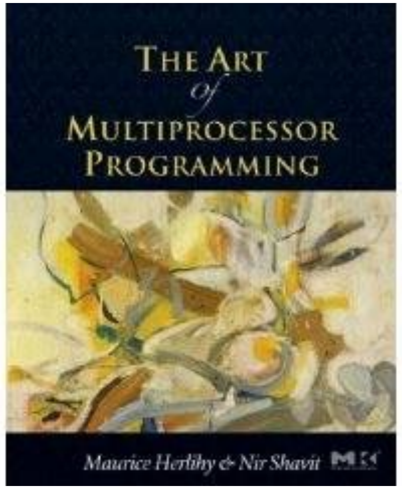
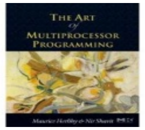
Linked Lists: Locking, Lock-Free, and Beyond …


THE ART of MULTIPROCESSOR PROGRAMMING
Maurice Herlihy & Nir Shavit

Hyungsoo Jung

Recall Three Design Patterns

- Fine-grained synchronization
- Optimistic Synchronization
- Lazy Synchronization

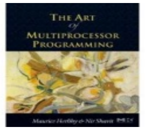


First:
Fine-Grained Synchronization

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

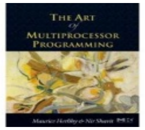
Second:
Optimistic Synchronization

- 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

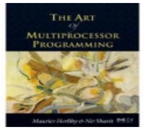
- Don’t use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

Reminder: Lock-Free Data Structures

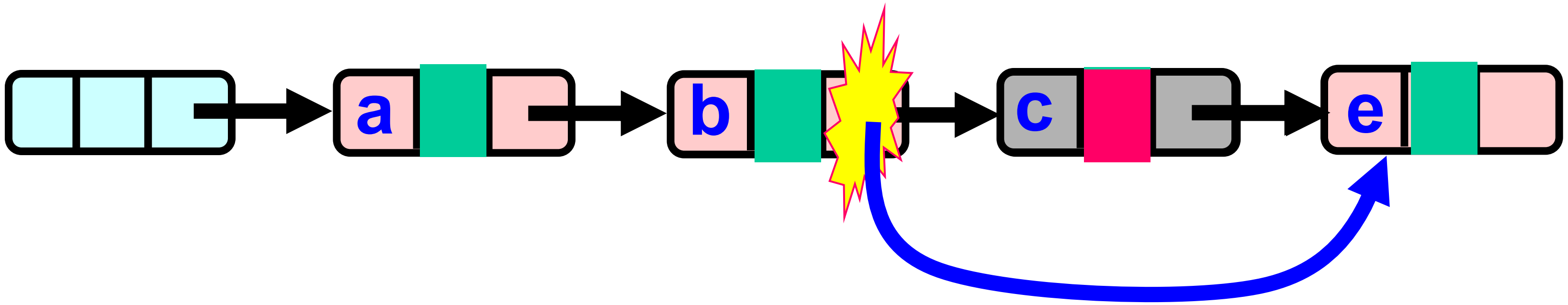
- No matter what …
  - – Guarantees minimal progress in any execution
  - – i.e. Some thread will always complete a method call
  - – Even if others halt at malicious times
  - – Implies that implementation can’t use locks

Lock-free Lists

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



Lock-free Lists

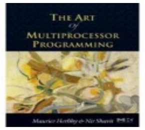


Logical Removal

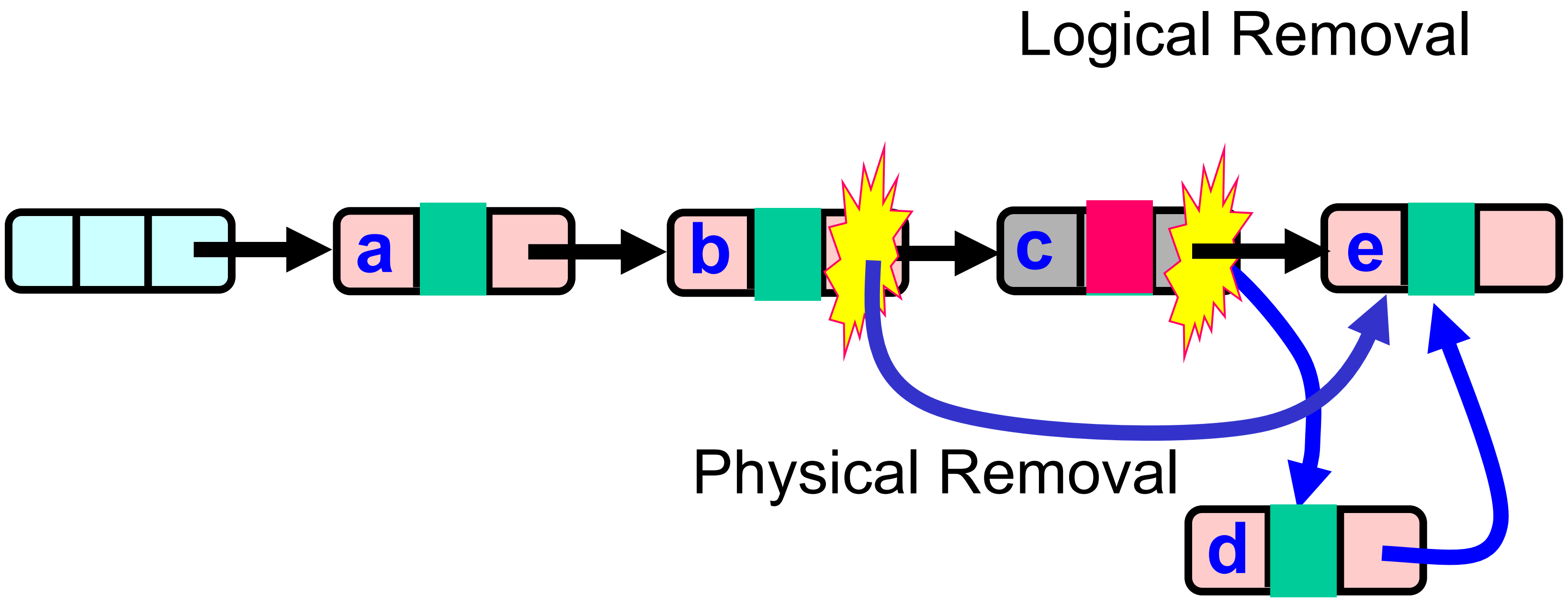
Physical Removal

Use CAS to verify pointer is correct

Not enough!



