Lecture 4:
Concurrent Data Structures
(Concurrent Linked Lists)

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

With modifications by Lamont Samuels

Concurrent Data Structures

- We assume
  - shared-memory multiprocessors environment
  - concurrently execute multiple threads which communicate and synchronize through data structures in shared memory

Concurrent Data Structures

- Far more difficult to design than sequential ones
  - Correctness
    - Primary source of difficulty is concurrency
    - The steps of different threads can be interleaved arbitrarily
  - Scalability (performance)
- We will look at
  - Concurrent Linked List/Queue/Stack

Main performance issue of lock based system

- Sequential bottleneck
  - At any point in time, at most one lock-protected operation is doing useful work.
- Memory contention
  - Overhead in traffic as a result of multiple threads concurrently attempting to access the same memory location.
- Blocking
  - If thread that currently holds the lock is delayed, then all other threads attempting to access are also delayed.
  - Consider non-blocking (lock-free) algorithm

Nonblocking algorithms

- implemented by a hardware operation
  - atomically combines a load and a store
  - Ex) compare-and-swap(CAS)
- lock-free
  - if there is guaranteed system-wide progress,
  - while a given thread might be blocked by other threads, all CPUs can continue doing other useful work without stalls.
- wait-free
  - if there is also guaranteed per-thread progress.
  - in addition to all CPUs continuing to do useful work, no computation can ever be blocked by another computation.

Linked List

- Illustrate these patterns …
- Using a list-based Set
  - Common application
  - Building block for other apps

Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - add(x) put x in set
  - remove(x) take x out of set
  - contains(x) tests if x in set

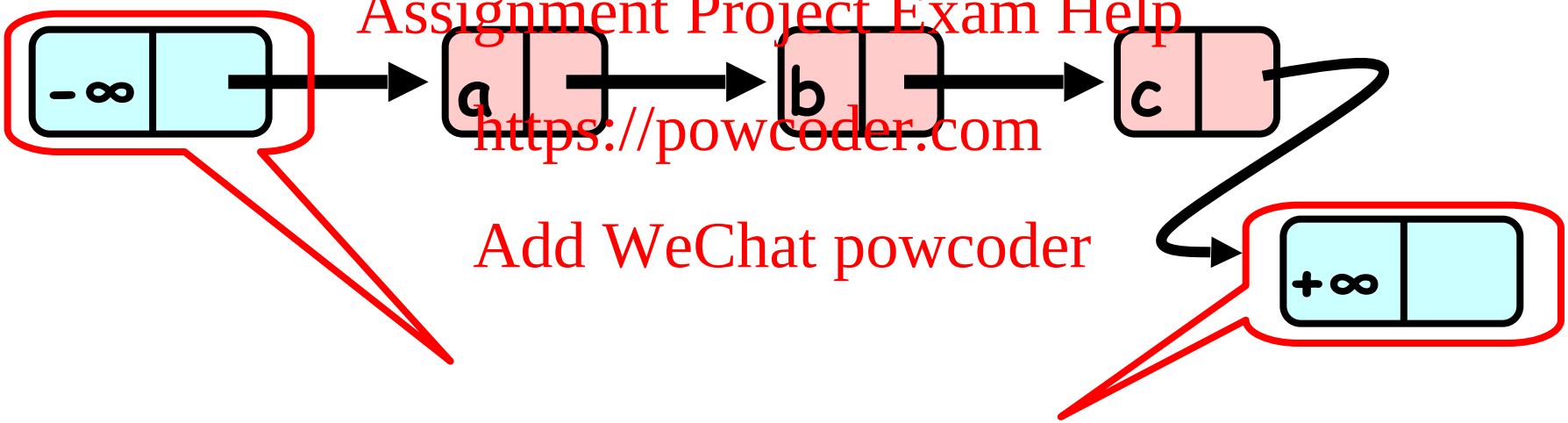
List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

List Node

```
public class Node {
 public T item;      // item of interest
 public int key;     // usually hash code
 public Node next;   // reference to next node
}
```

The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

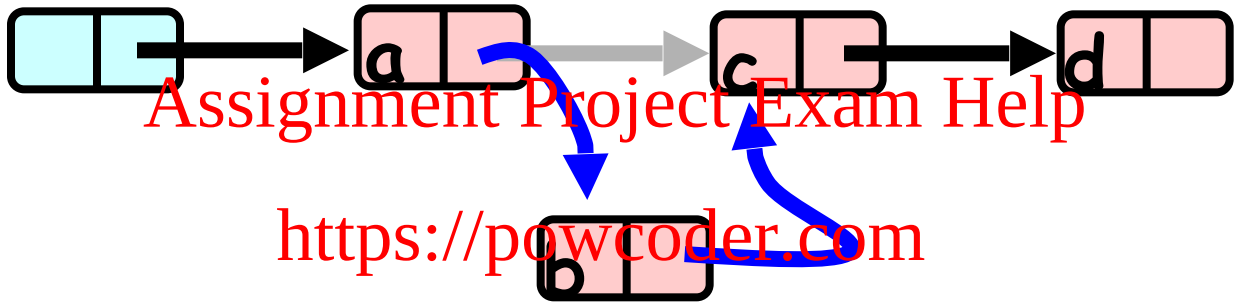
Sequential List Based Set

Add()

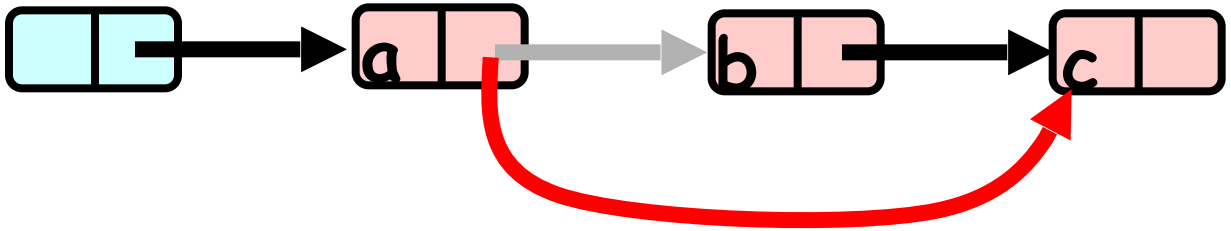
Remove()

Sequential List Based Set

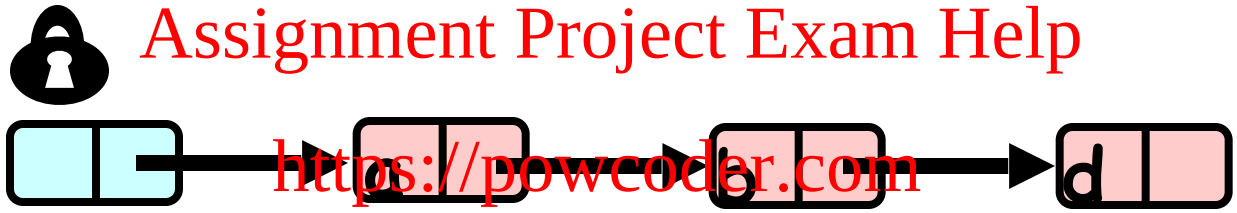
Add()



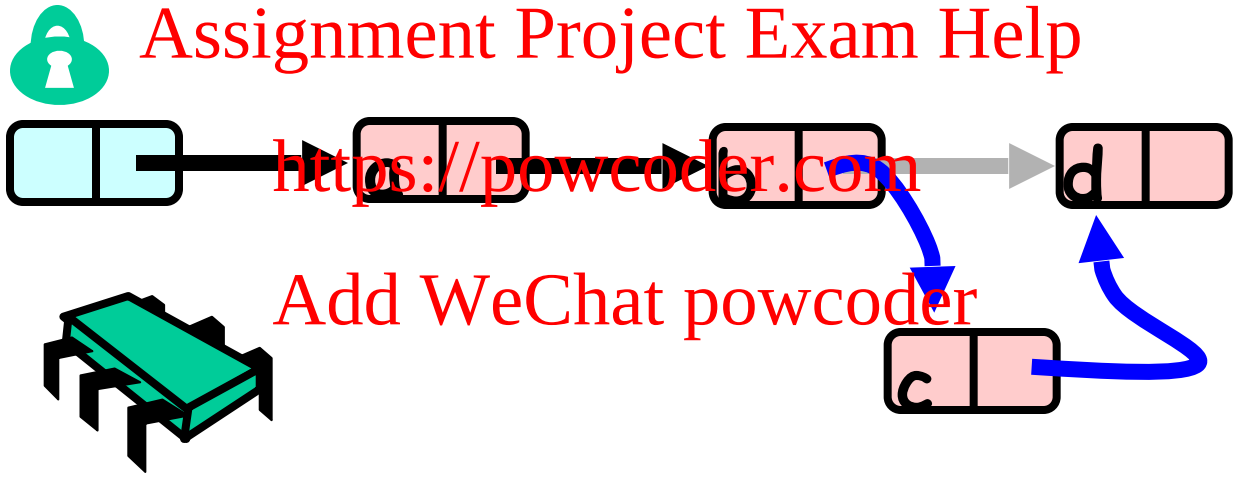
Remove()



