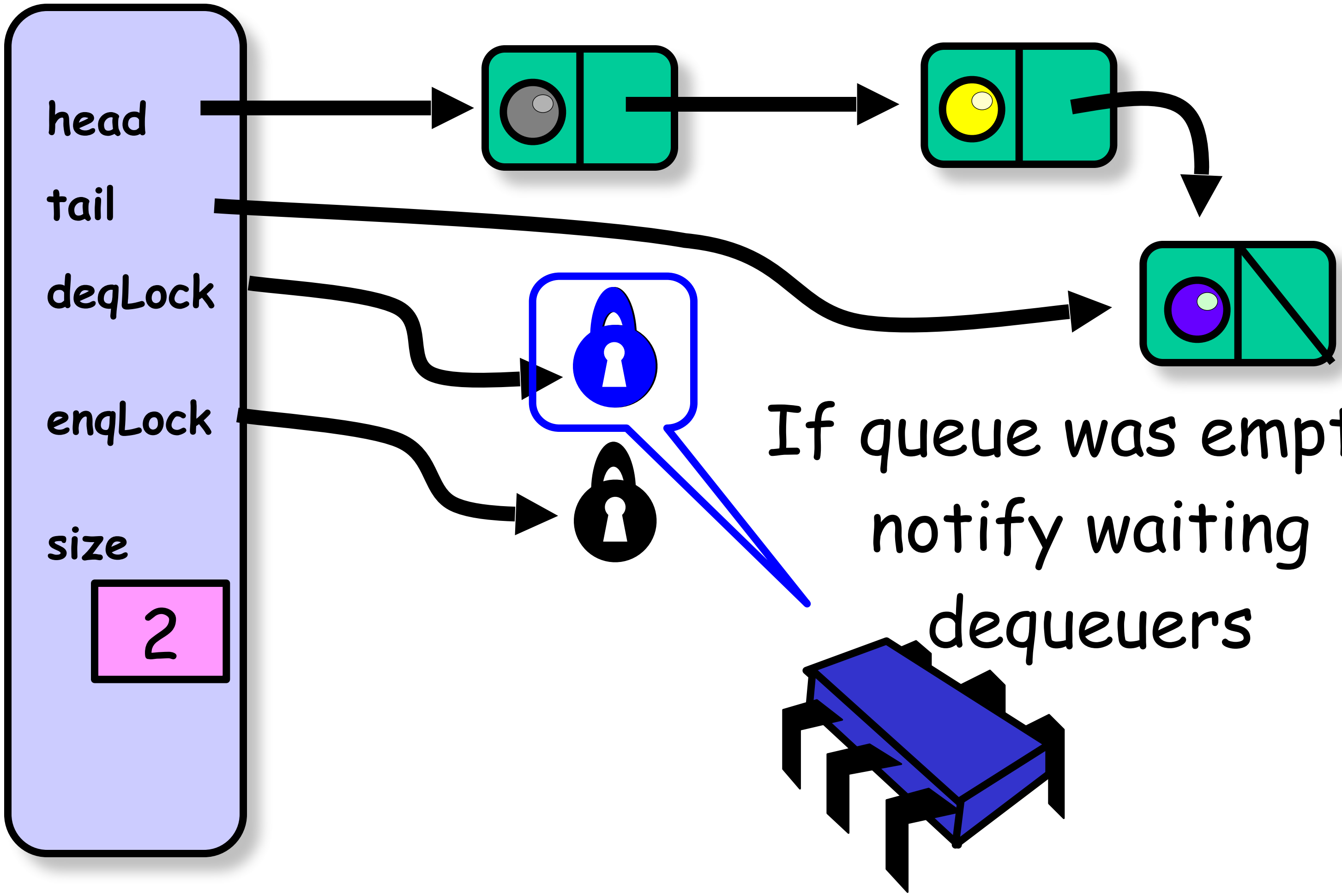


Enqueuer



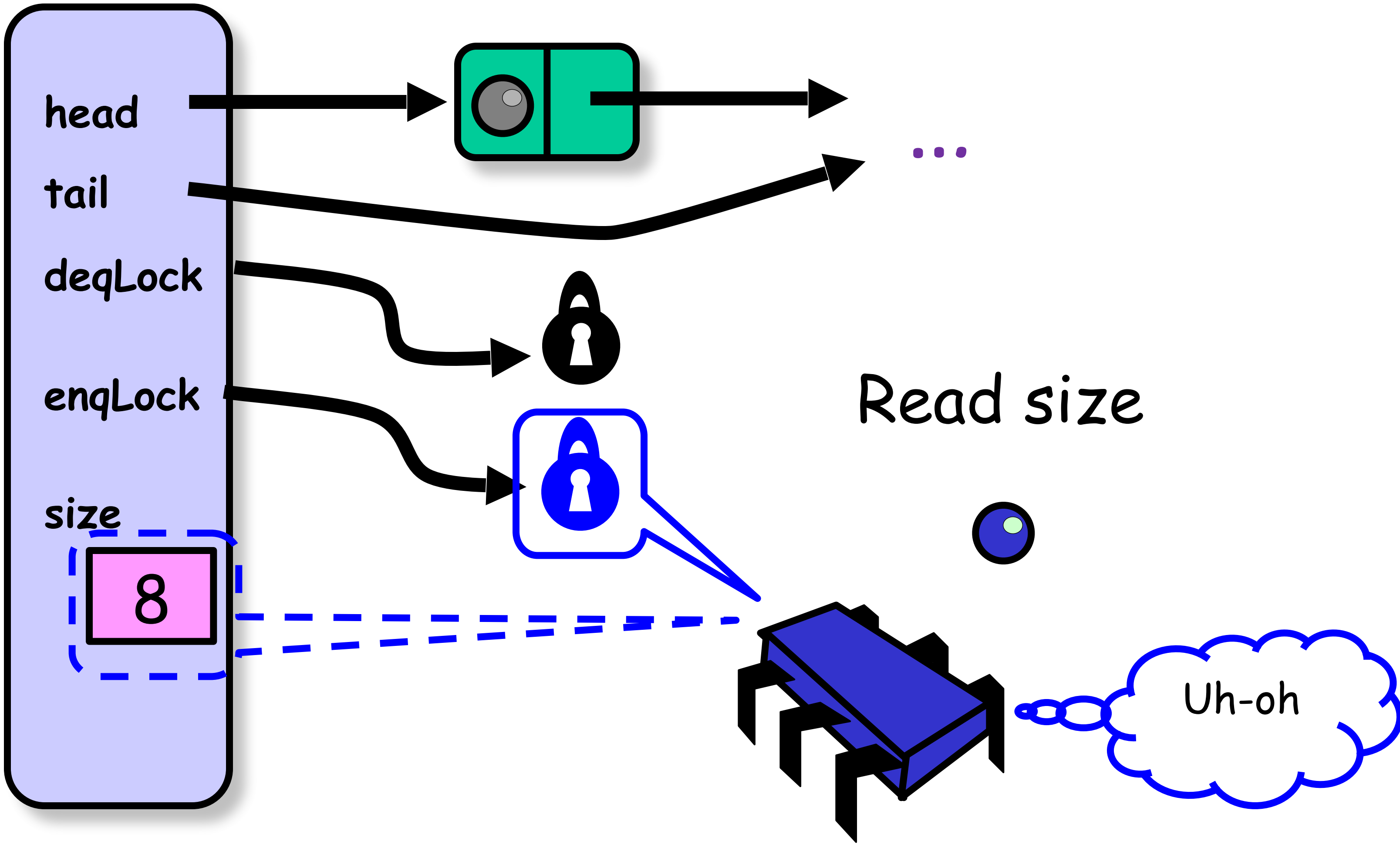
Release lock

Enqueuer

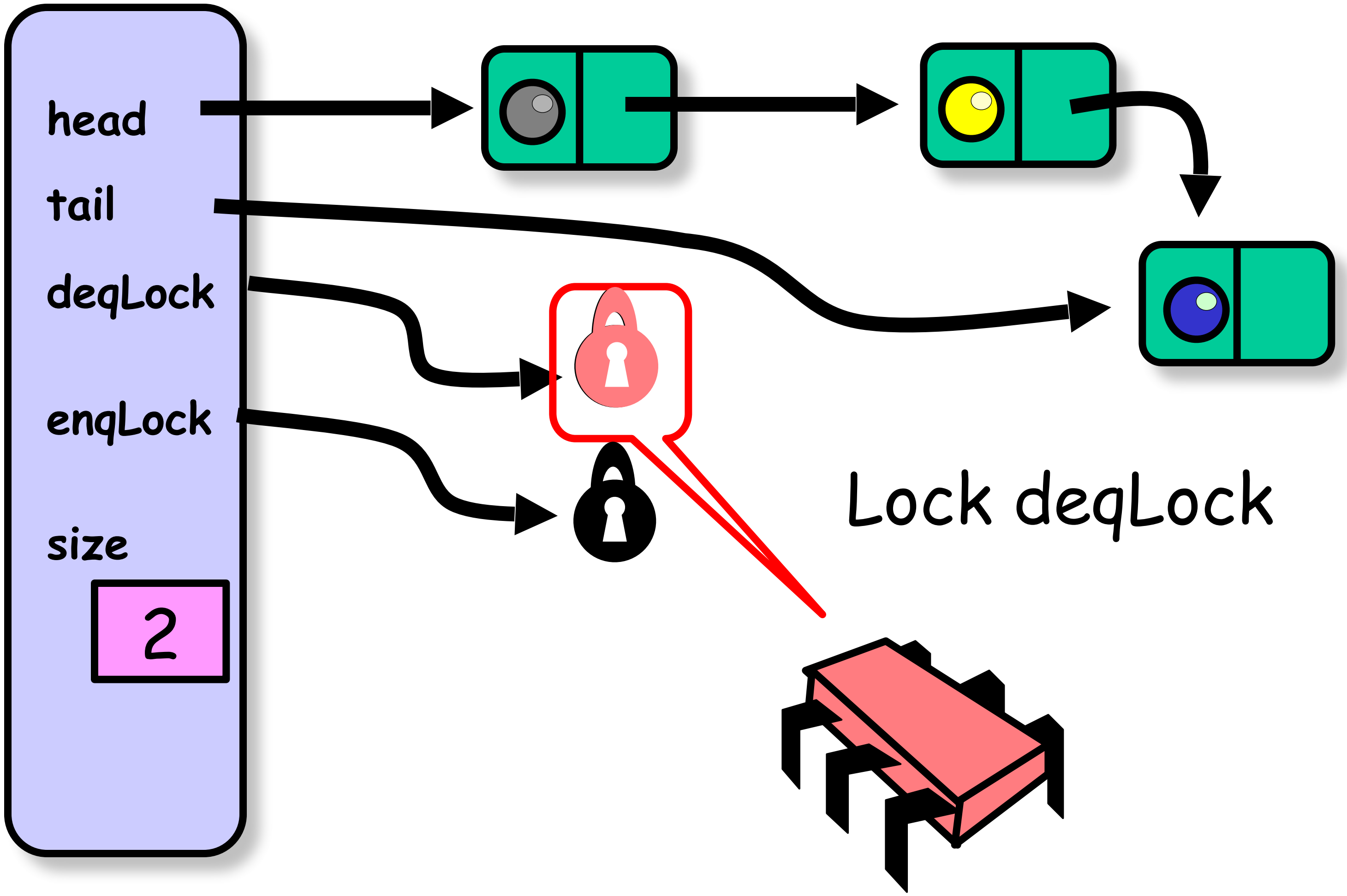


If queue was empty, notify waiting dequeuers

Unsuccesful Enqueuer

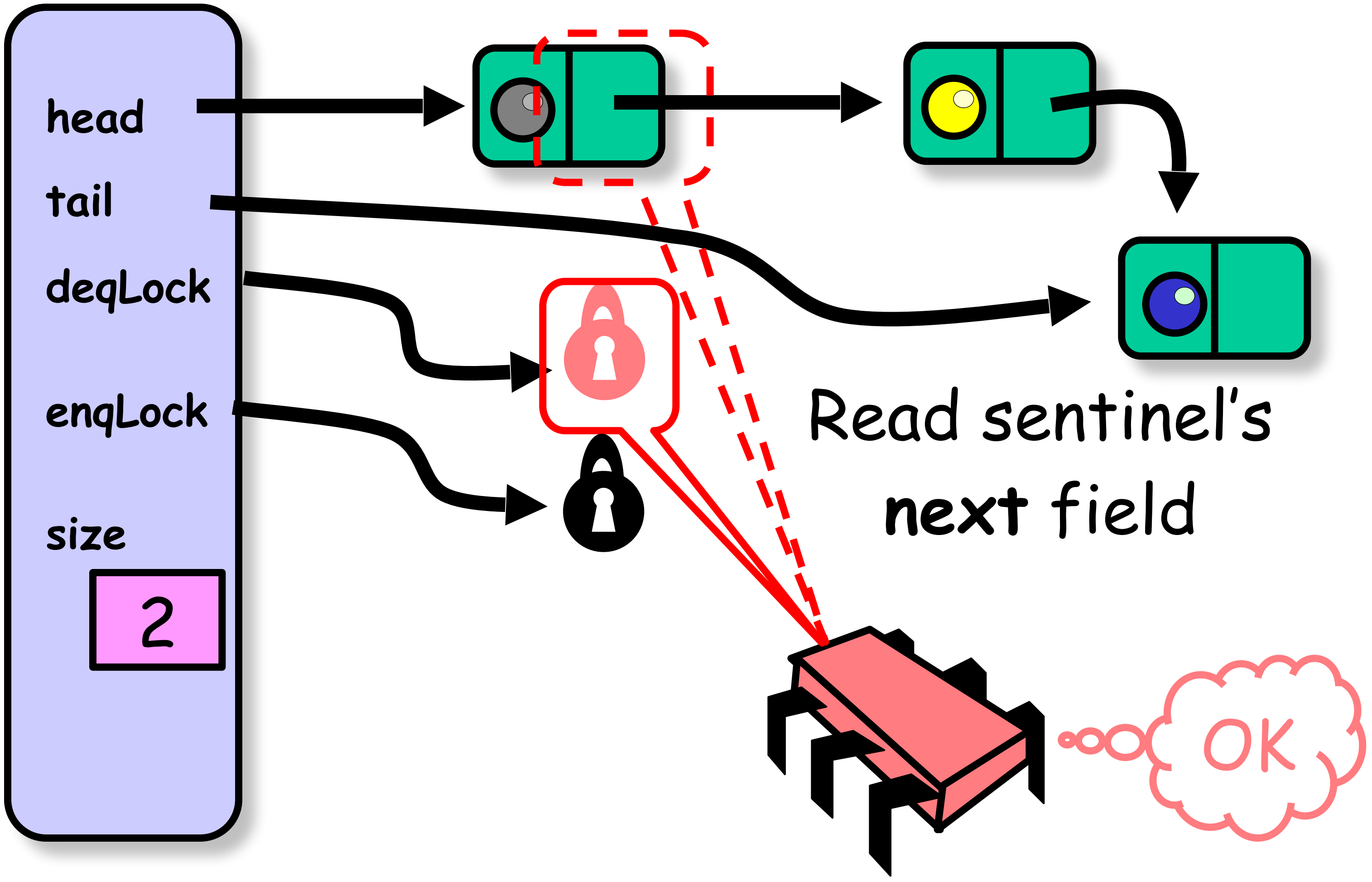


Dequeuer



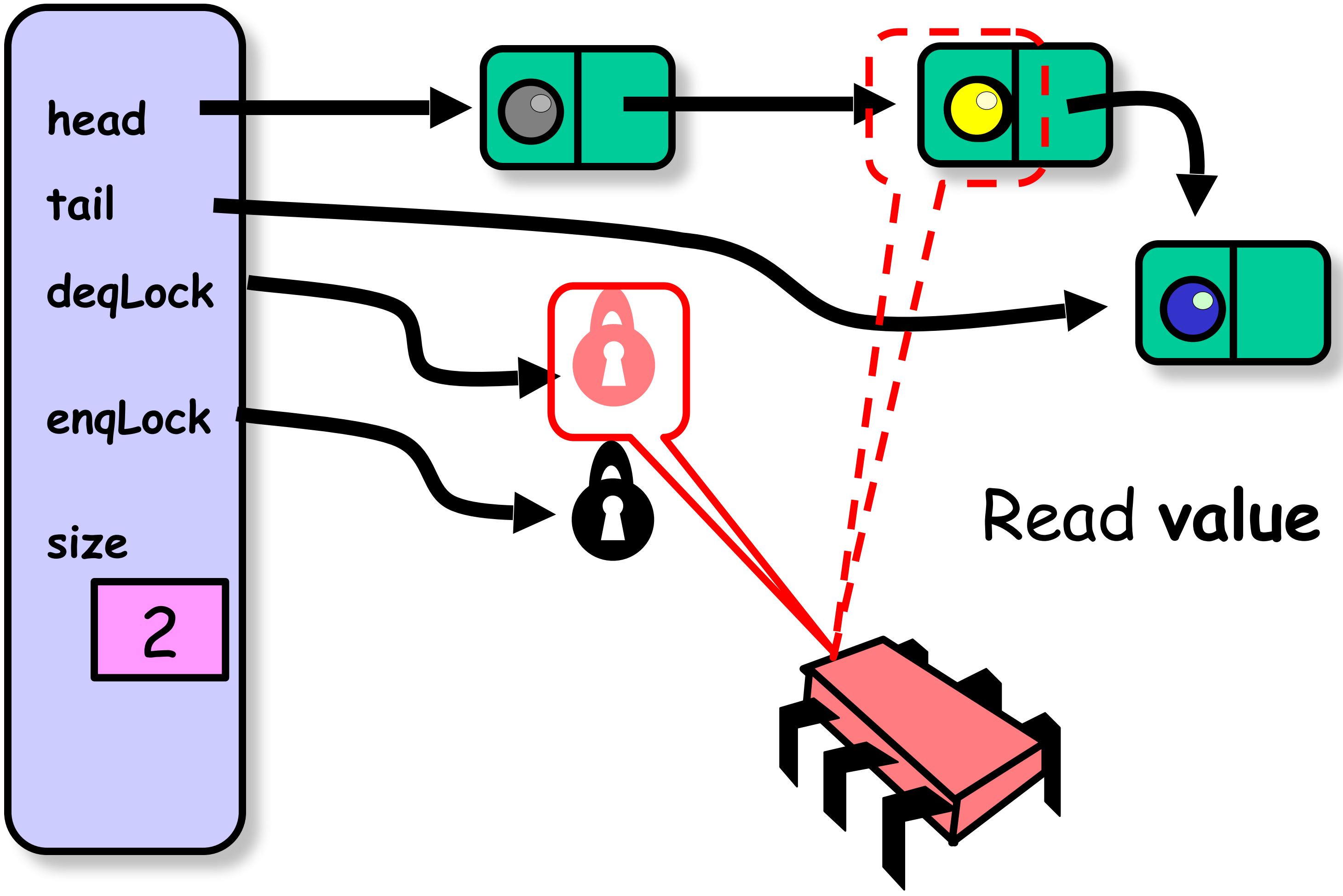
Lock deqLock

Dequeuer