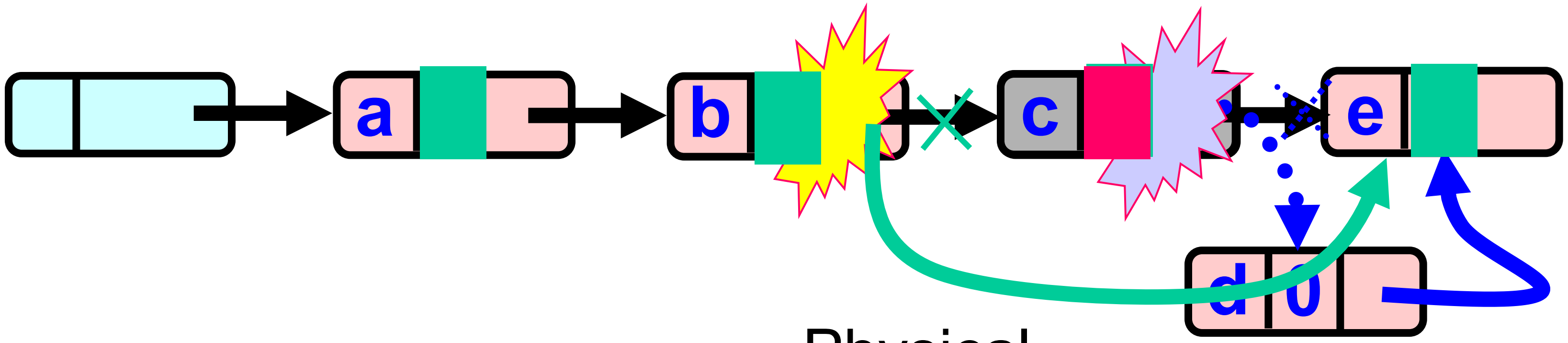
Problem…



Node added

The Solution: Combine Bit and Pointer

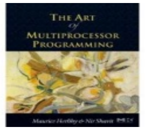
Logical Removal = Set Mark Bit



Physical Removal CAS

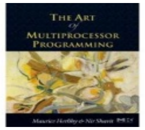
Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)



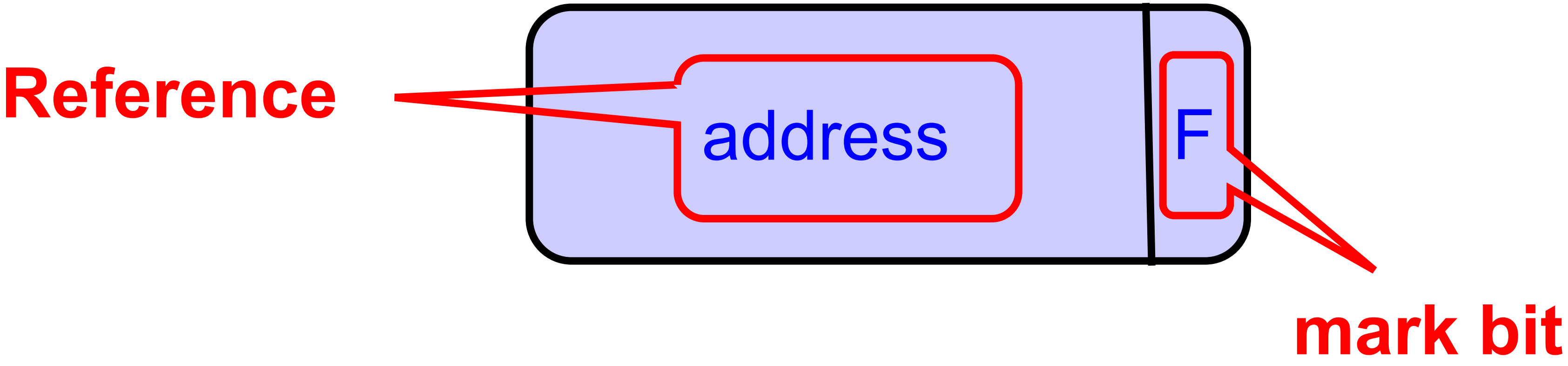
Solution

- 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

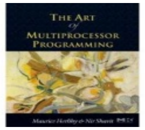
```
Public Object get(boolean[] marked);
```

Extracting Reference & Mark

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

Returns reference

Returns mark at array index 0!



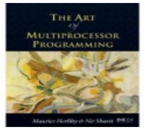
Extracting Mark Only

```
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);
```