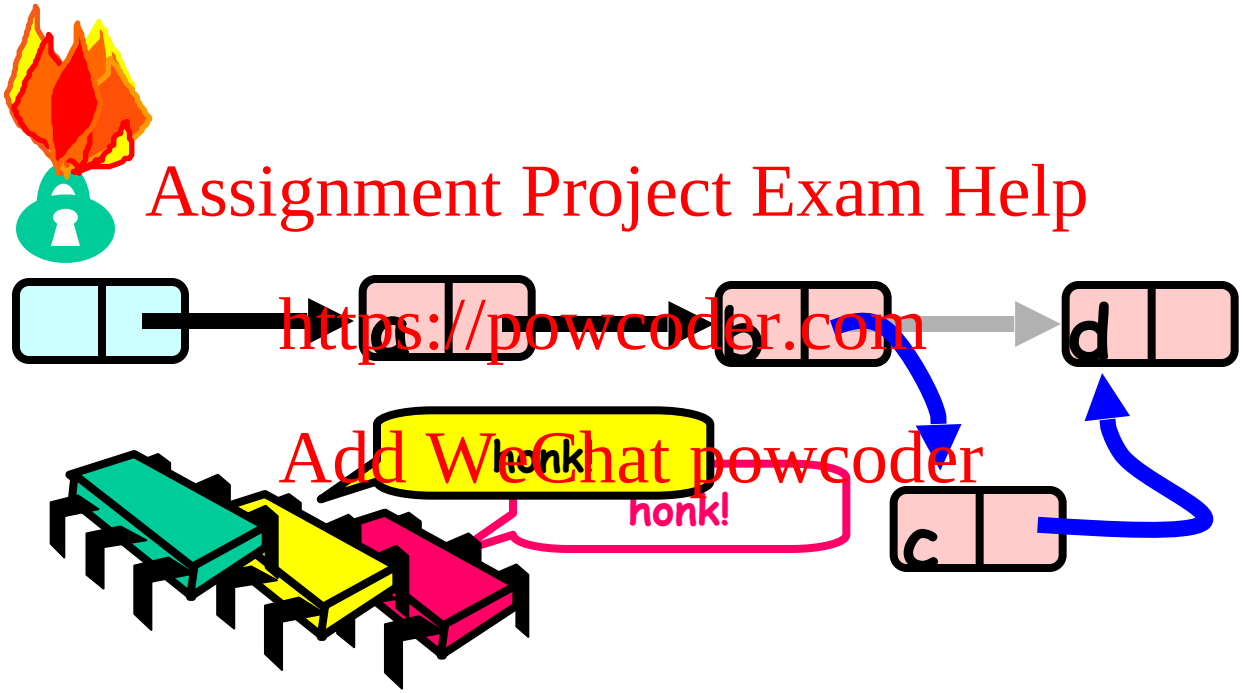
Course Grained Locking



Course Grained Locking



Course Grained Locking



Simple but hotspot + bottleneck

Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads "stand in line"
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel …

Coarse-Grained Synchronization (Linked List)

```
public class CoarseList<T> {
private Node head;
private Node tail;
private Lock lock = new ReentrantLock();

public CoarseList() {
    // Add sentinels to start and end
    head  = new Node(Integer.MIN_VALUE);
    tail  = new Node(Integer.MAX_VALUE);
    head.next = this.tail;
  }
```

```
public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(item);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      lock.unlock();
    }
  }
```

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   lock.lock();
   try {
     pred = this.head;
     curr = pred.next;
     while (curr.key < key) {
       pred = curr;
       curr = curr.next;
     }
     if (key == curr.key)
       pred.next = curr.next;
       return true;
     } else {
       return false;
     }
   } finally {
     lock.unlock();
   }
 }
```