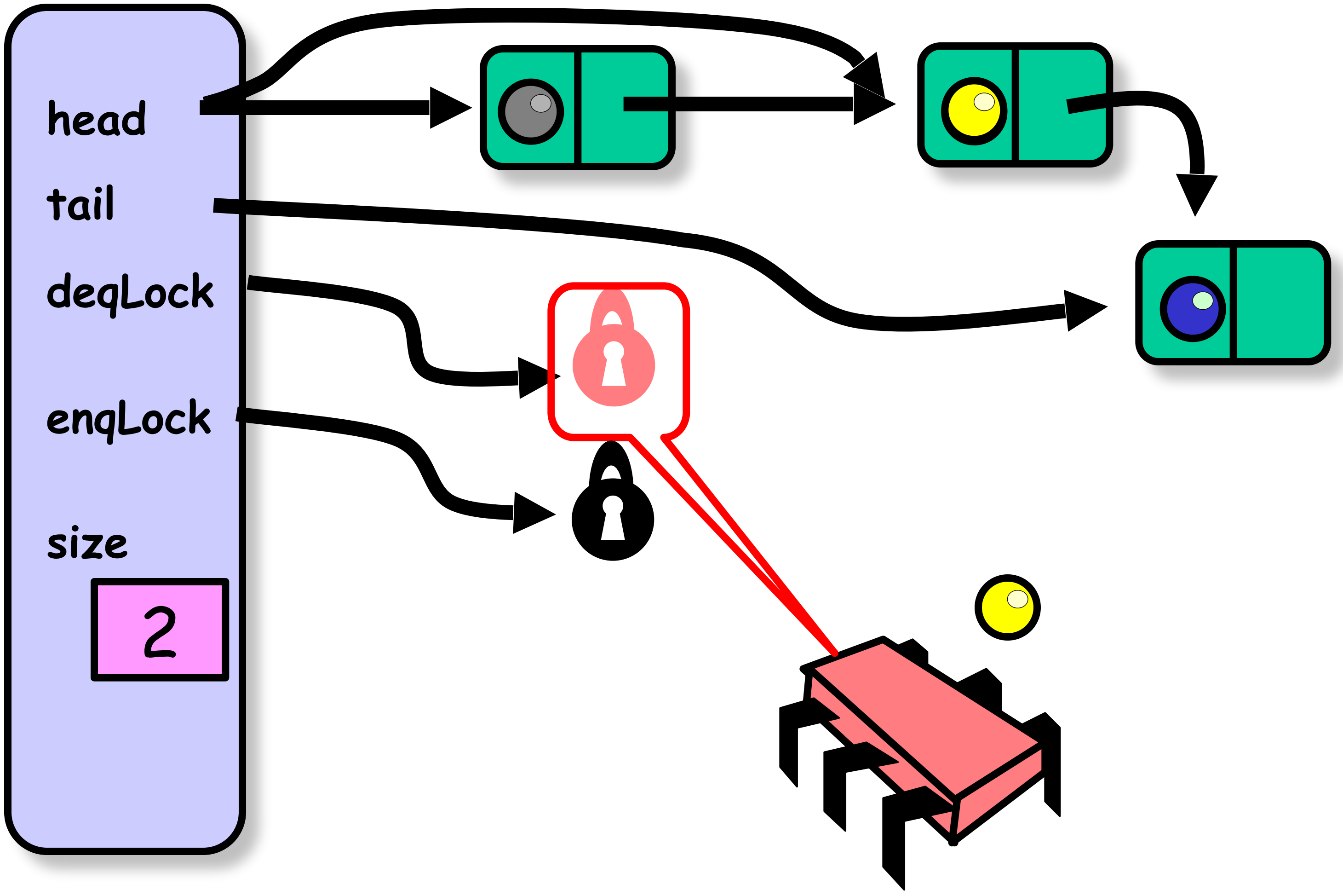
Read sentinel's next field

Dequeuer

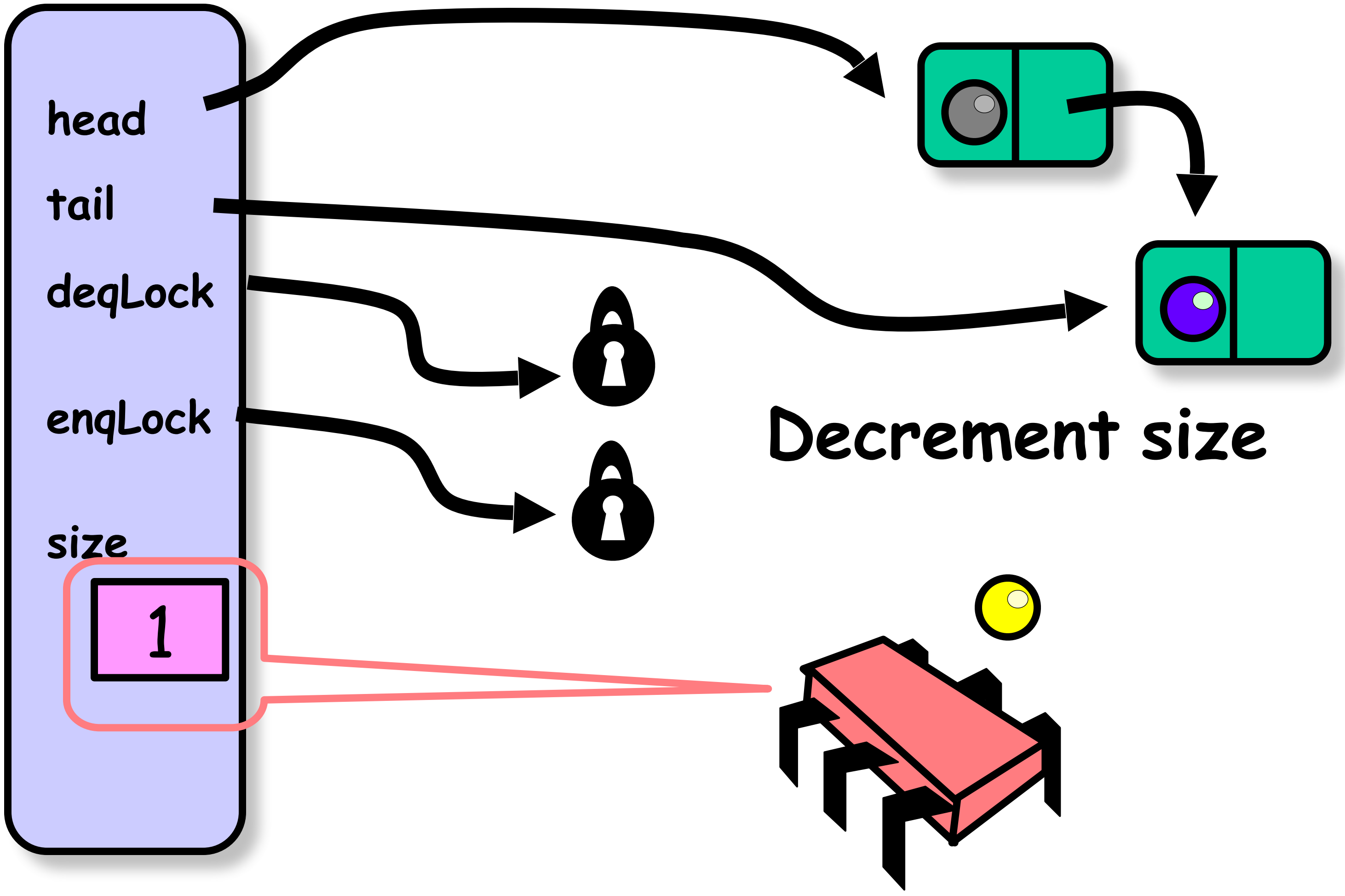


Dequeuer

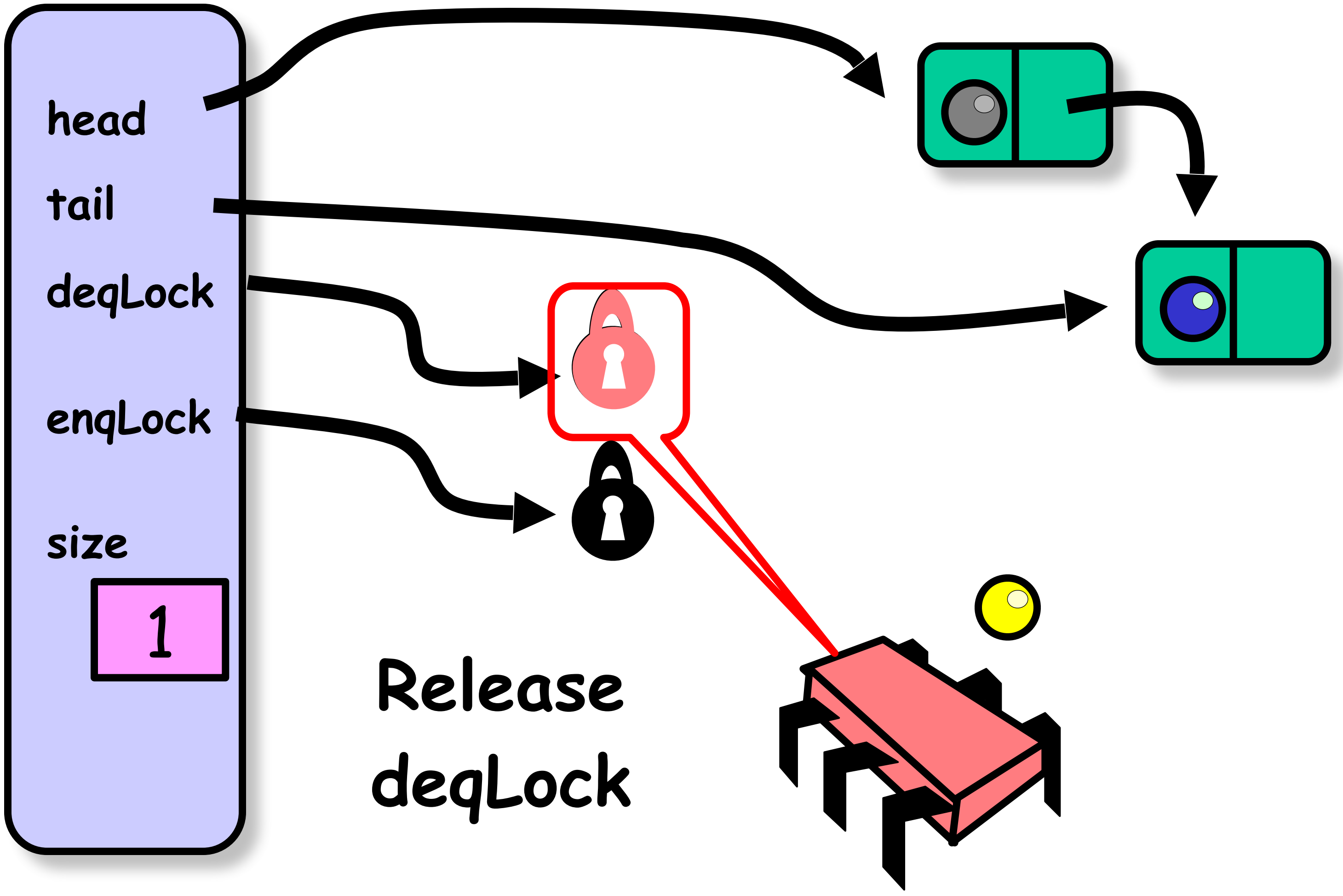
Make first Node new sentinel



Dequeuer



Dequeuer

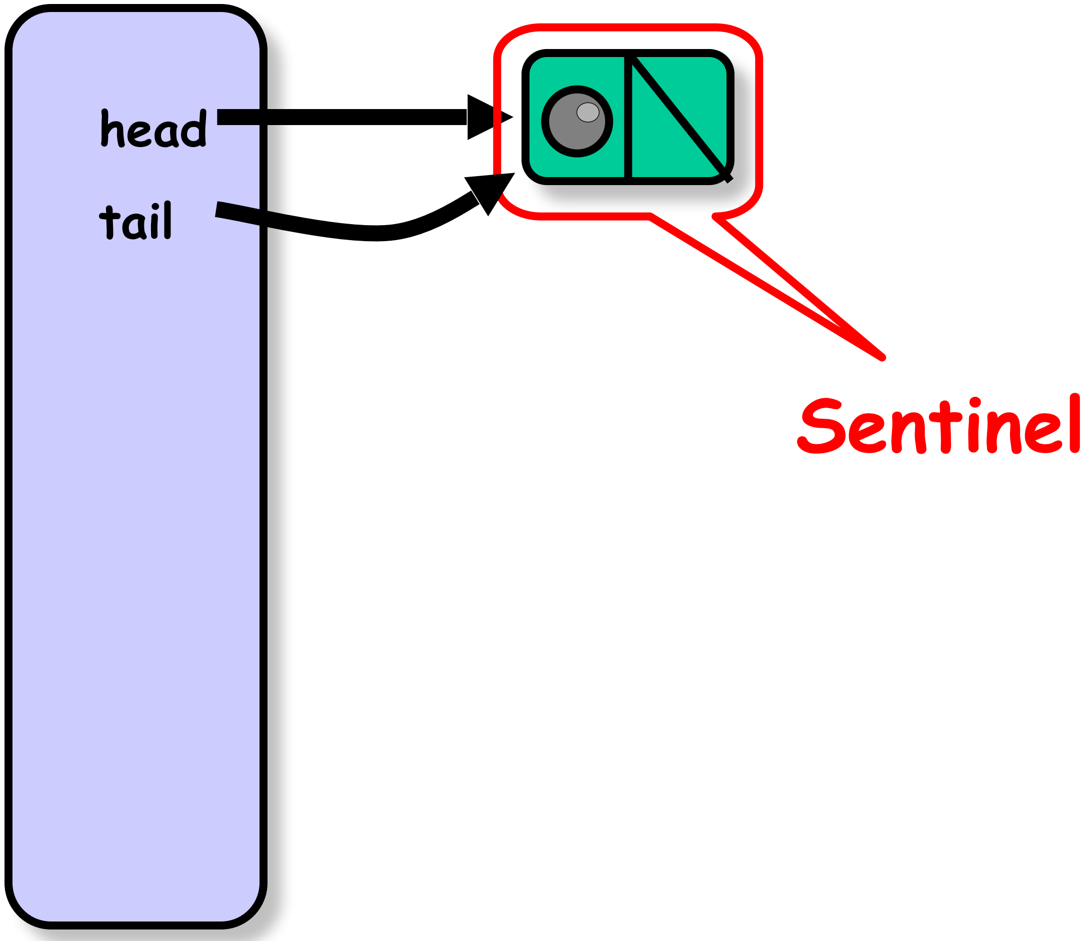


Release deqLock

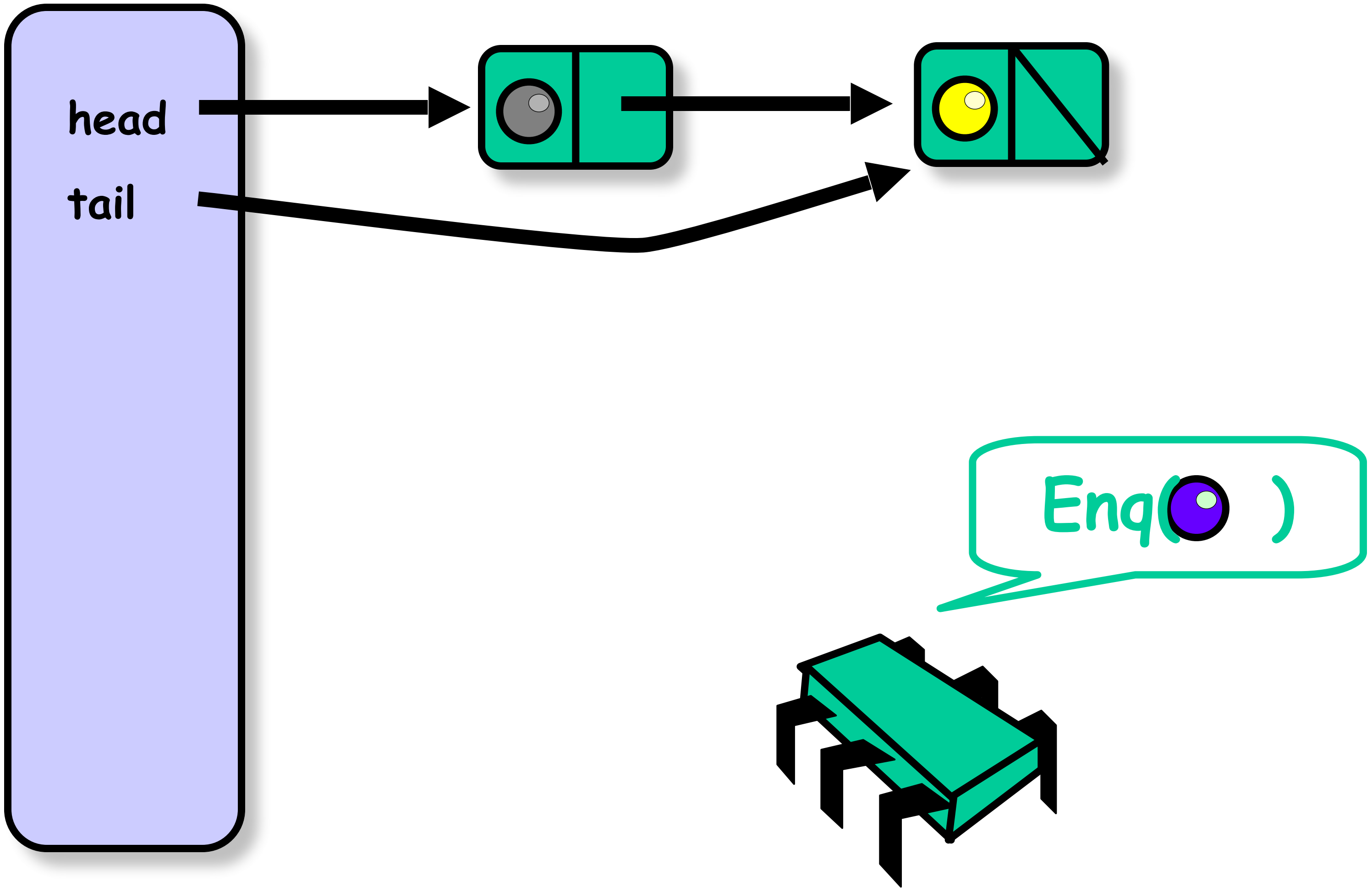
Unbounded Lock-Free Queue (Nonblocking)

- Unbounded
  - No need to count the number of items
- Lock-free
  - Use AtomicReference<V>
    - An object reference that may be updated atomically.
  - boolean compareAndSet(V expect, V update)
    - Atomically sets the value to the given updated value if the current value == the expected value.
- Nonblocking
  - No need to provide conditions on which to wait