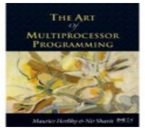
And this is the current mark …

Changing State

…then change to this new reference …

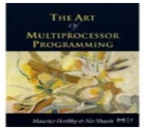
```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

… and this new mark



Changing State

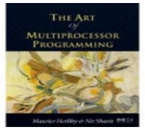
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```



Changing State

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

If this is the current reference …

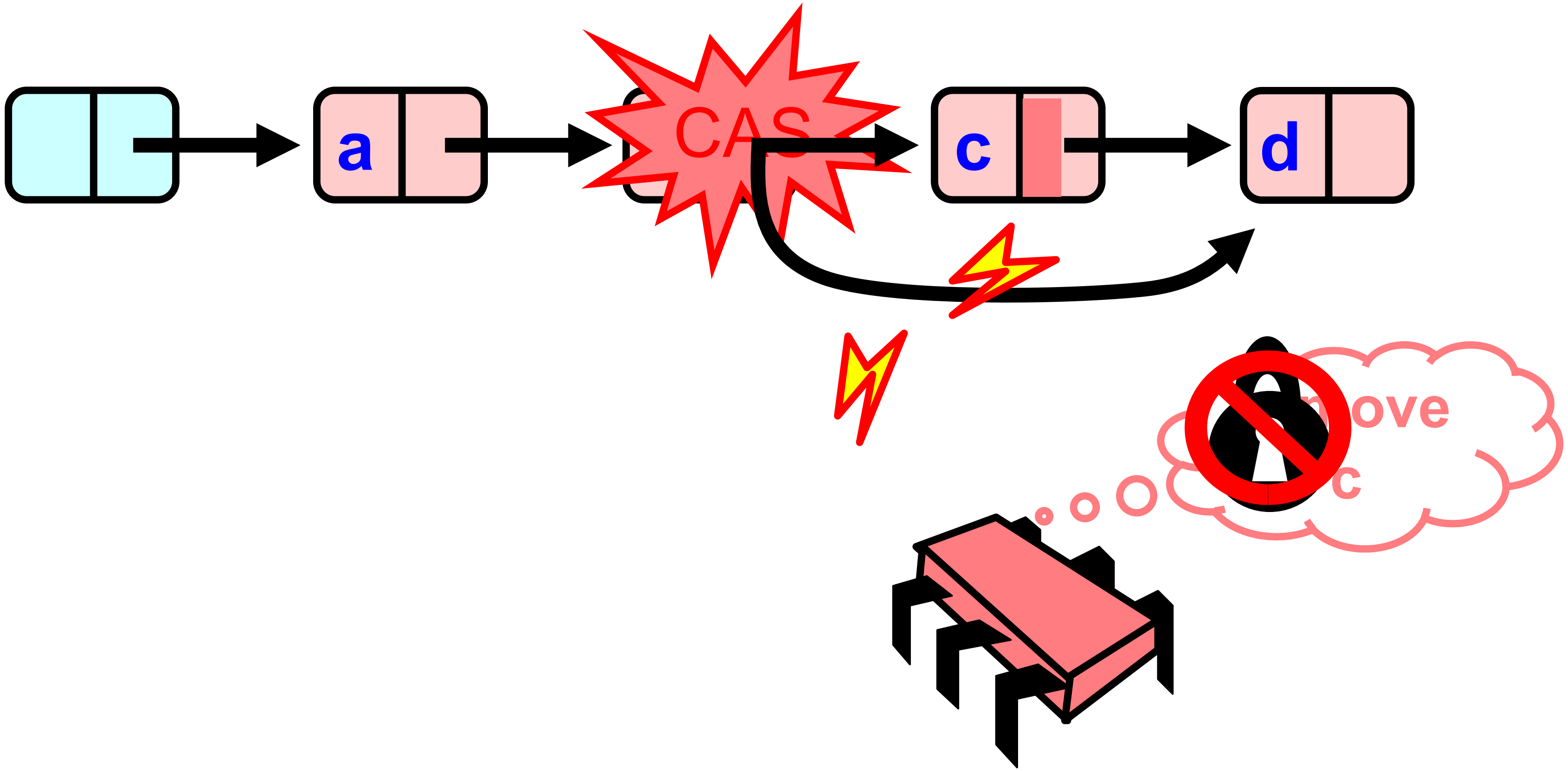


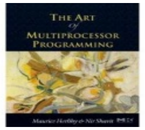
Changing State

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

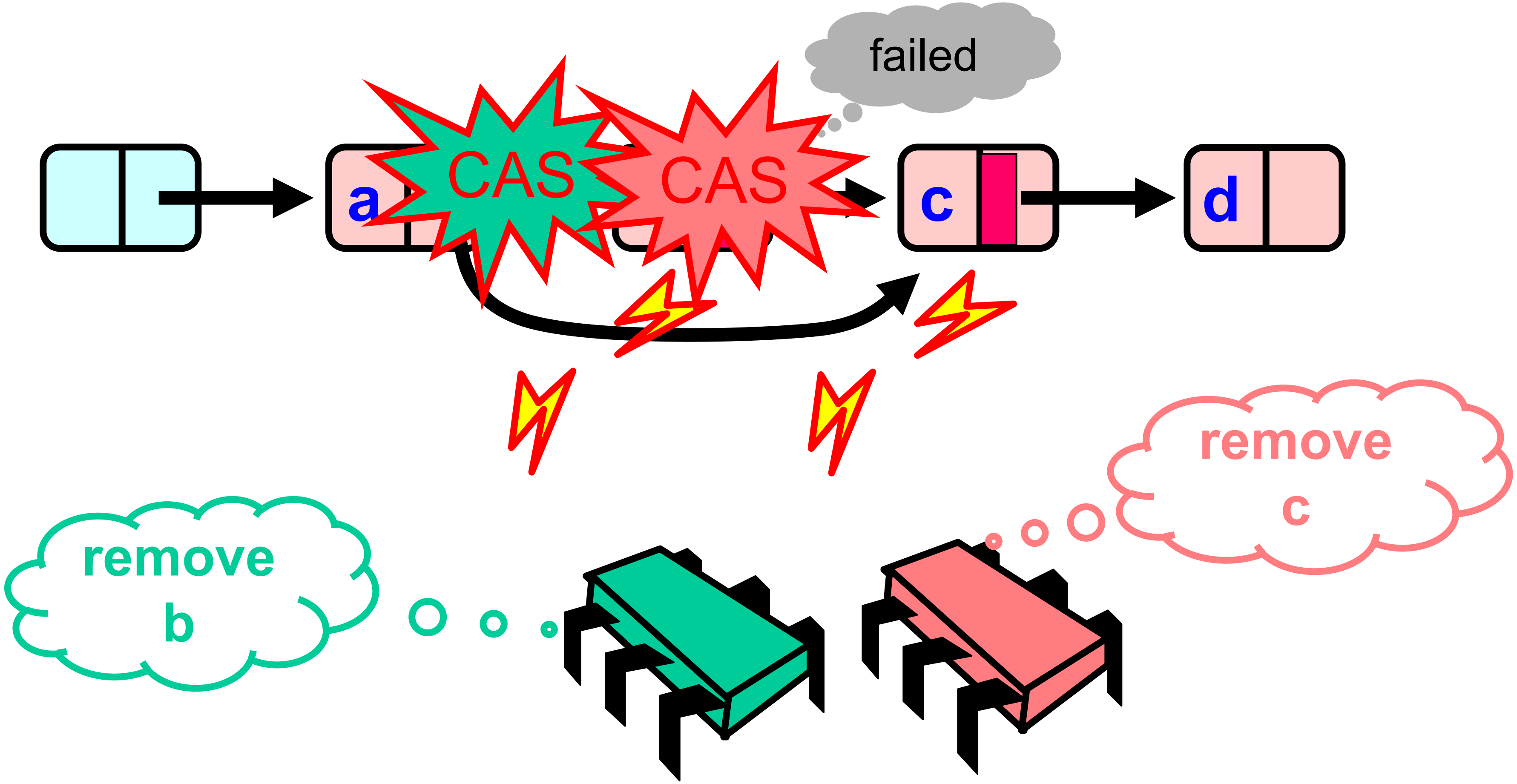