```
public boolean contains(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      return (key == curr.key);
    } finally {
      lock.unlock();
    }
  }
```

Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all …"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention

Performance Improvement

- For highly-concurrent objects
- Goal:
  - Concurrent access
  - More threads, more throughput

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
  - Mistakes are expensive

Third:
Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component that should 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

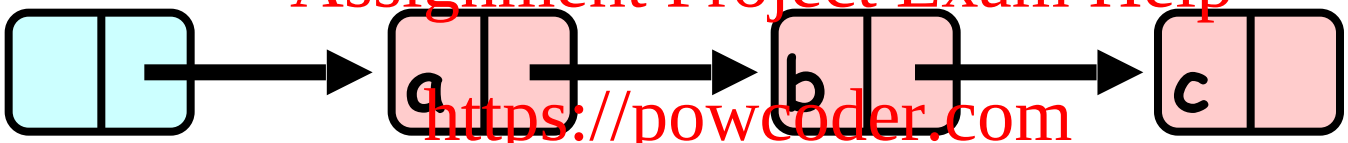
Fine-grained Locking

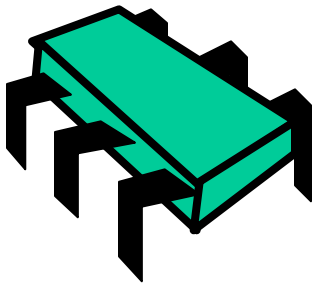
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Fine-grained Locking

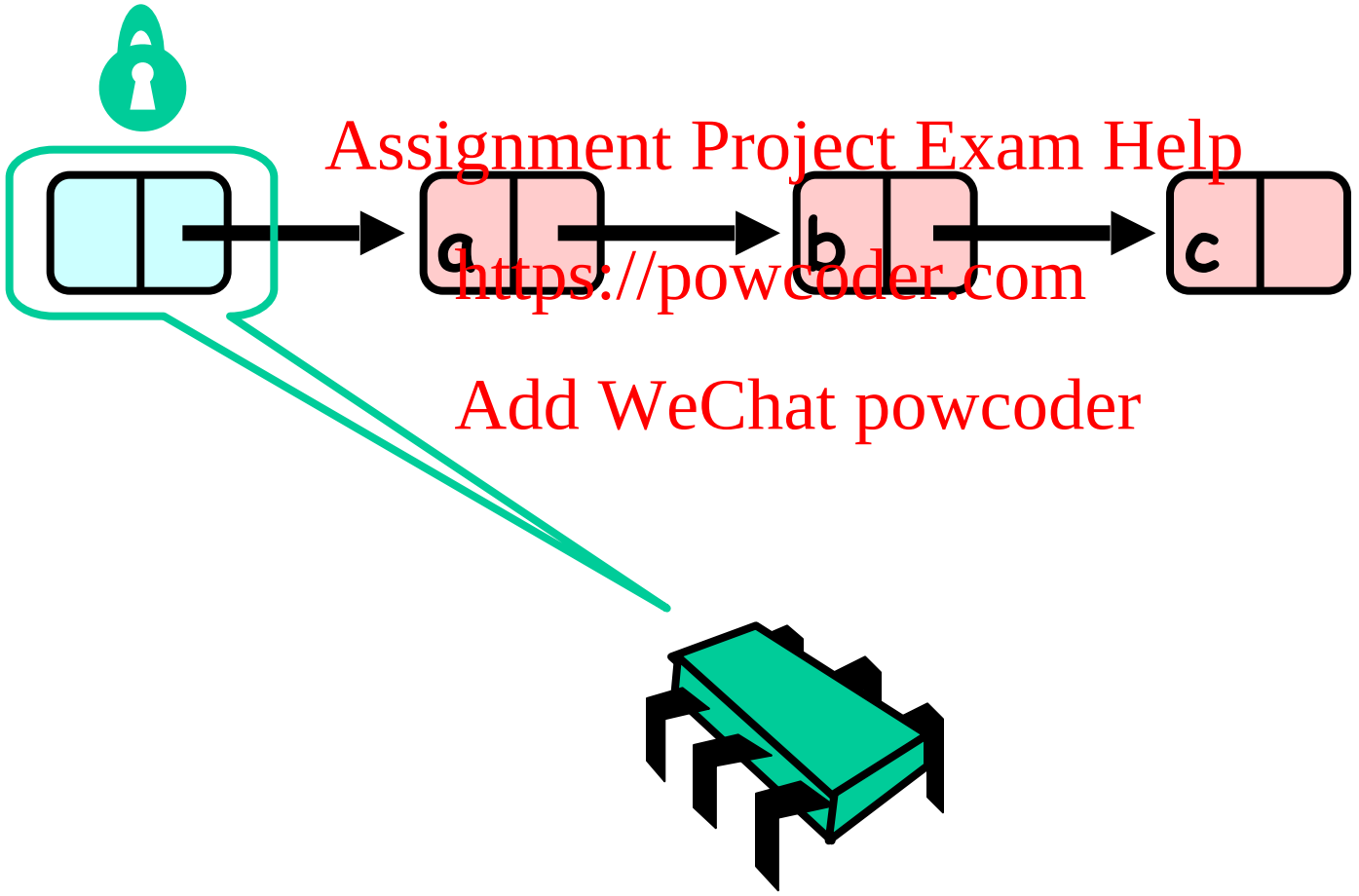
- Use multiple locks of small granularity to protect different parts of the data structure
- Goal
  - To allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure

Hand-over-Hand locking