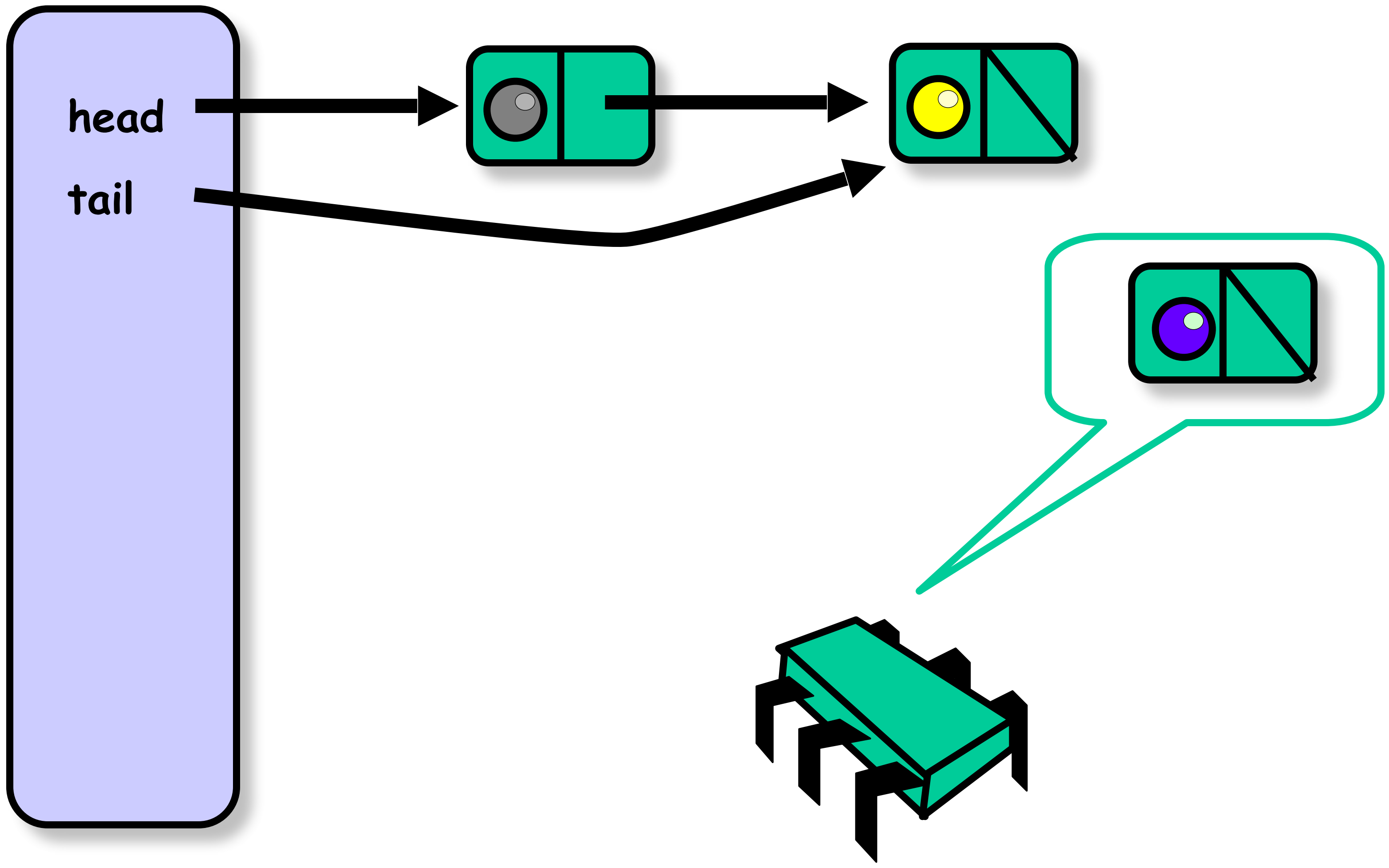
A Lock-Free Queue



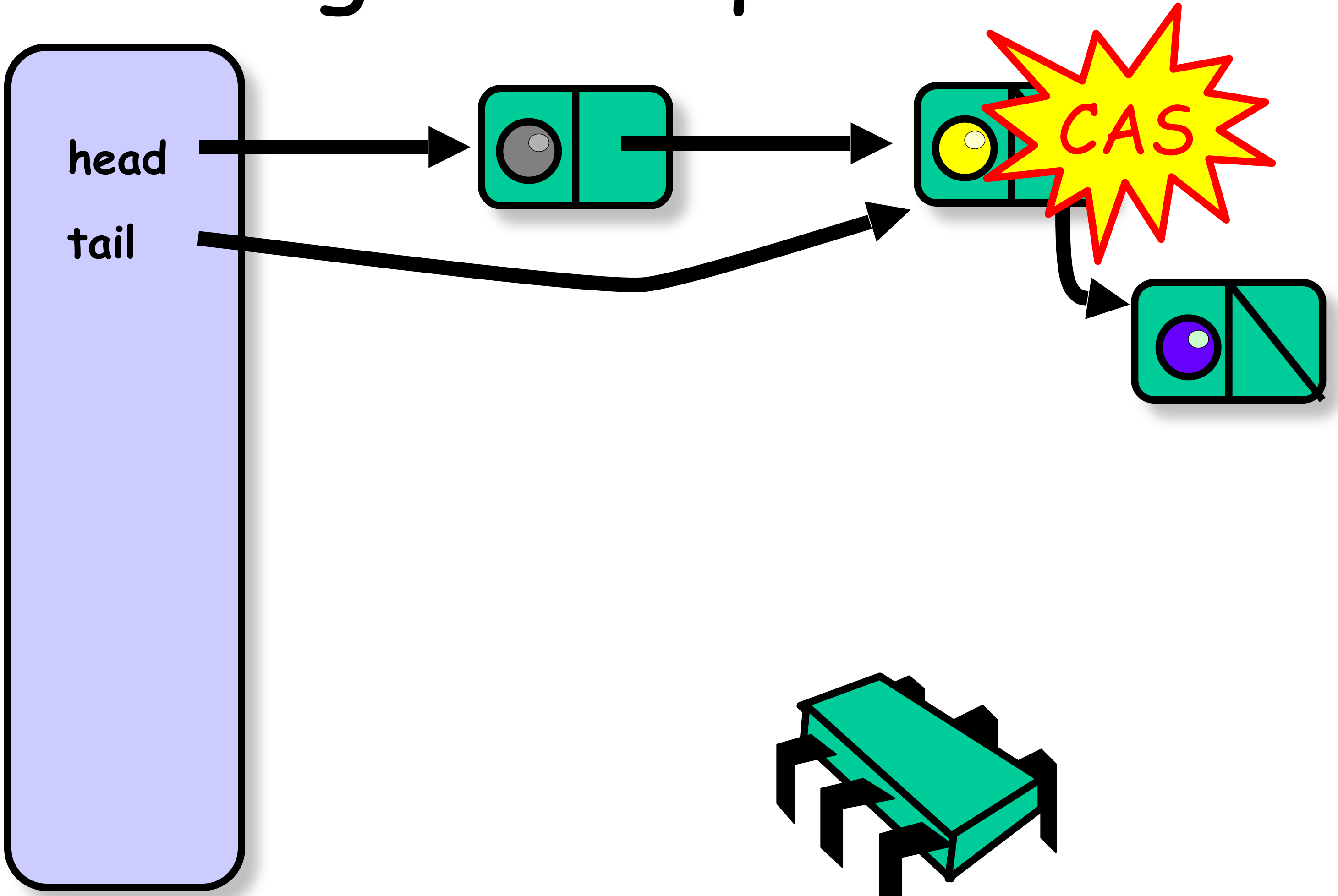
Enqueue

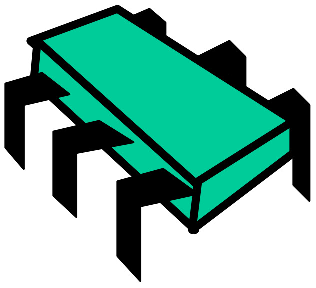


Enqueue

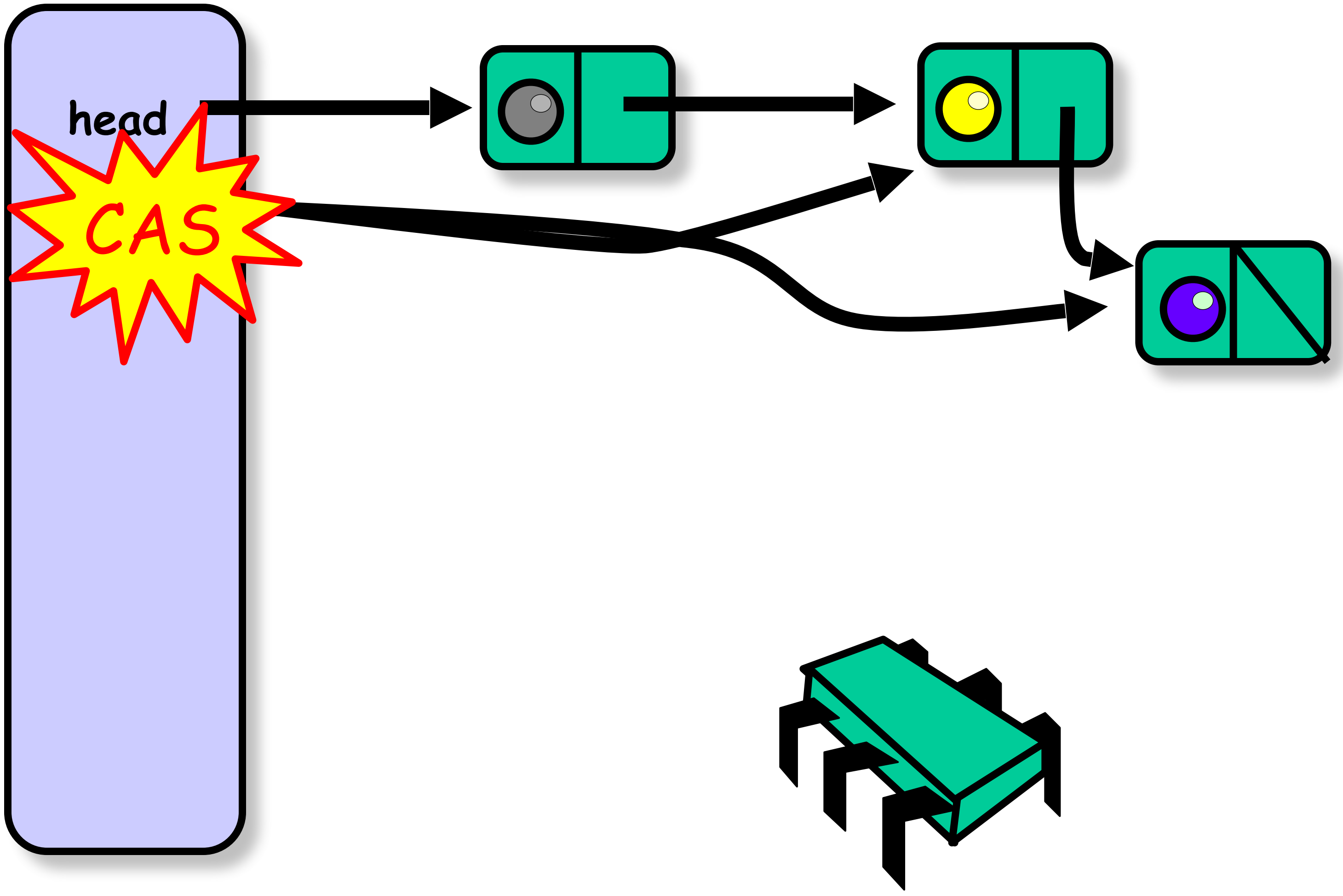


Logical Enqueue





Physical Enqueue



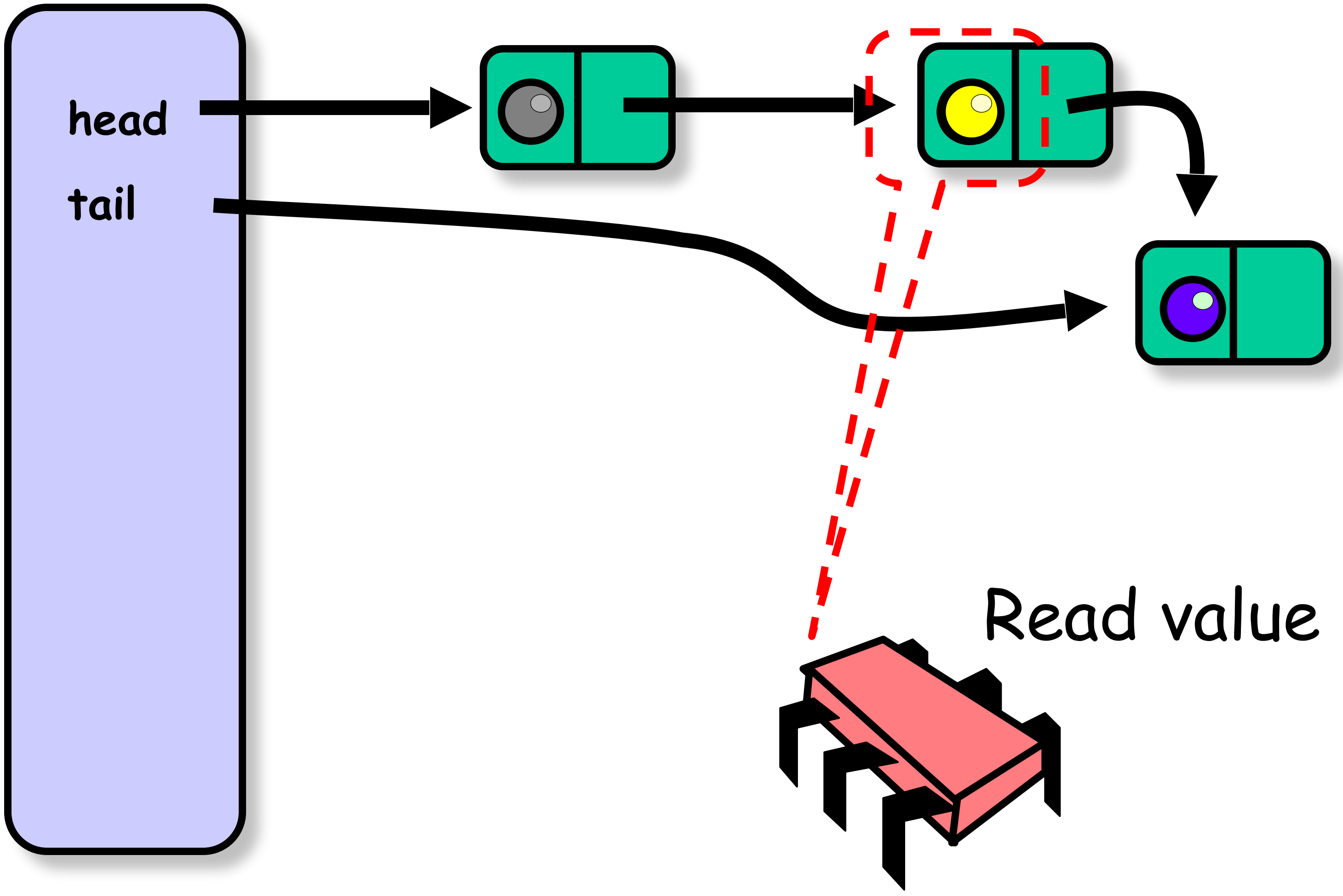
Enqueue

- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good)
  - Penultimate Node (not so good)
- Be prepared!
- (For you to think about) How could you fix that?