.. then change to
this new mark.

Removing a Node

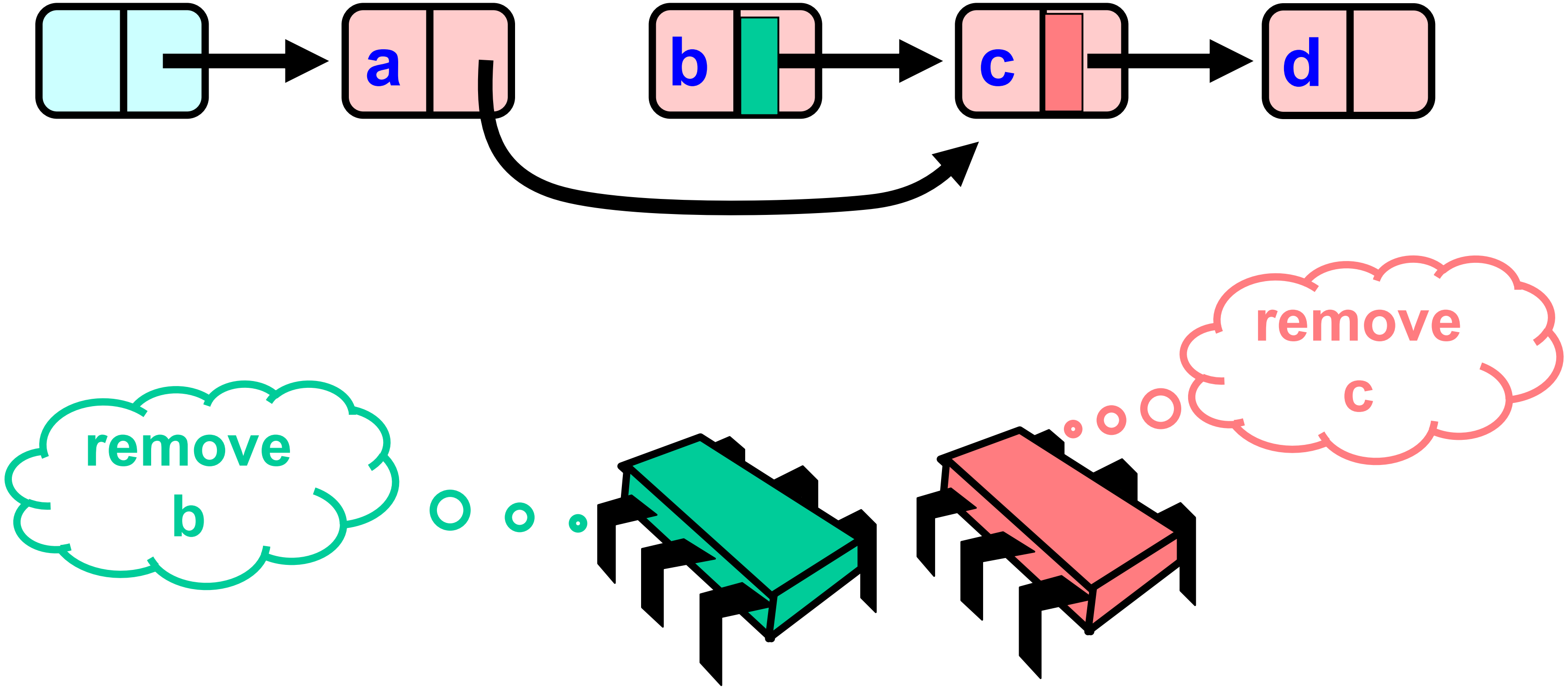


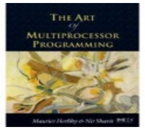


Removing a Node

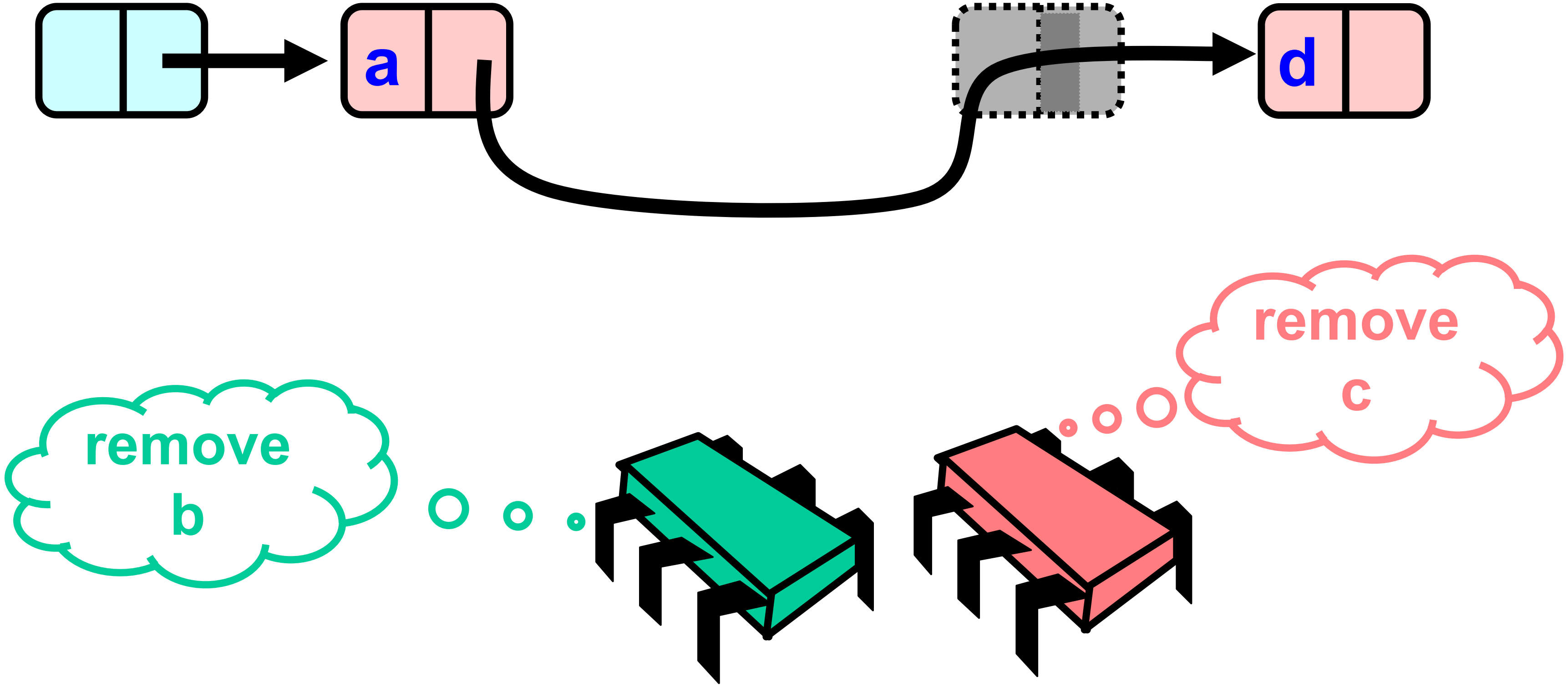


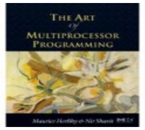
Removing a Node





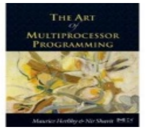
Removing a Node



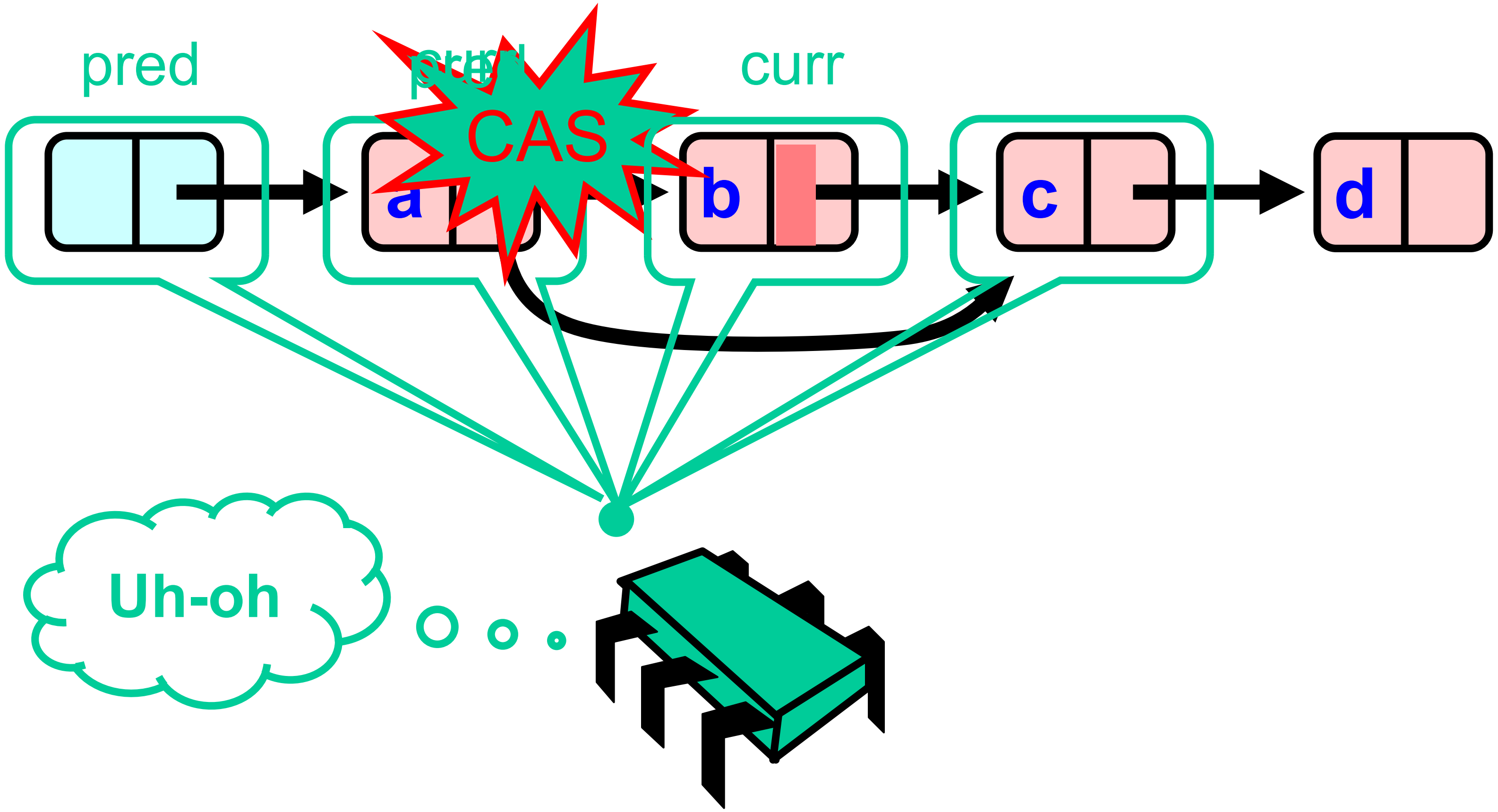


Traversing the List

- 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)

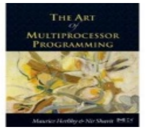


Lock-Free Traversal
(only Add and Remove)



The Window Class

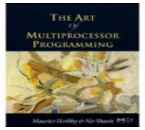
```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```