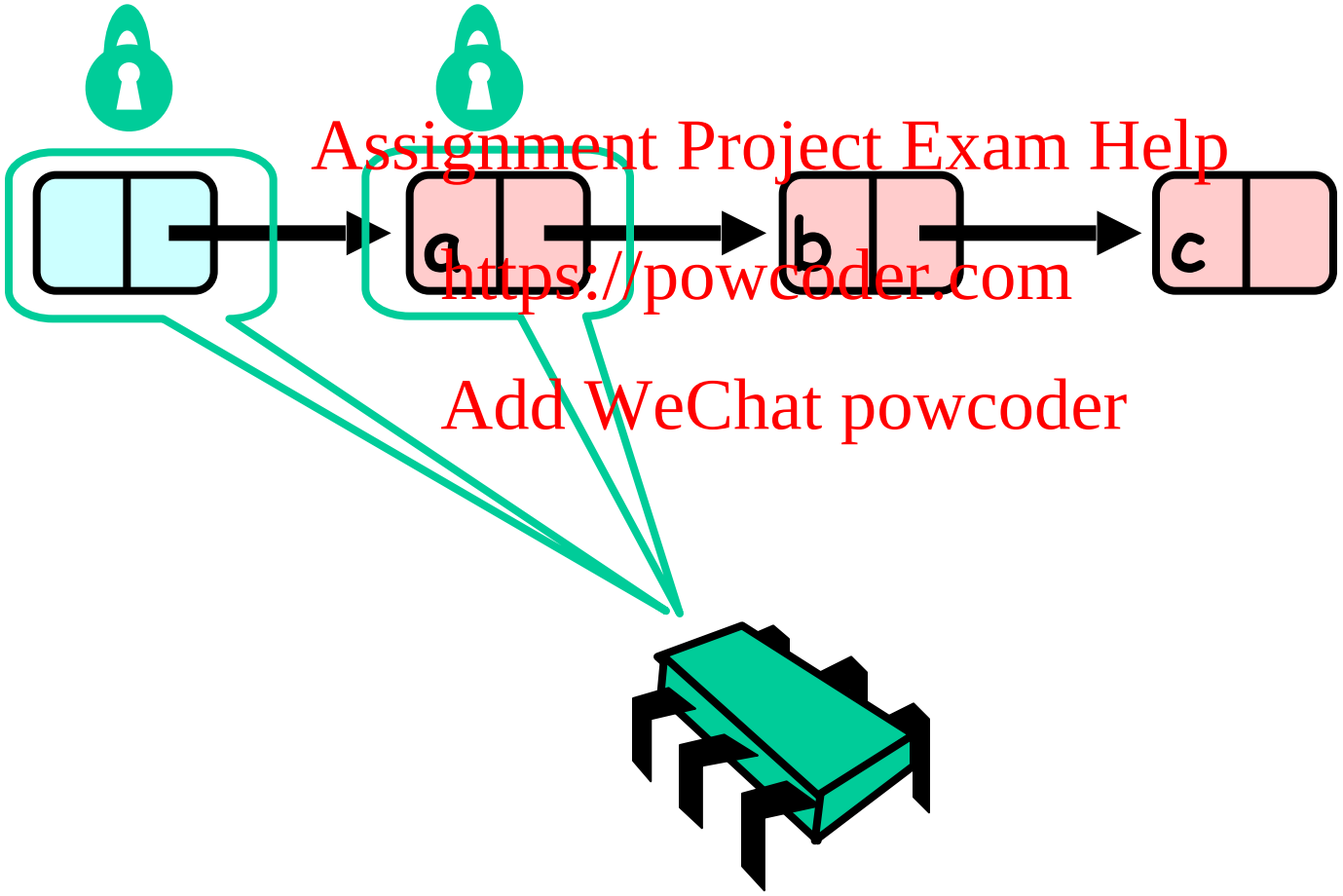




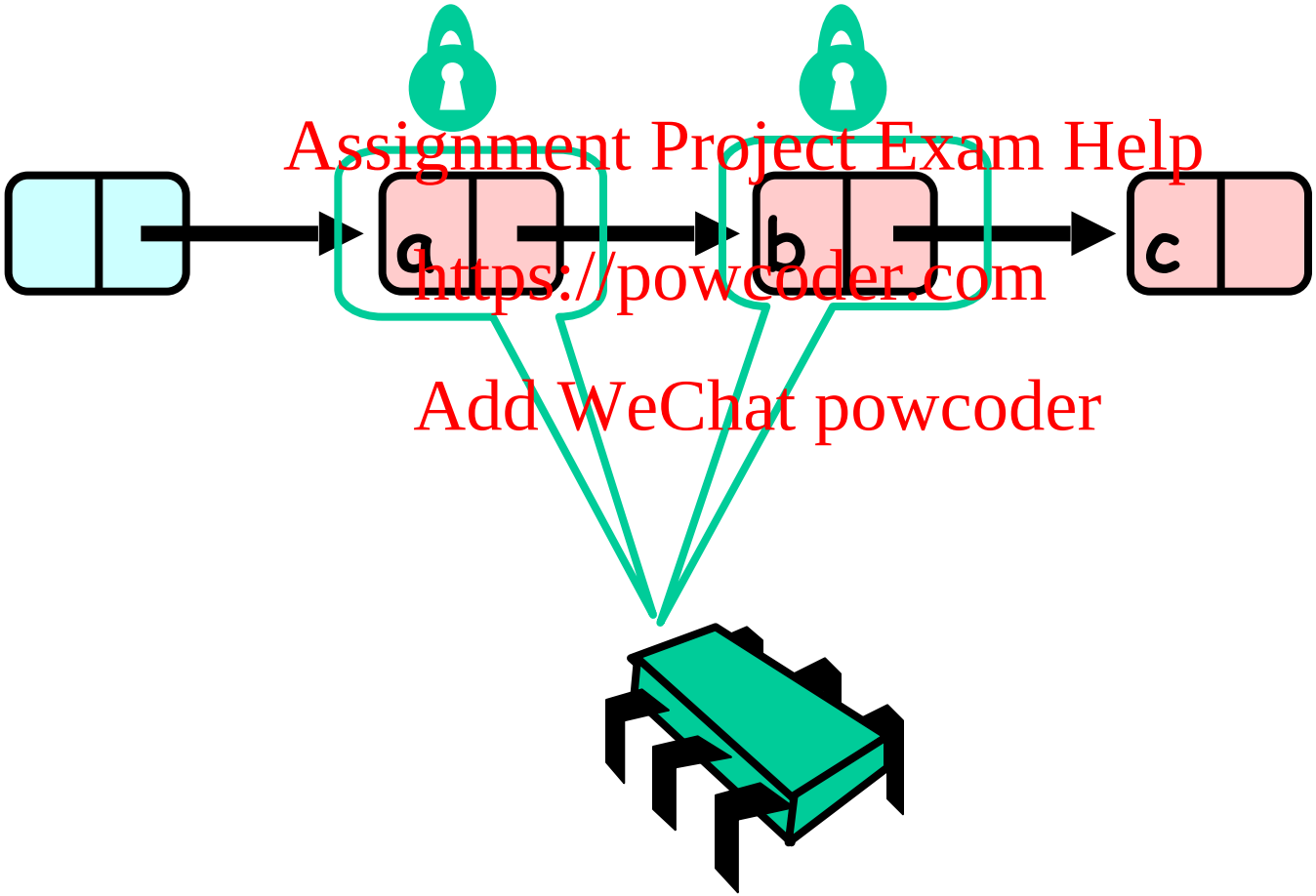
Hand-over-Hand locking



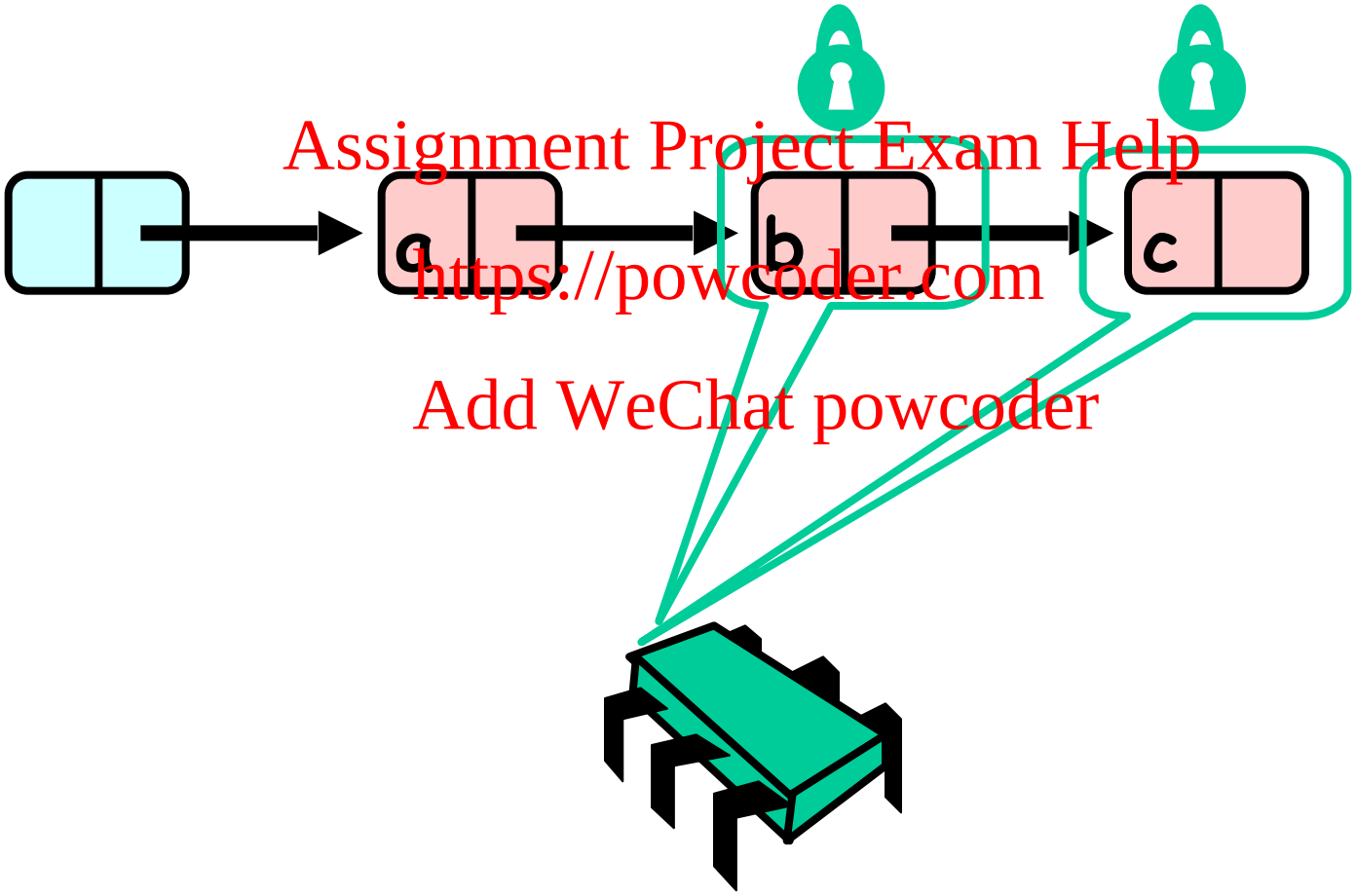
Hand-over-Hand locking



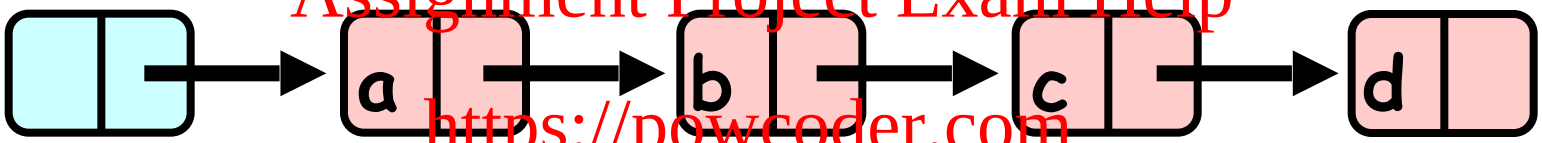
Hand-over-Hand locking

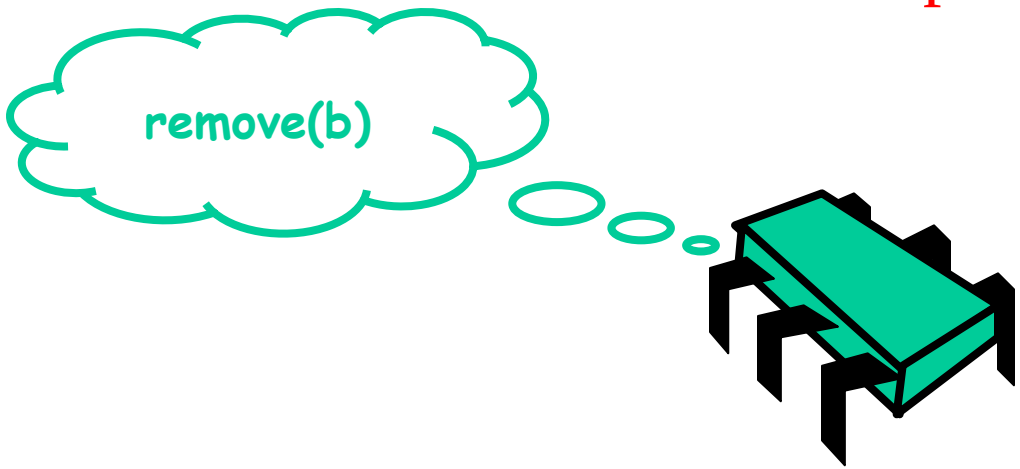


Hand-over-Hand locking

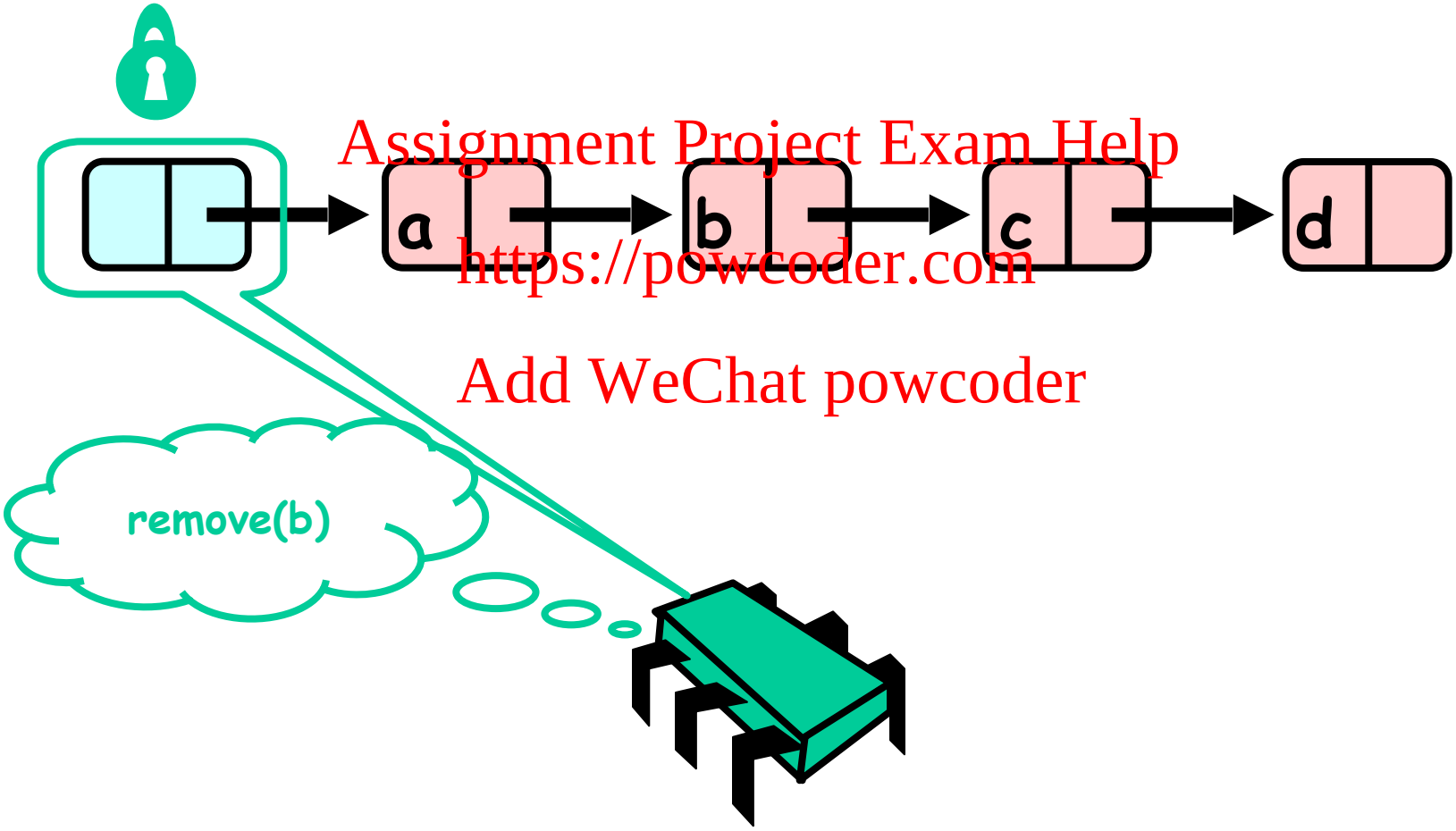