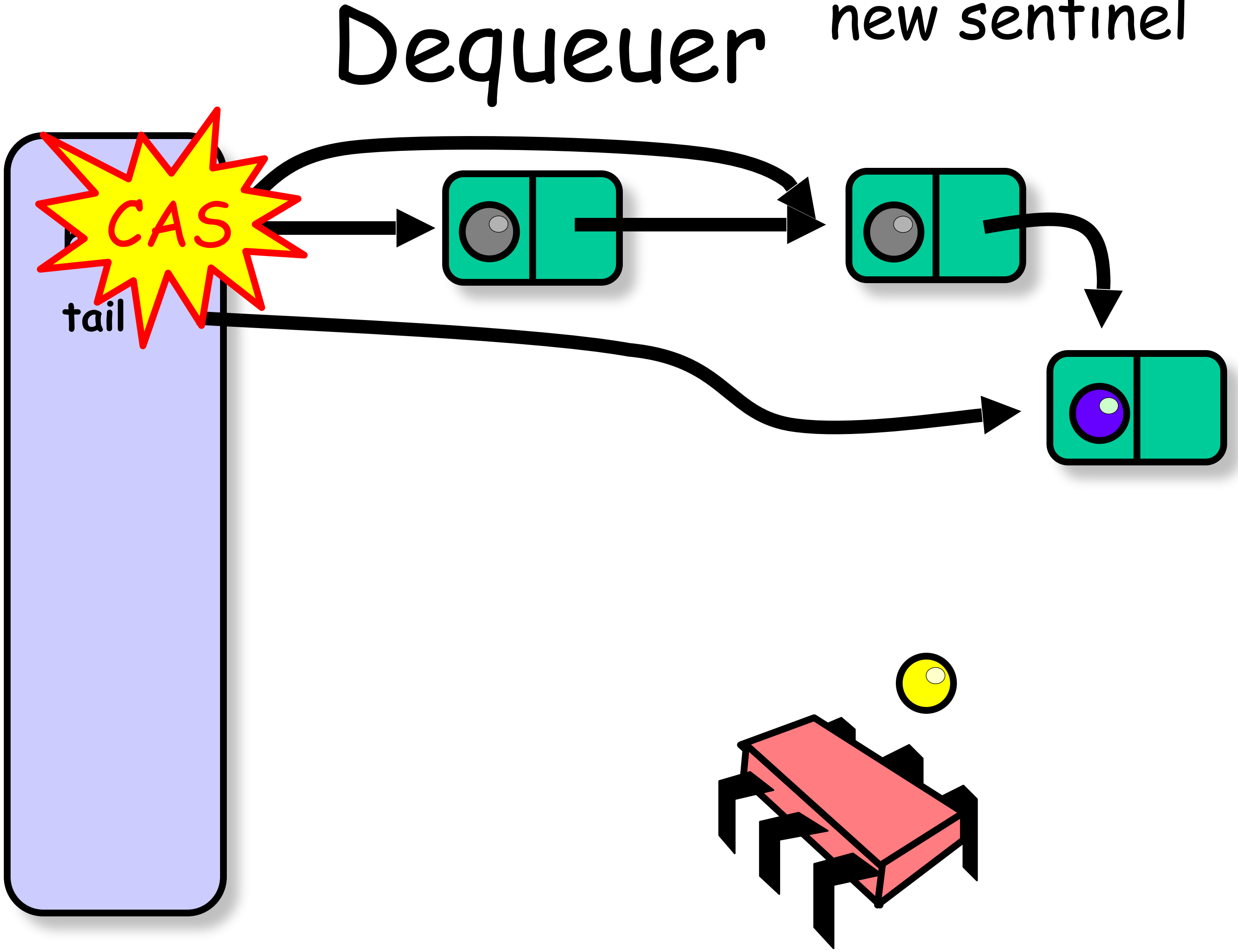
When CASs Fail

- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

Dequeuer



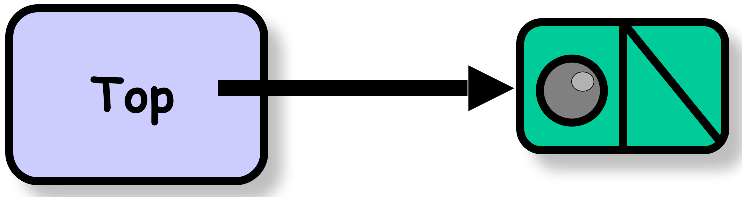
Make first Node new sentinel



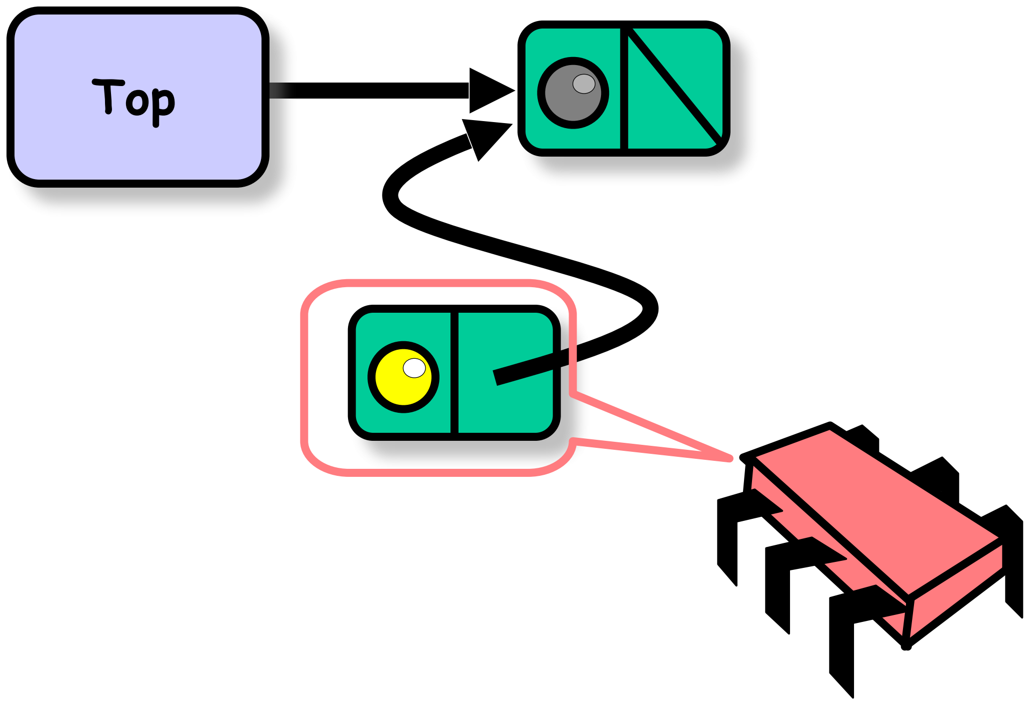
Concurrent Stack

- Methods
  - **push(x)**
  - **pop()**
- Last-in, First-out (LIFO) order
- Lock-Free!