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

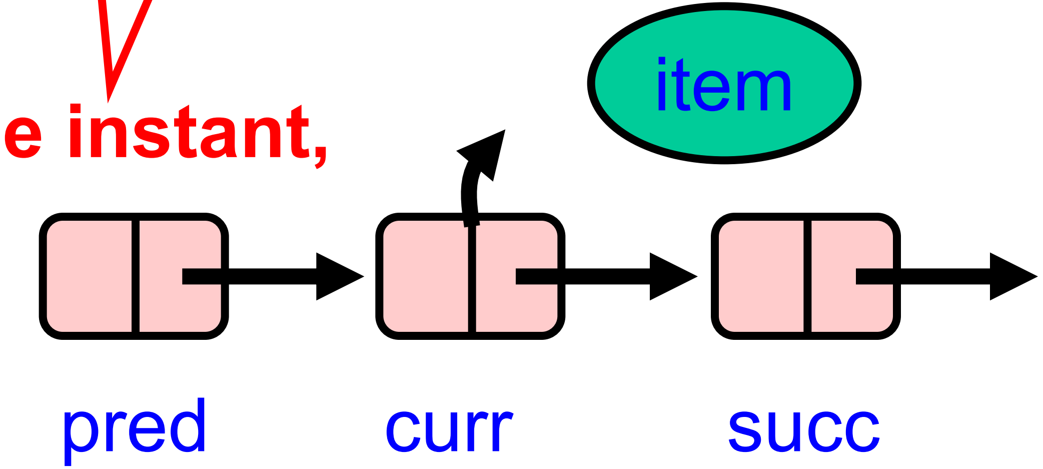
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Extract pred and curr