Removing a Node

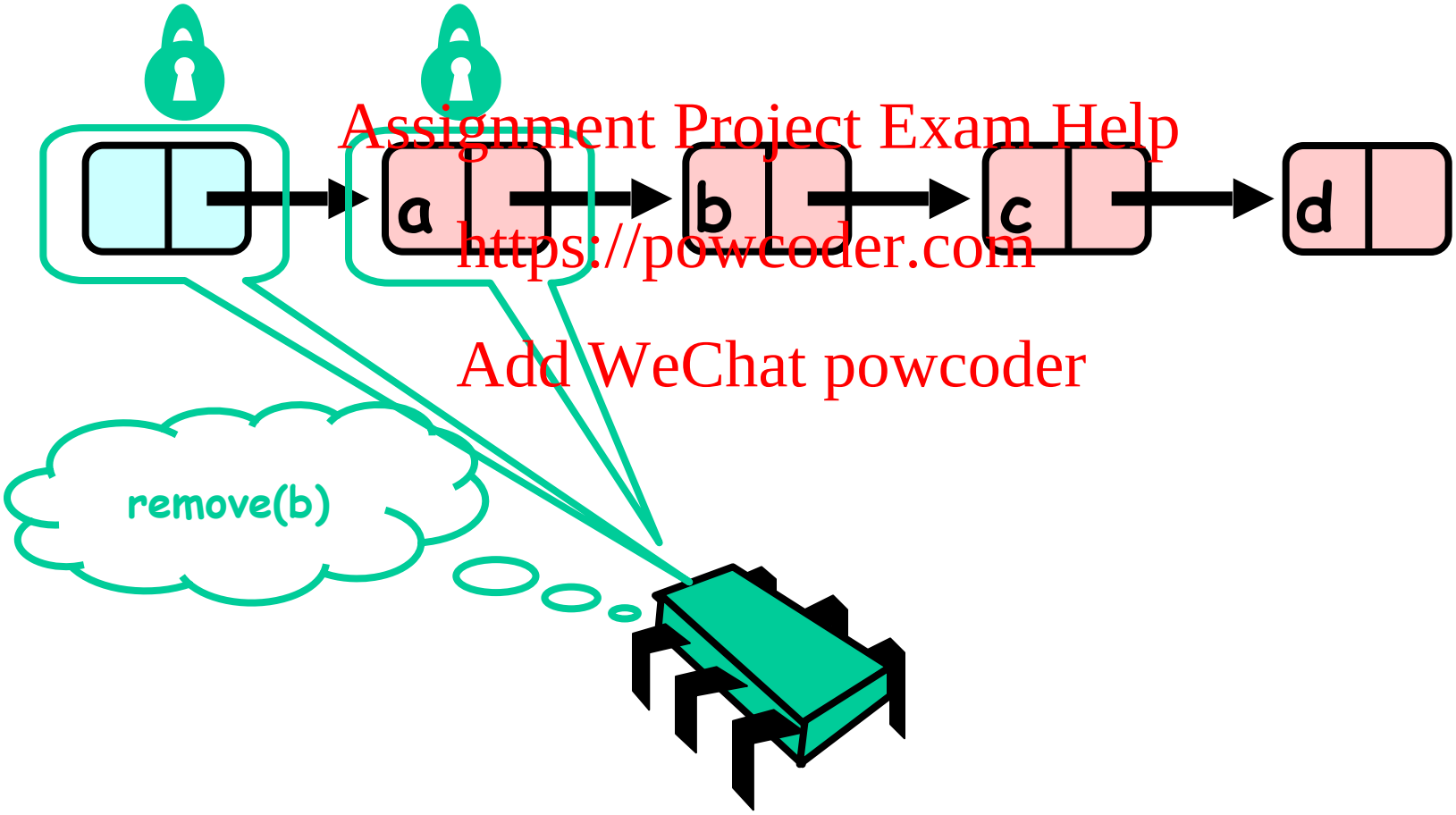




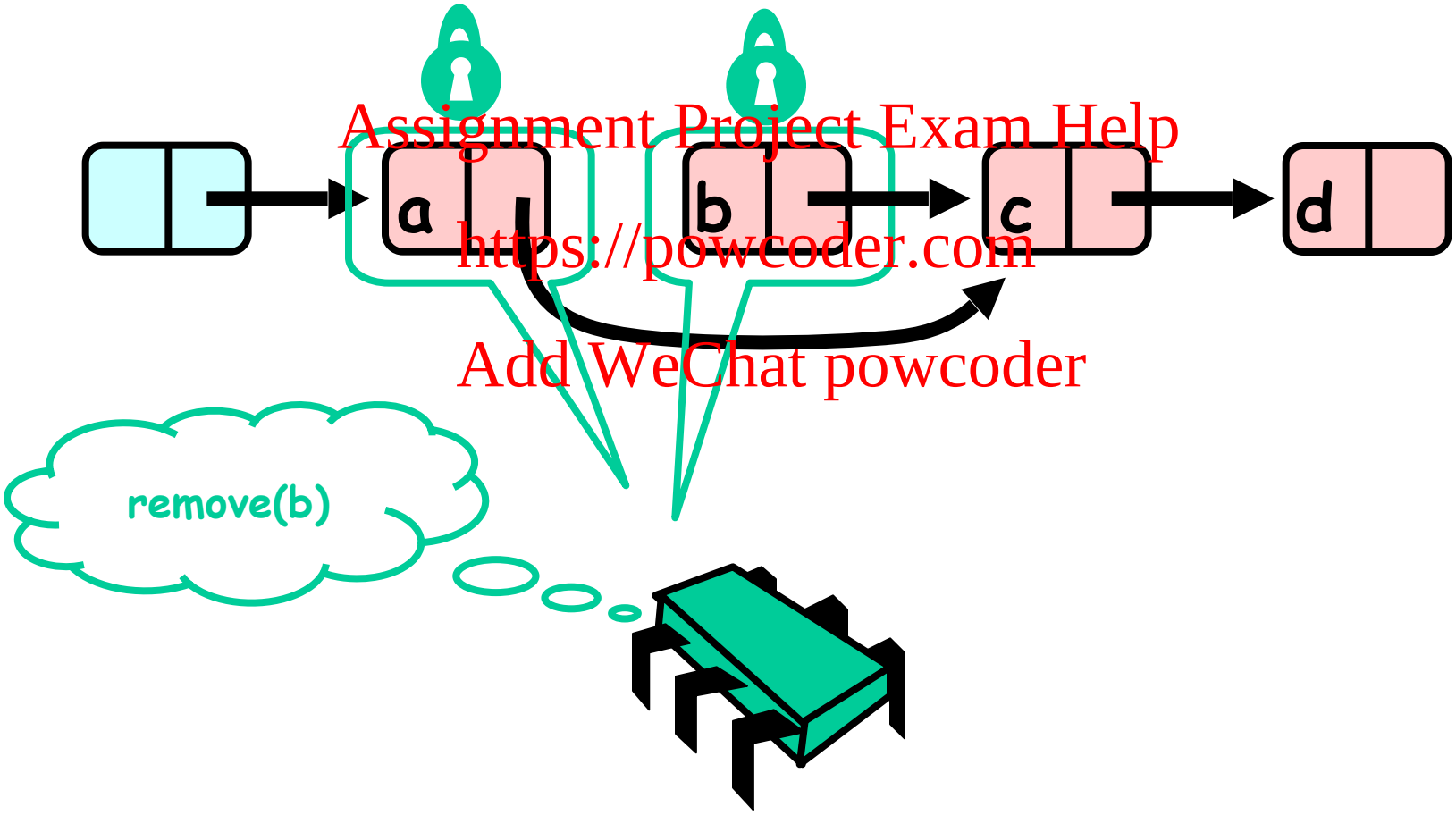
Removing a Node



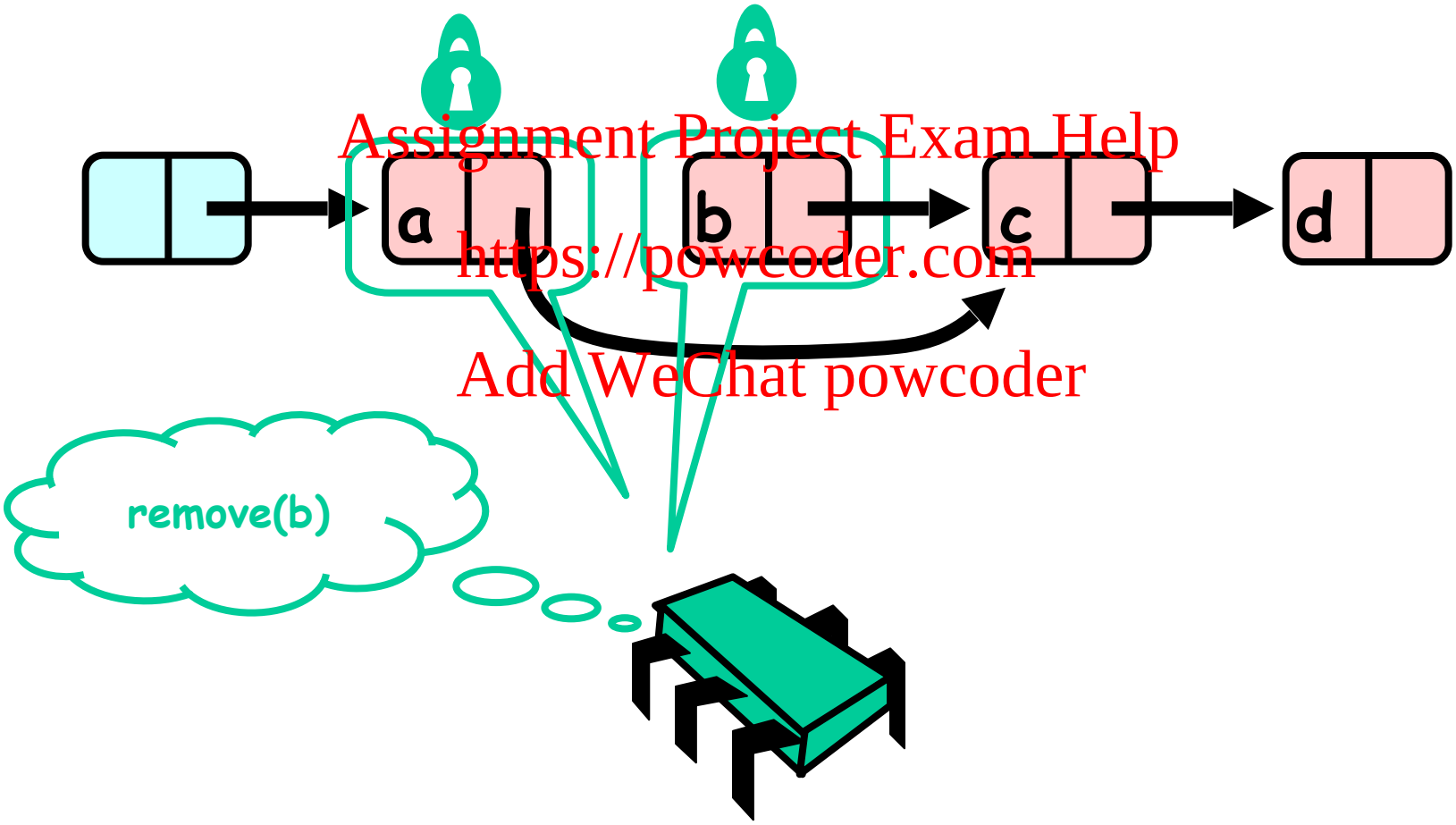
Removing a Node



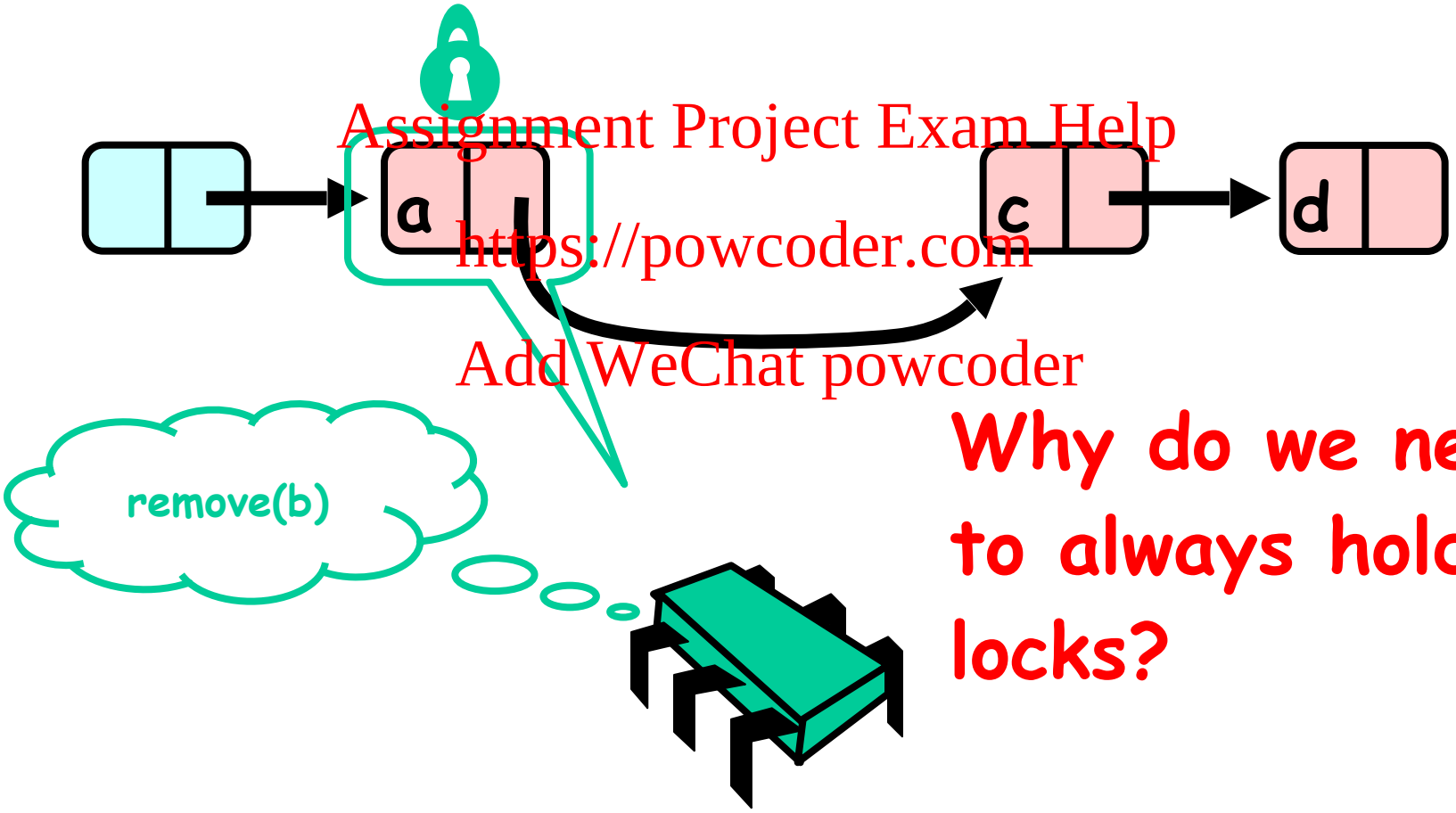
Removing a Node



Removing a Node

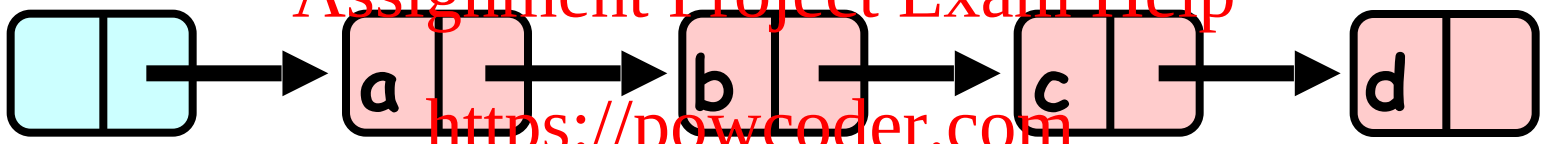


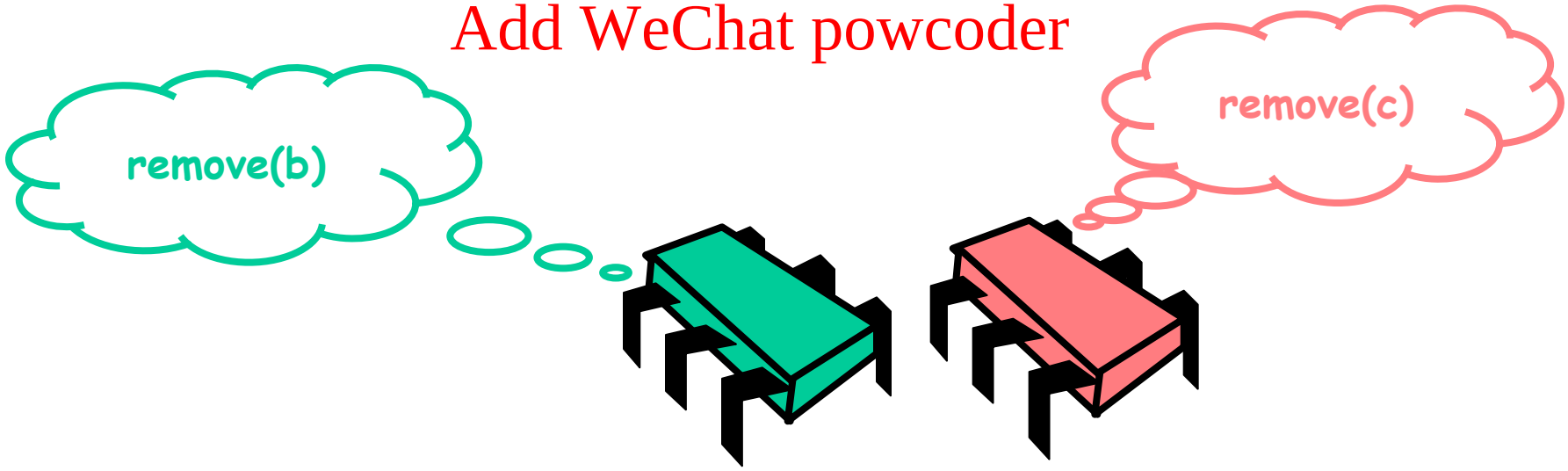