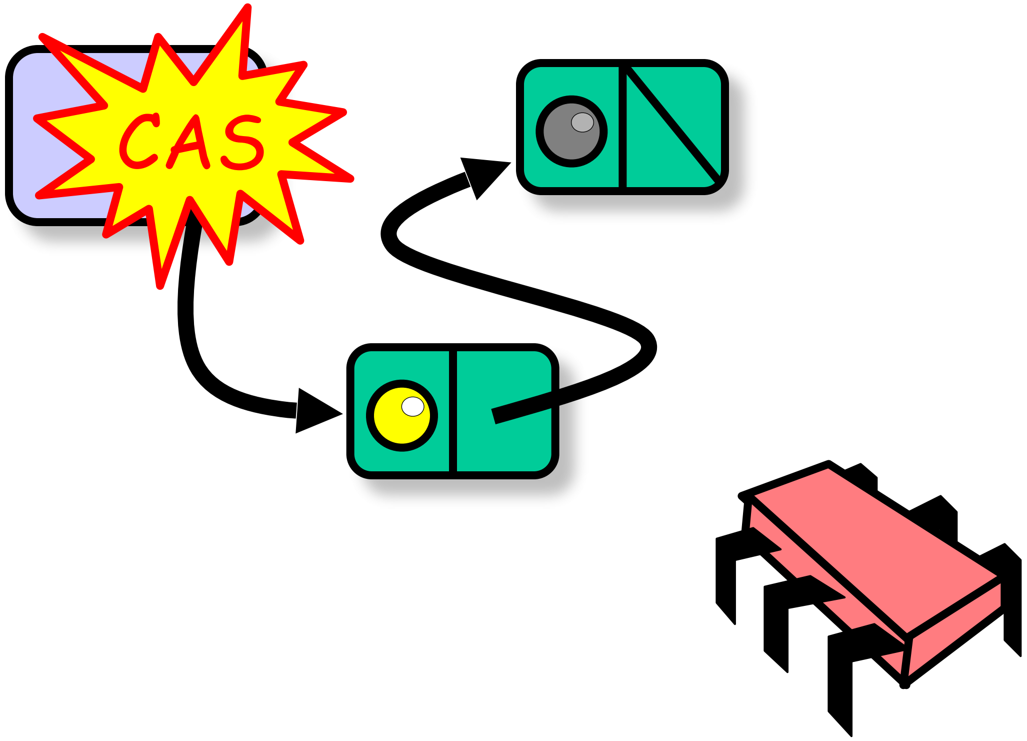
Empty Stack



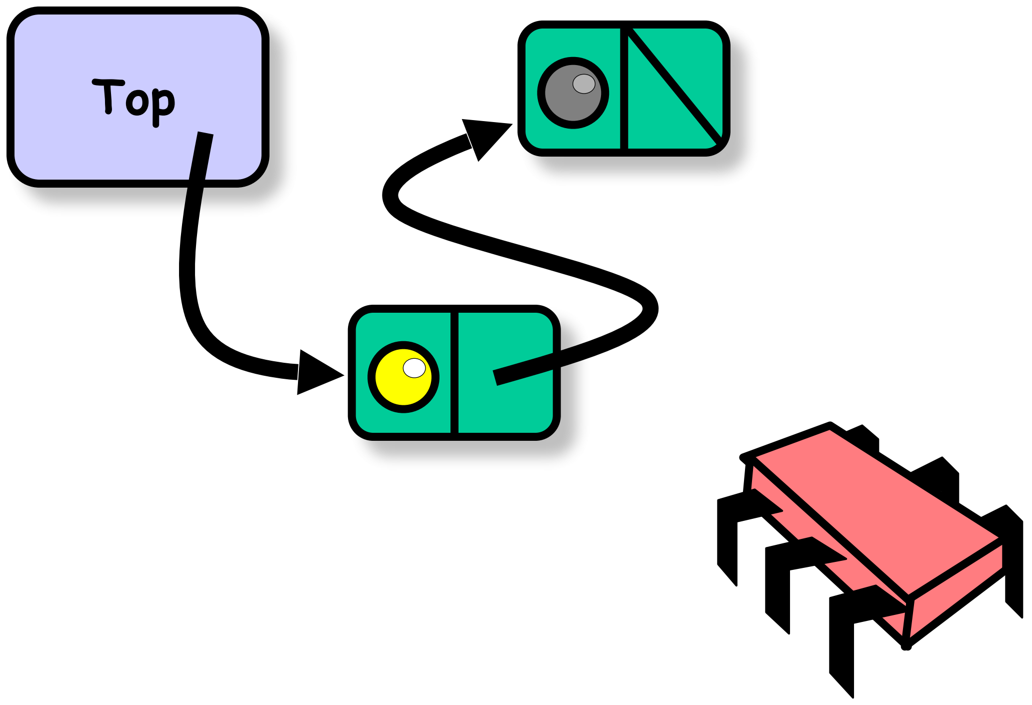
Push



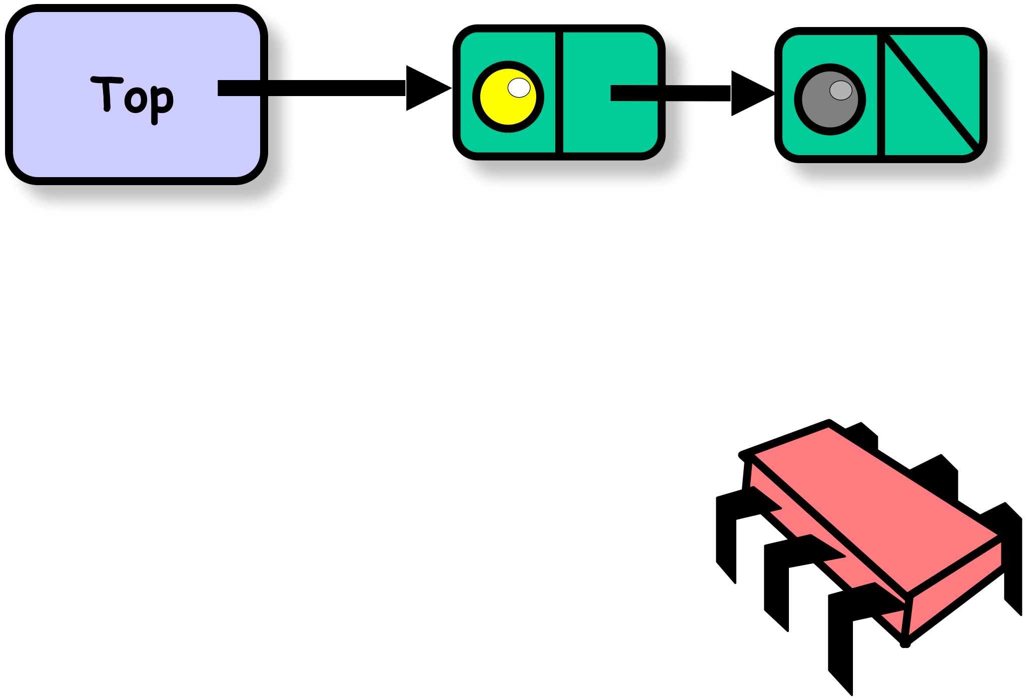
Push



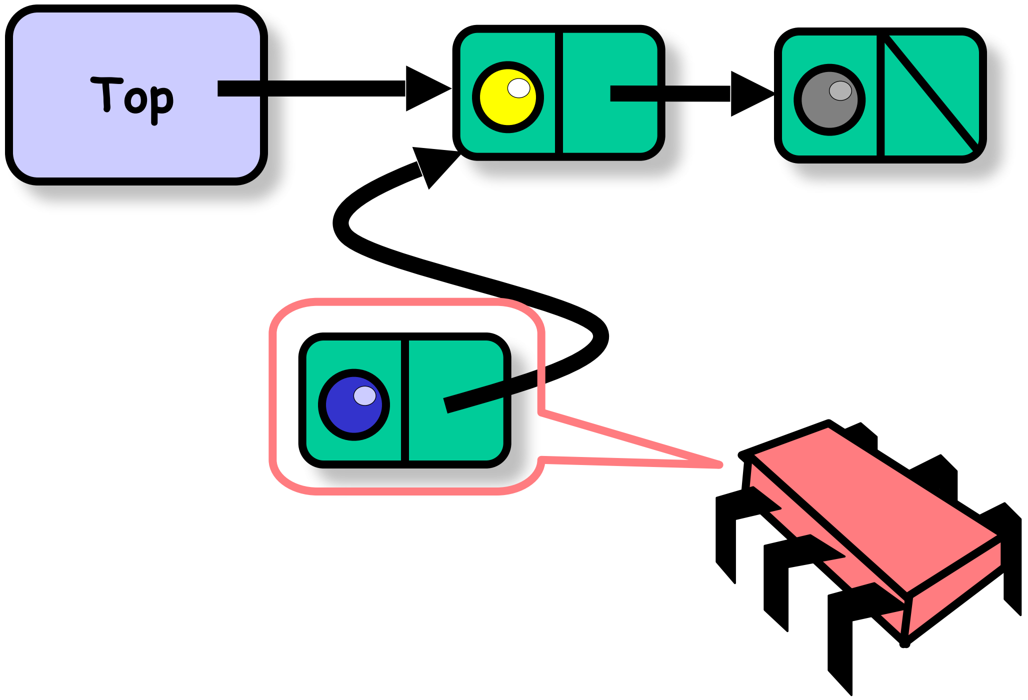
Push



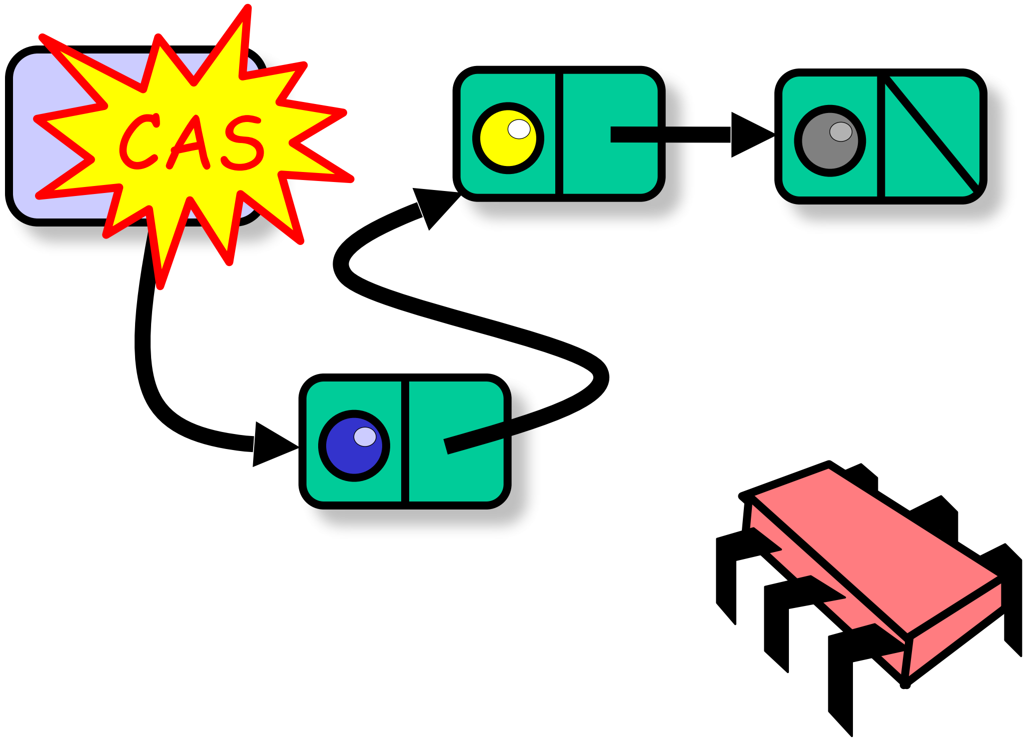
Push



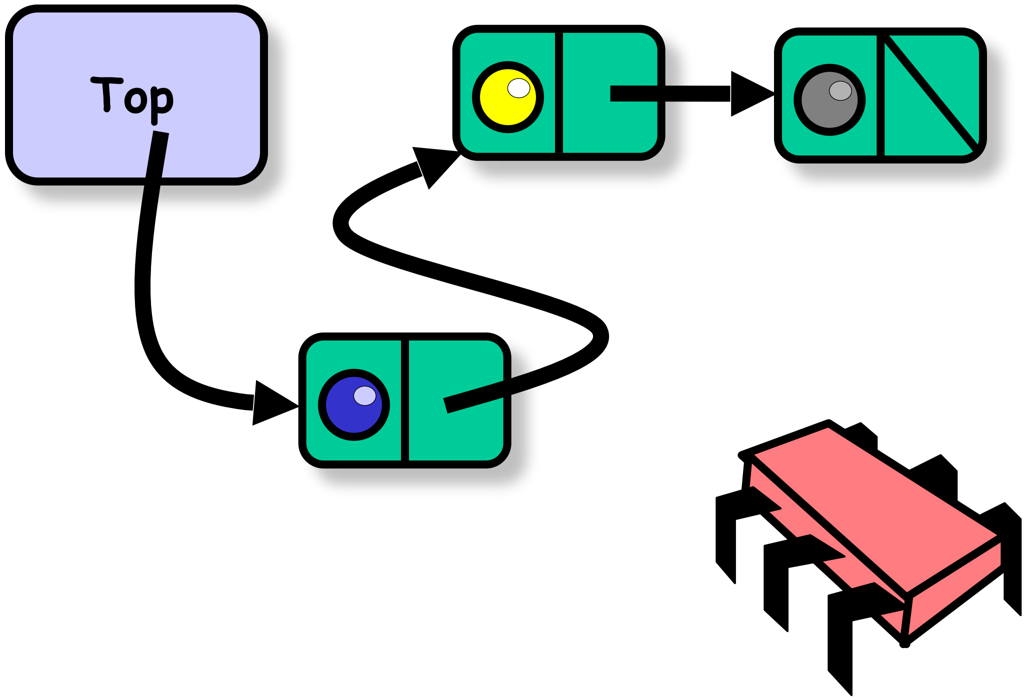
Push



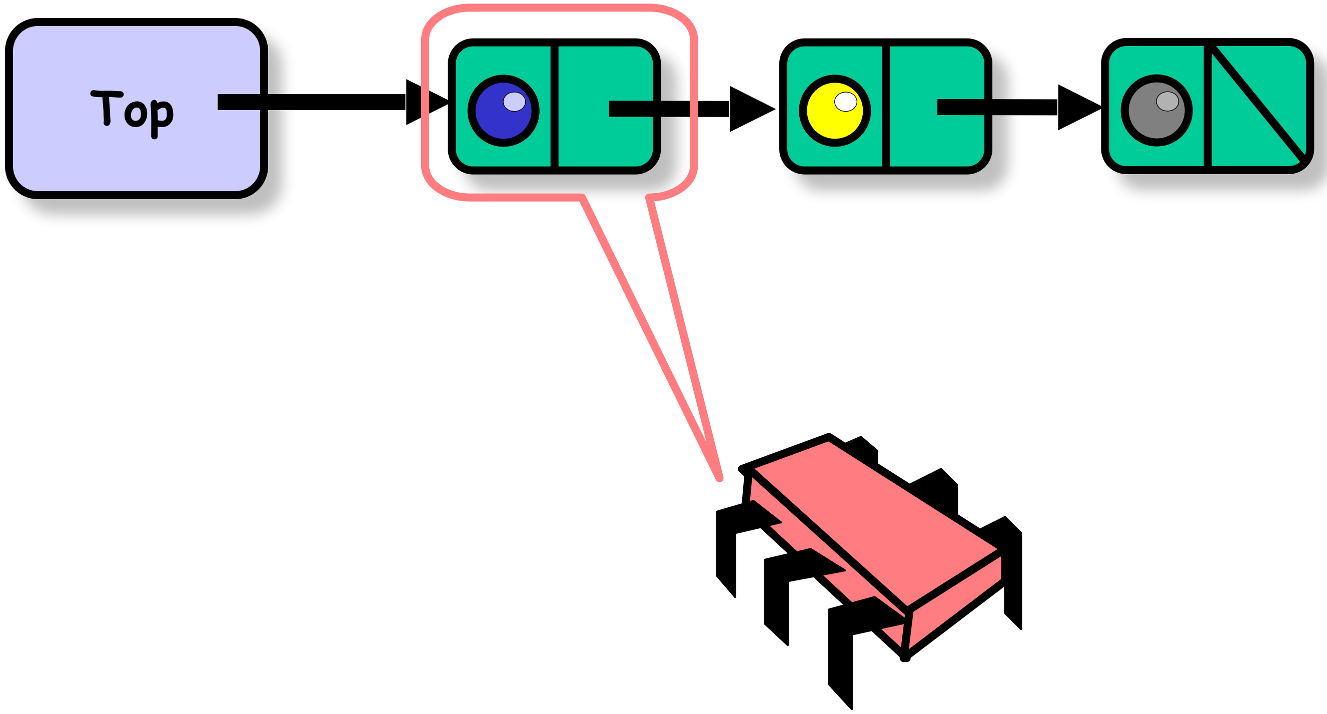
Push



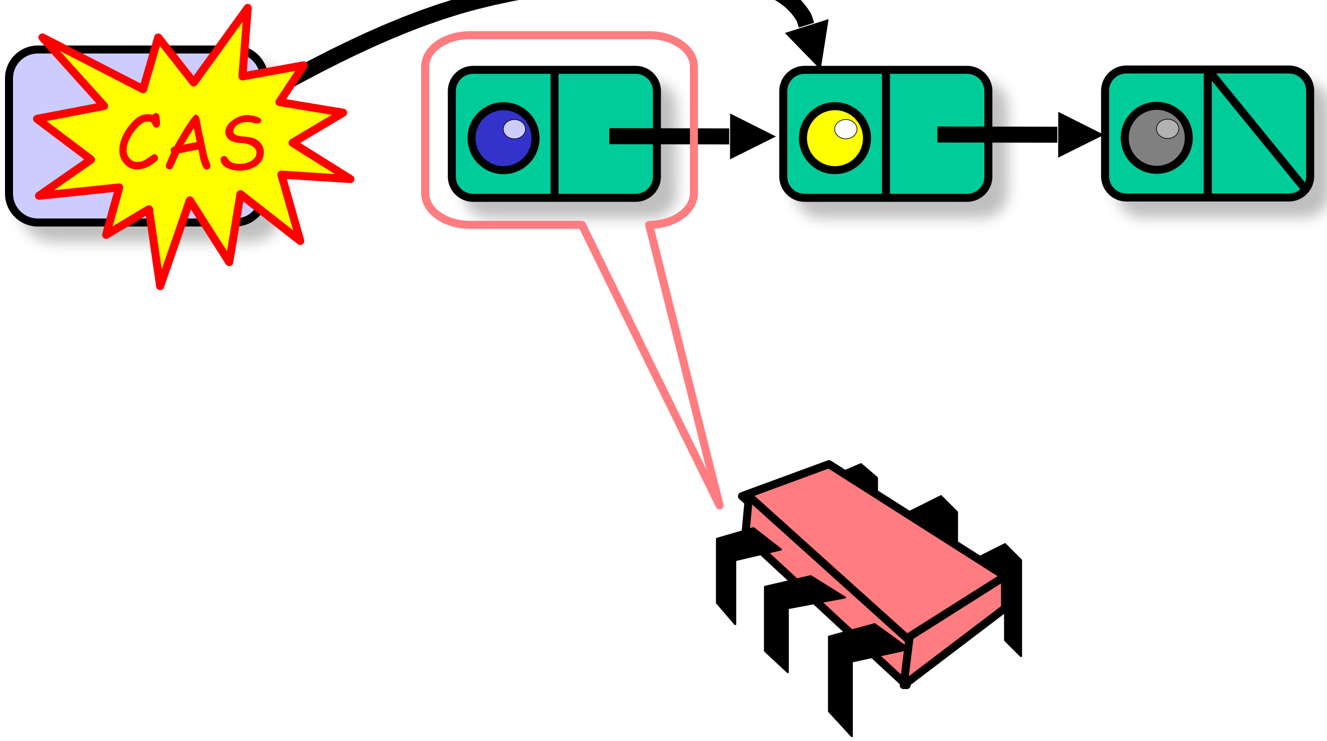
Push



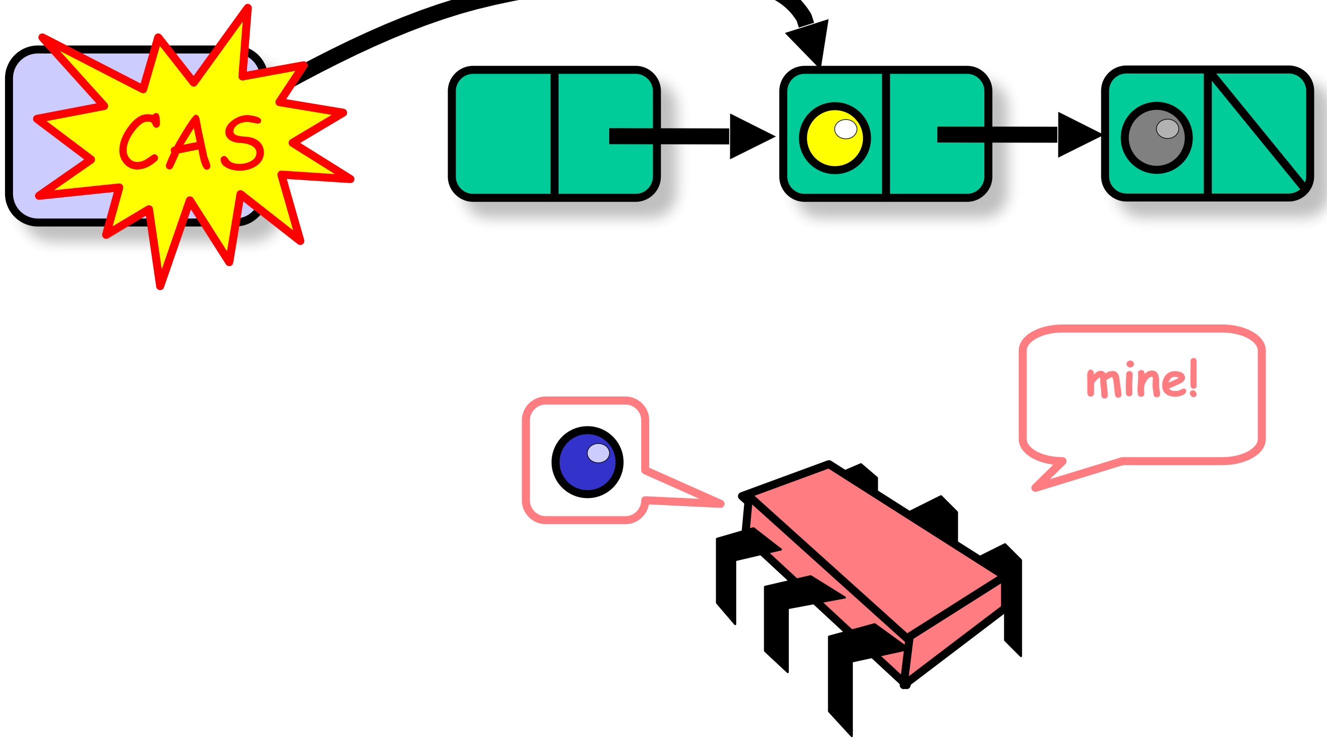
Pop



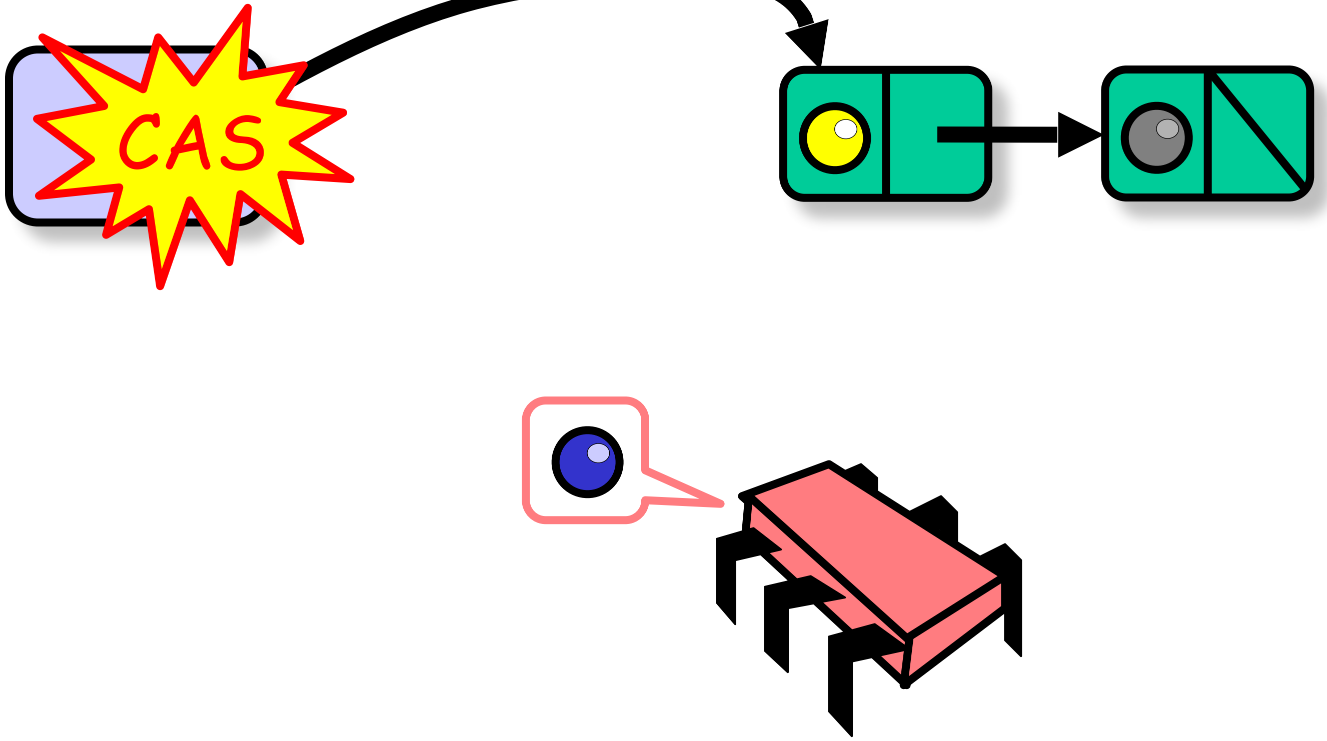
Pop



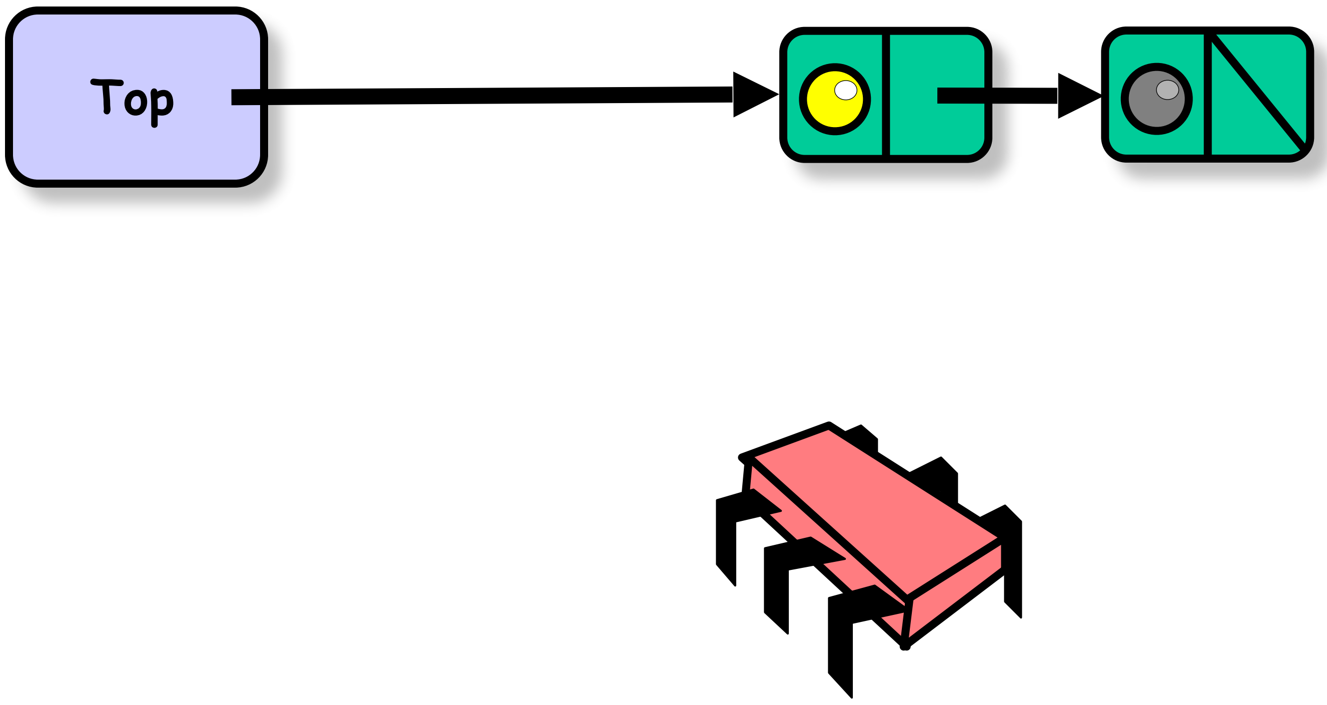
Pop



Pop



Pop



Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top =
    new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
  Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff();
  }}
```

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public Boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