The Find Method

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

At some instant,

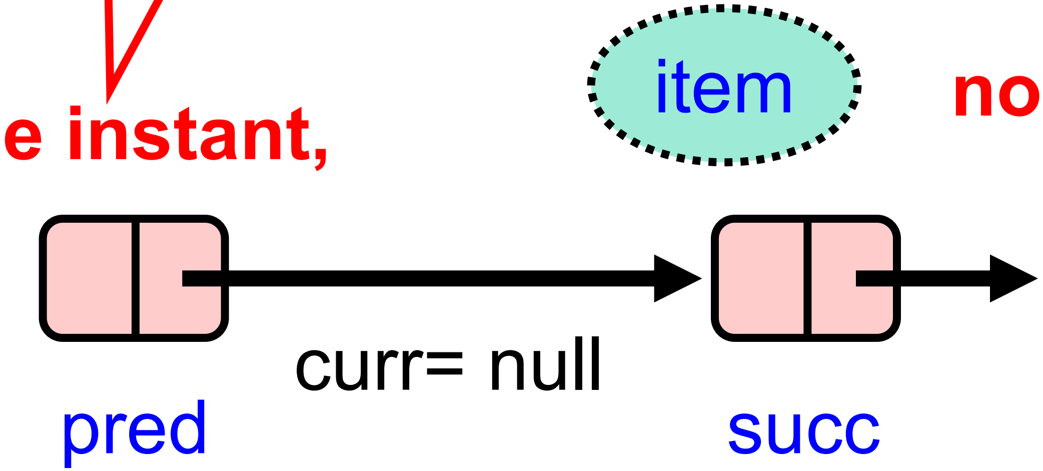


or …

The Find Method

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

At some instant,



item

not in list

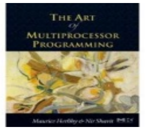
curr= null

pred

succ

Remove

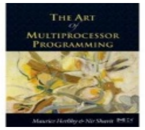
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if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

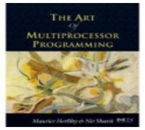
Keep trying



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;
}}}
```

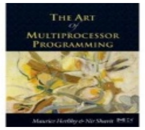
Find neighbors



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;
}}}
```

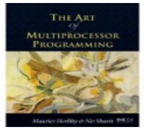
She’s not there …



Remove

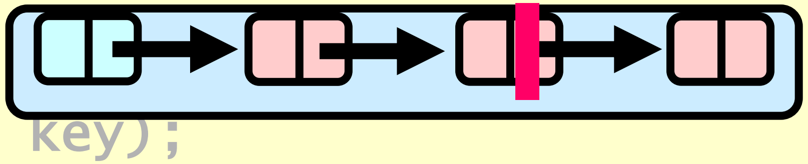
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
 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

Try to mark node as deleted



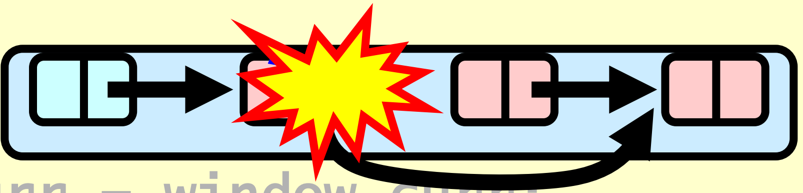
Remove

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



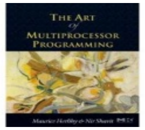
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
 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

Remove



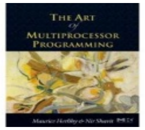
```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head,
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
    return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip)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



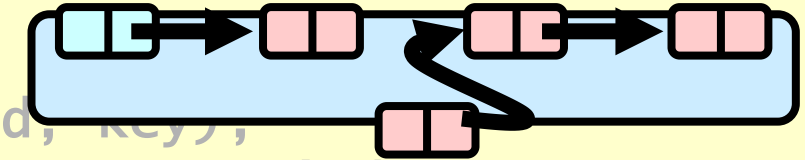
Add

```
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;}
}}}
```

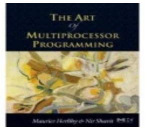
Item already there.

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;}
}}}
```

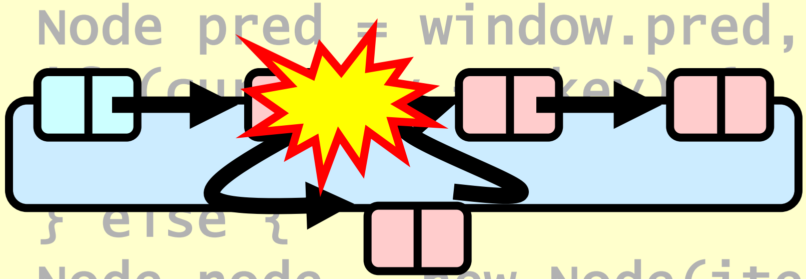


create new node

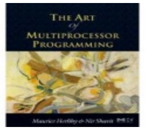


Add

```
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;}
}}}
```

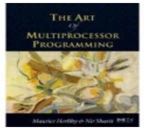


Install new node, else retry loop



Wait-free Contains

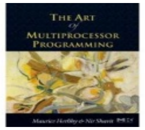
```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```



Wait-free Contains

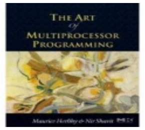
```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Only diff is that we get and check marked



Lock-free Find

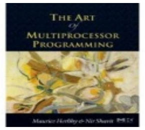
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if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

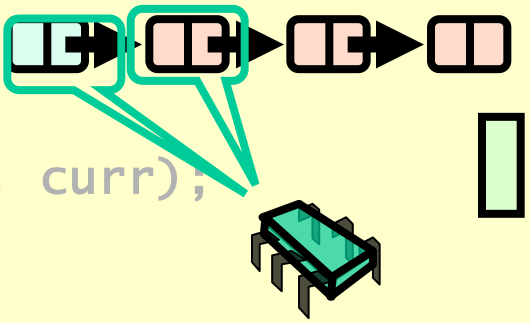
If list changes while traversed, start over



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, 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;
    }
}}
```

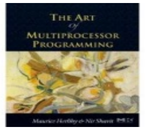
Start looking from head



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

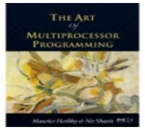
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if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

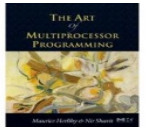
Try to remove deleted nodes in path…code details soon



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;
     }
}}
```

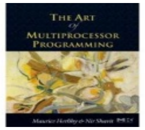
If curr key that is greater or equal, return pred and curr



Lock-free Find

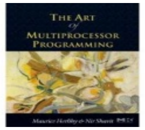
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
     …
    }
   if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

Otherwise advance window and loop again



Lock-free Find

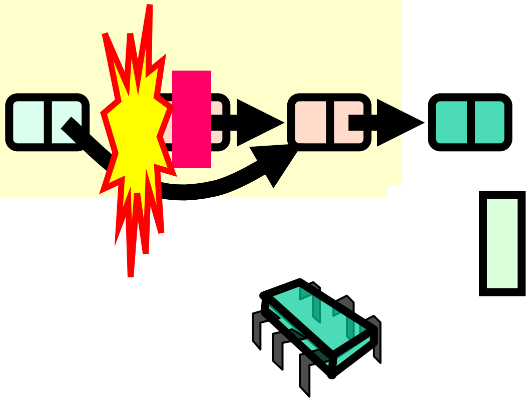
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                    succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

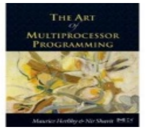


Lock-free Find

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);
   }
…
```

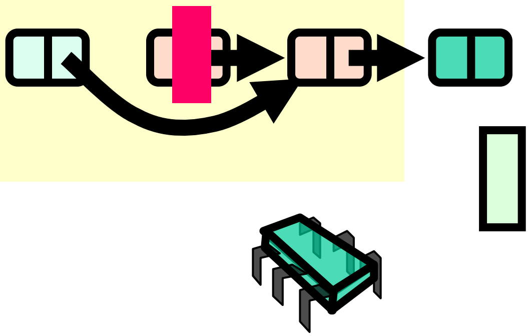


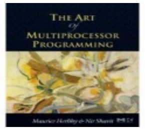


Lock-free Find

if predecessor's next field changed must 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

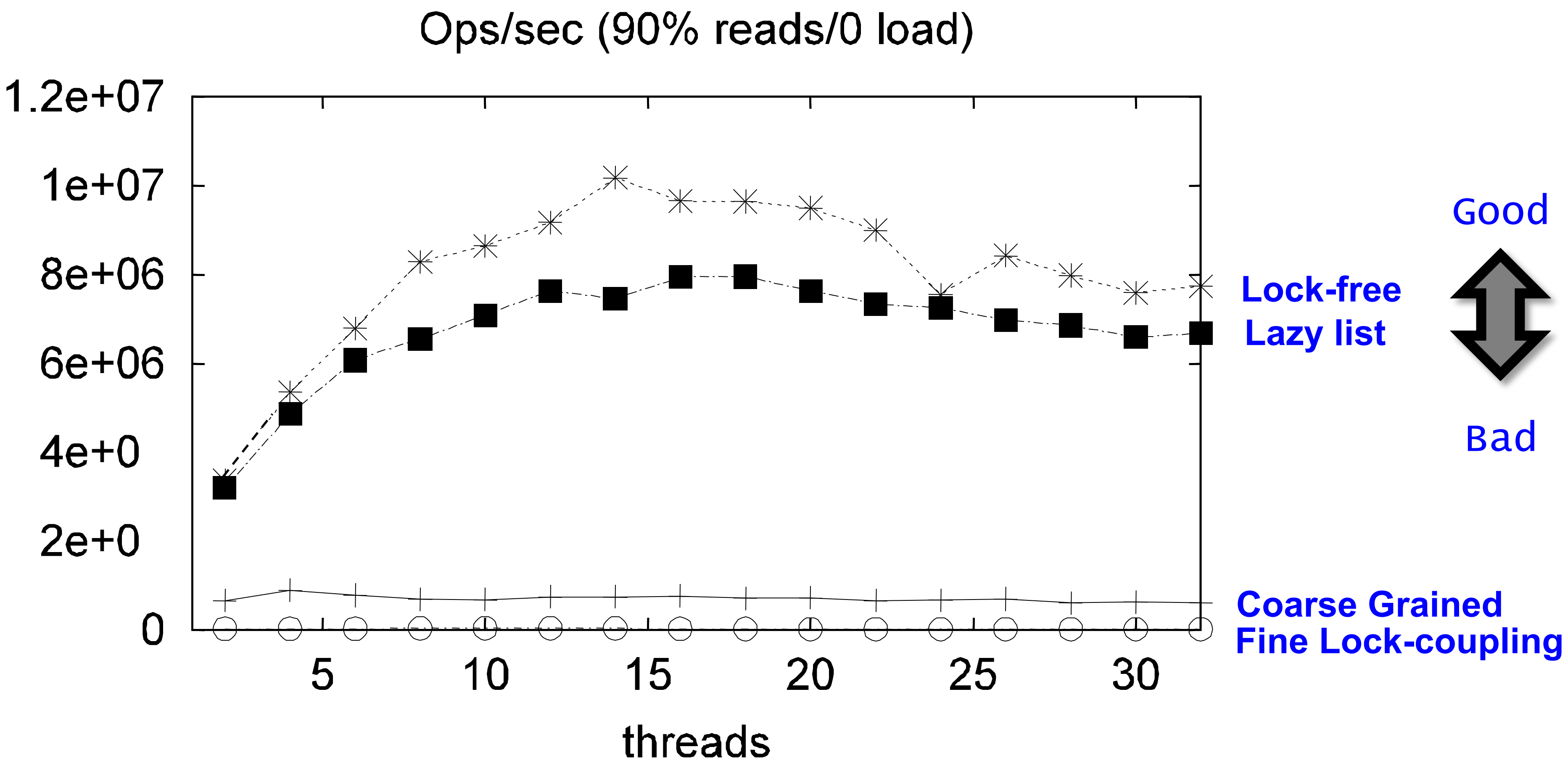
Otherwise move on to check if next node deleted