tryPush attempts to push a node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Read top value

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

current top will be new node's successor

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Try to swing top, return success or failure

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Push calls tryPush

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

Create new node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = o
return(top.compareAndSet(oldTop, node))
    }
public void push(T value) {
  Node node = new Node(value);
  while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff()
}}
```

If tryPush() fails,
back off before retrying

Unbounded Lock-Free Stack

```
protected boolean tryPush(Node node)
{
  Node oldTop = top.get();
  node.next = oldTop;
  return (top.compareAndSet(oldTop, node));
}

public void push( T value )
{
  Node node = new Node( value );
  while (true) {
    if (tryPush(node)) { return; }
    else { backoff.backoff( ); }
  }
}
```

```
protected Node tryPop( ) throws EmptyException
{
  Node oldTop = top.get();
  if ( oldTop == null ) {
    throw new EmptyException( );
  }
  Node newTop = oldTop.next;
  if ( top.compareAndSet( oldTop, newTop ) ) {
    return oldTop;
  } else { return null; }
}

public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop( );
    if ( returnNode != null ) {
      return returnNode.value;
    } else { backoff.backoff( ); }
  }
}
```

Lock-free Stack

- Good
  - No locking
- Bad
  - Without GC, fear ABA
  - Without backoff, huge contention at top
  - In any case, no parallelism

Question

- Are stacks inherently sequential?
- Reasons why
  - Every **pop()** call fights for top item
- Reasons why not
  - Think about it!