```
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);
   }
…
```

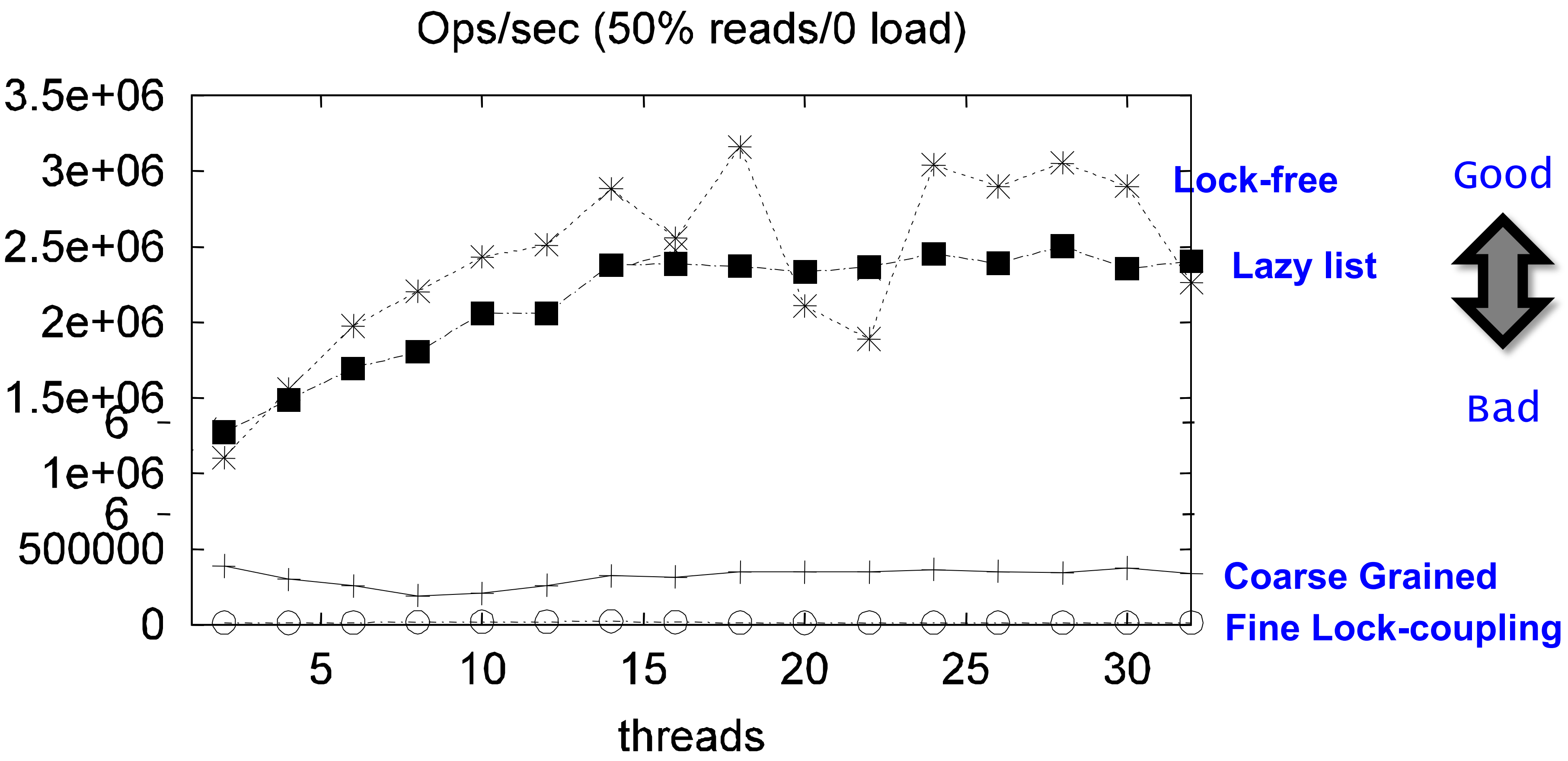
Performance

On 16 node shared memory machine
Benchmark throughput of Java List-based Set
algs. Vary % of Contains() method Calls.

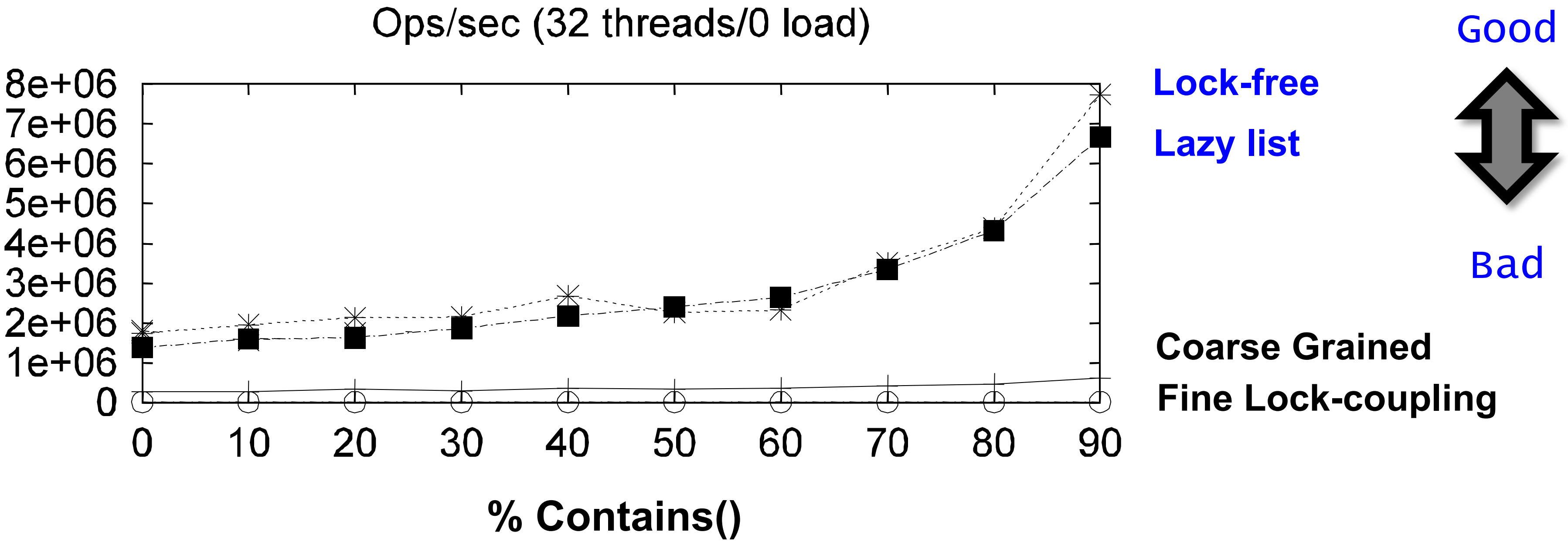
High Contains Ratio



Low Contains Ratio

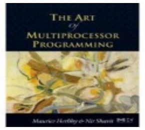


As Contains Ratio Increases



Summary

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



“To Lock or Not to Lock”

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