Removing a Node



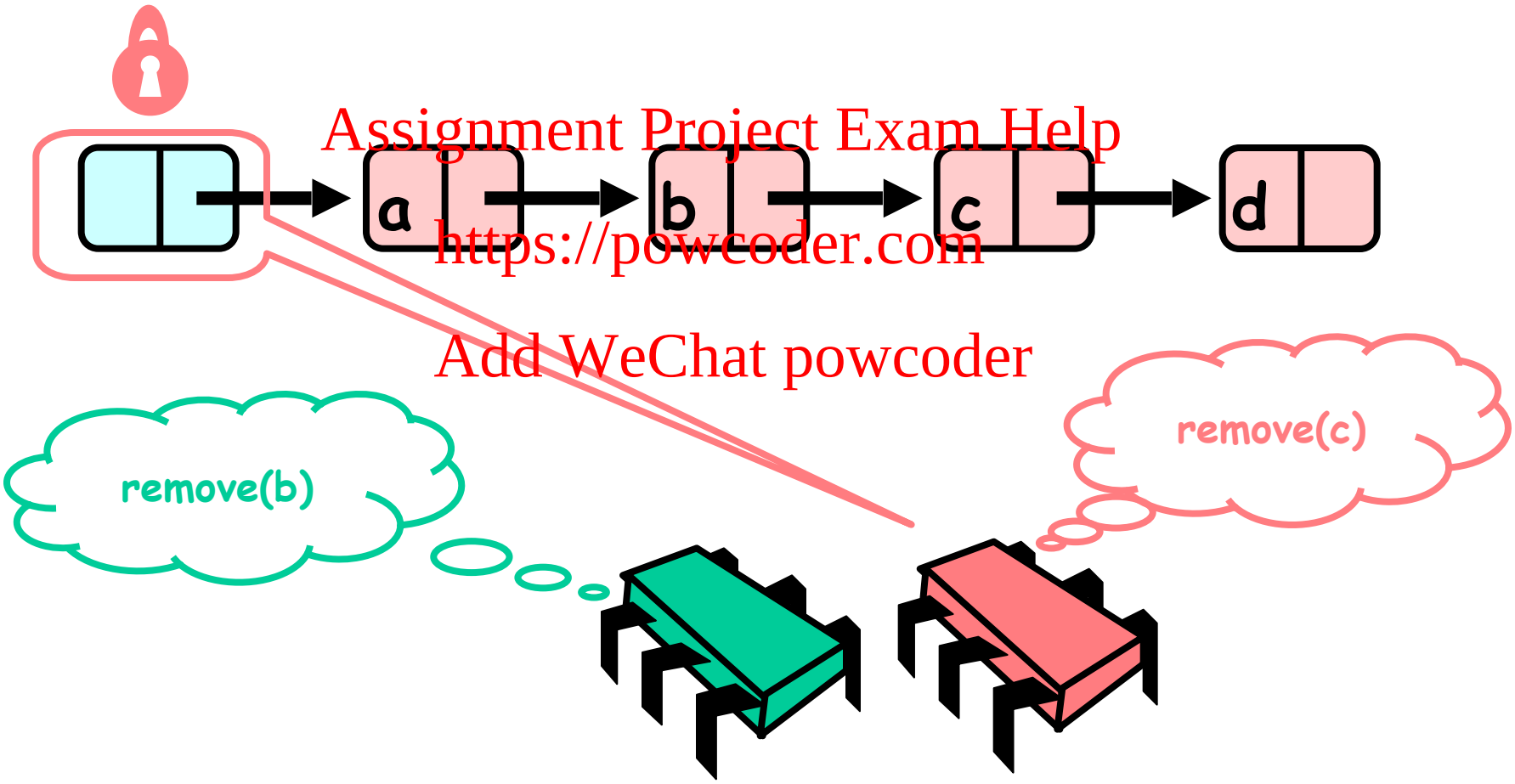
Why do we need to always hold 2 locks?

Concurrent Removes

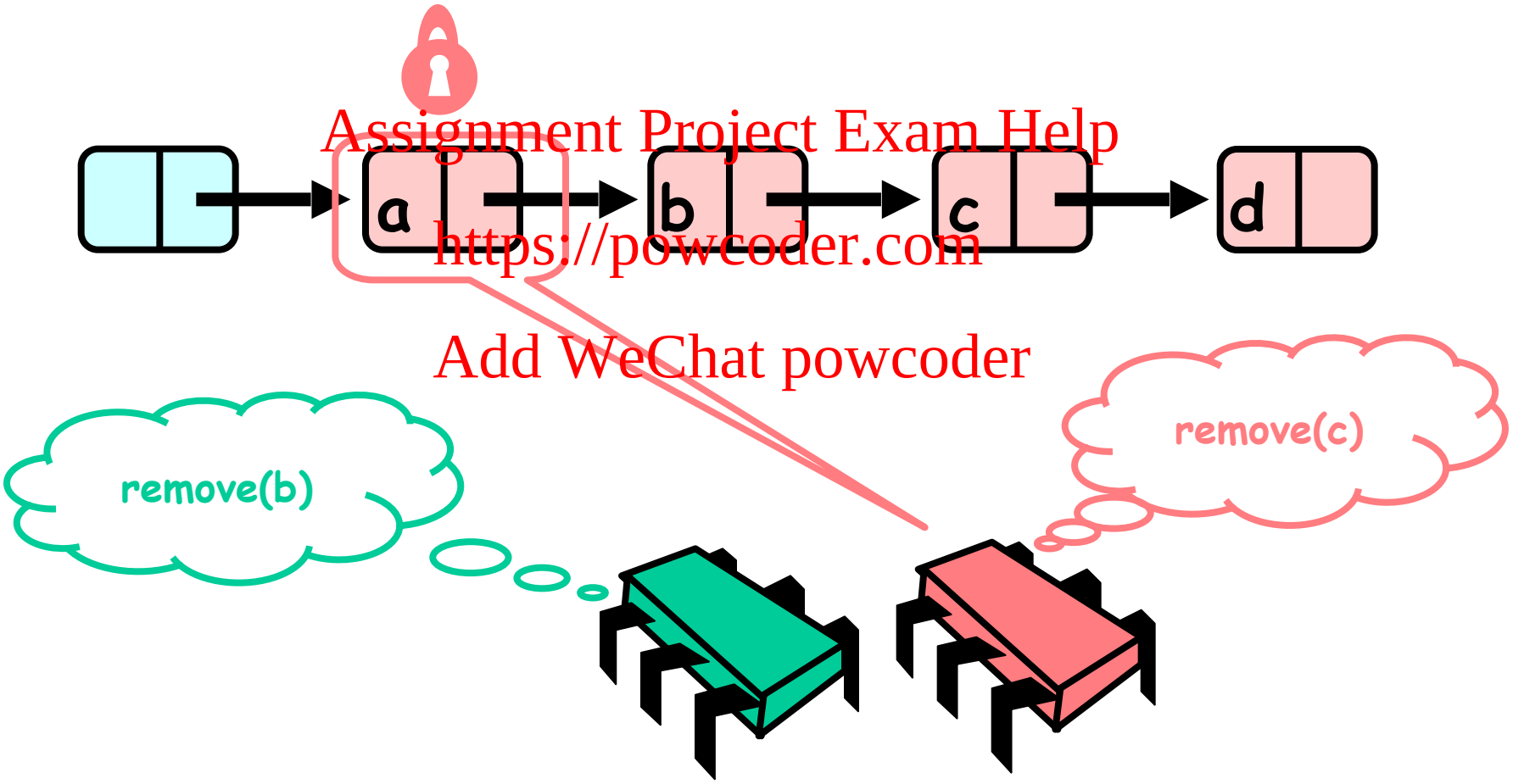




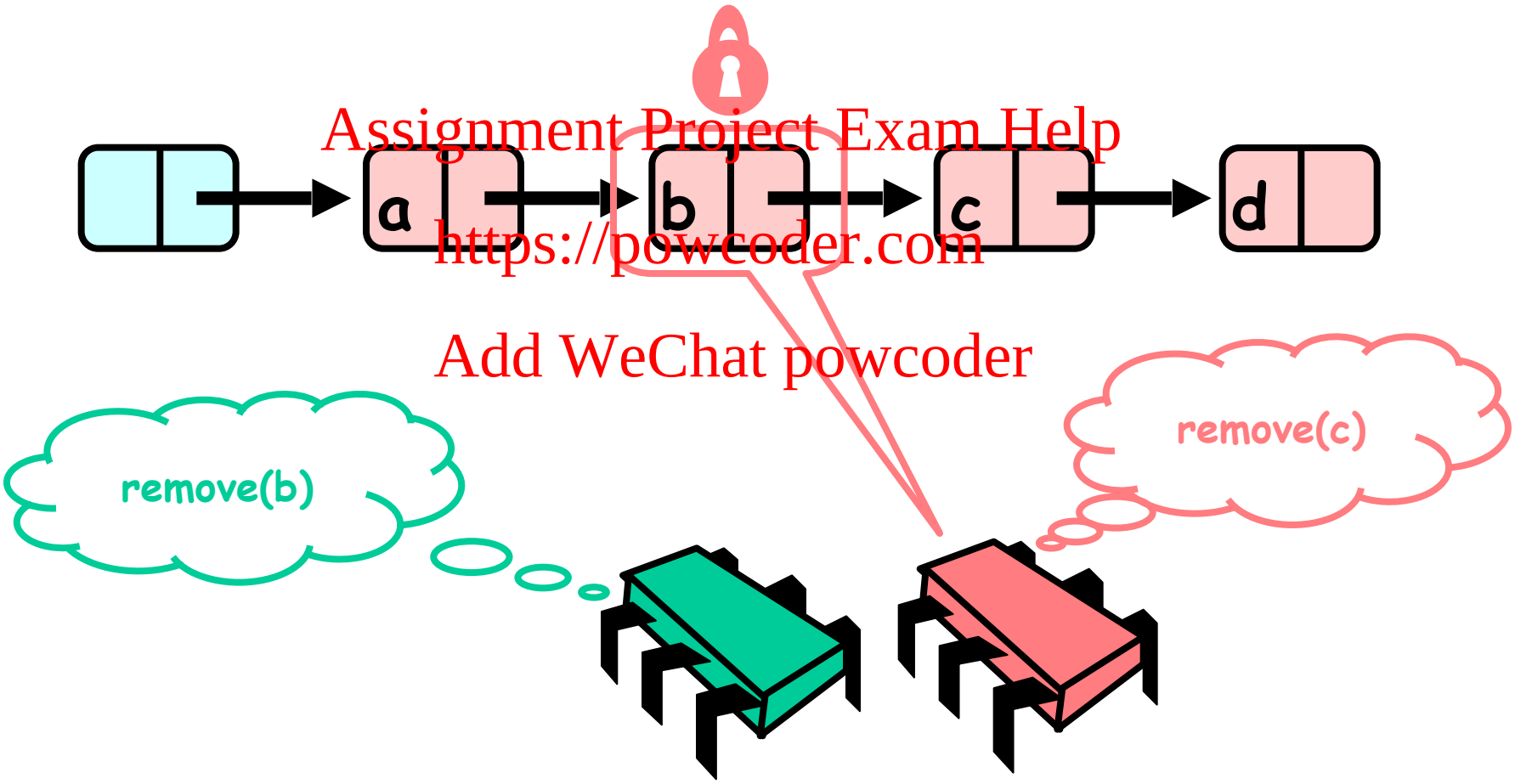
Concurrent Removes



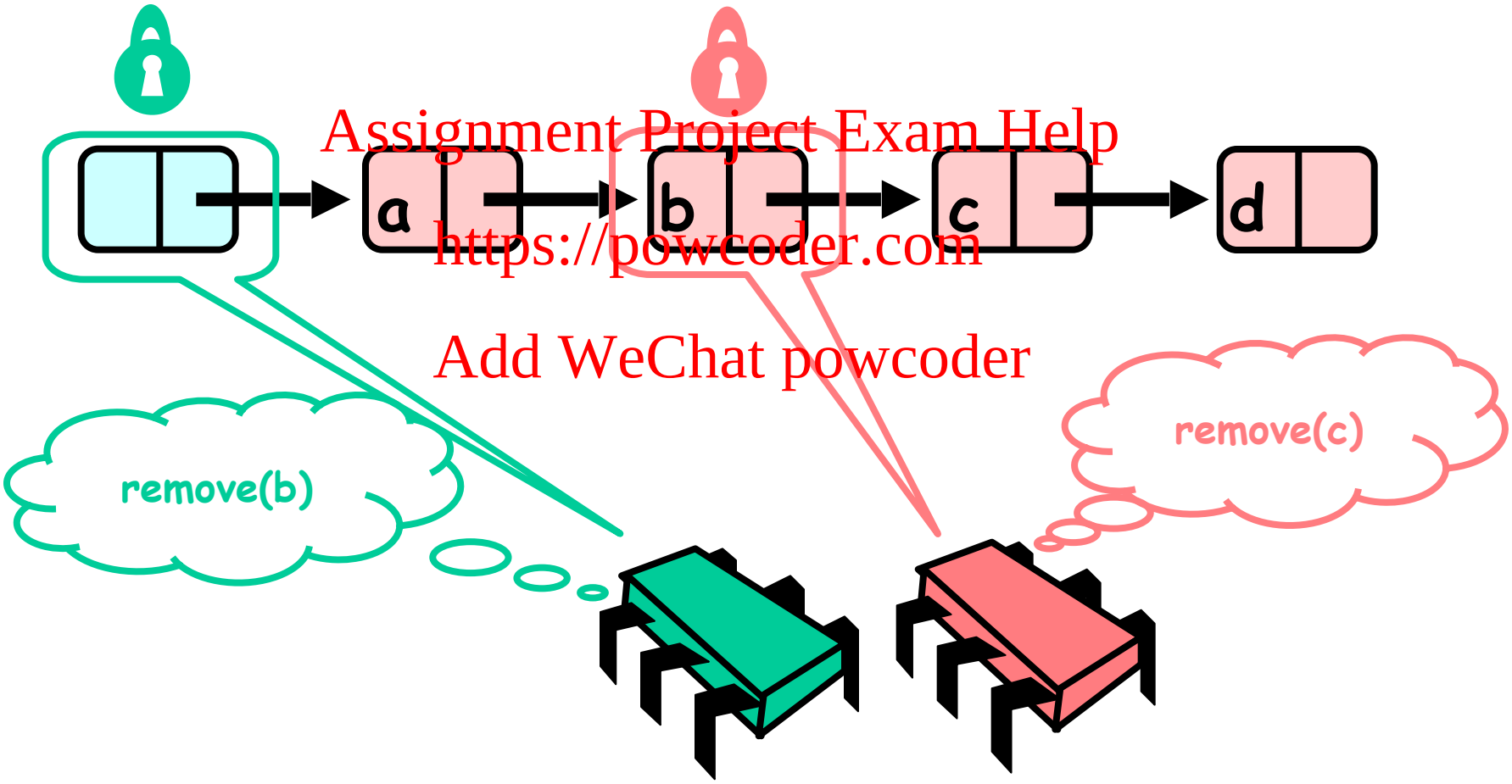
Concurrent Removes



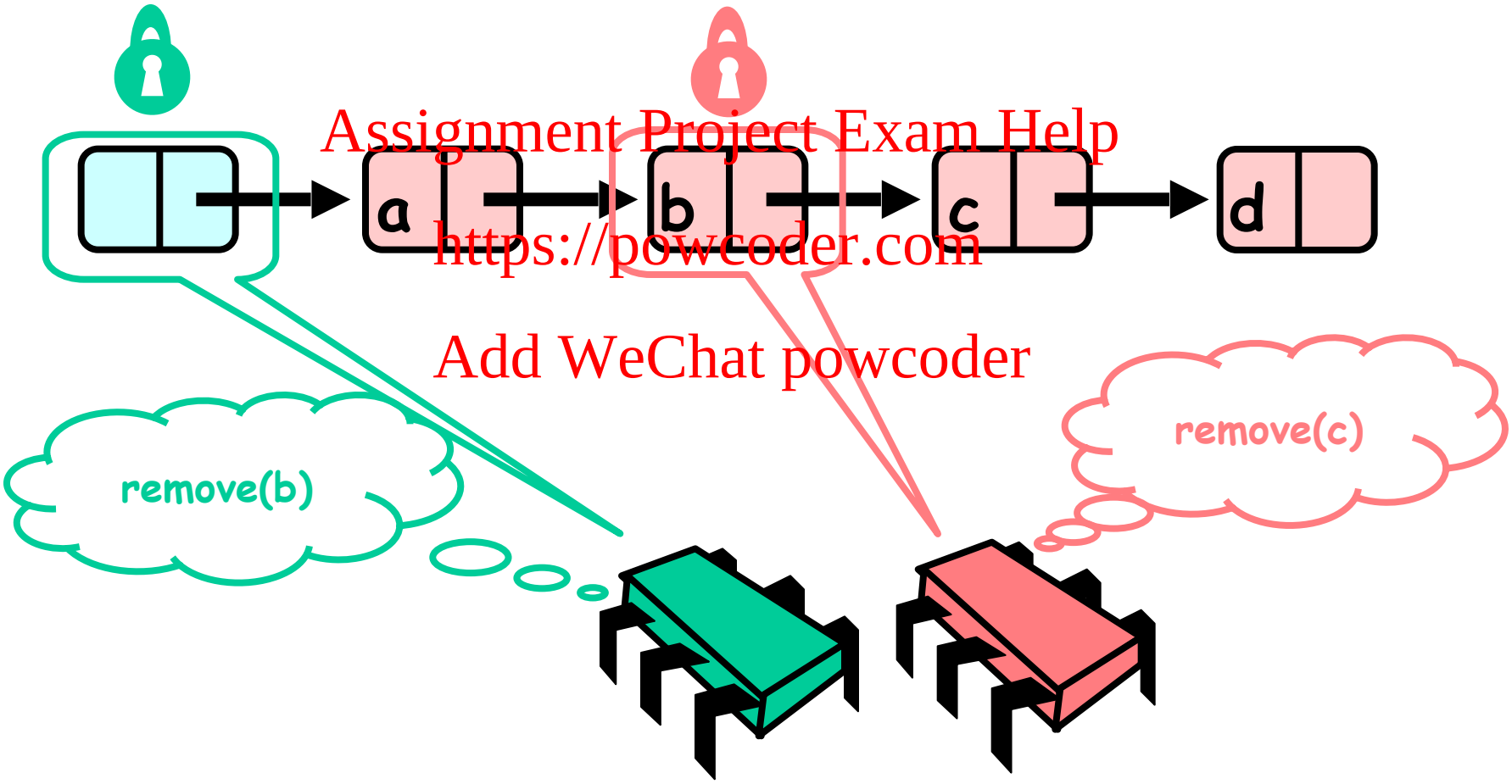
Concurrent Removes



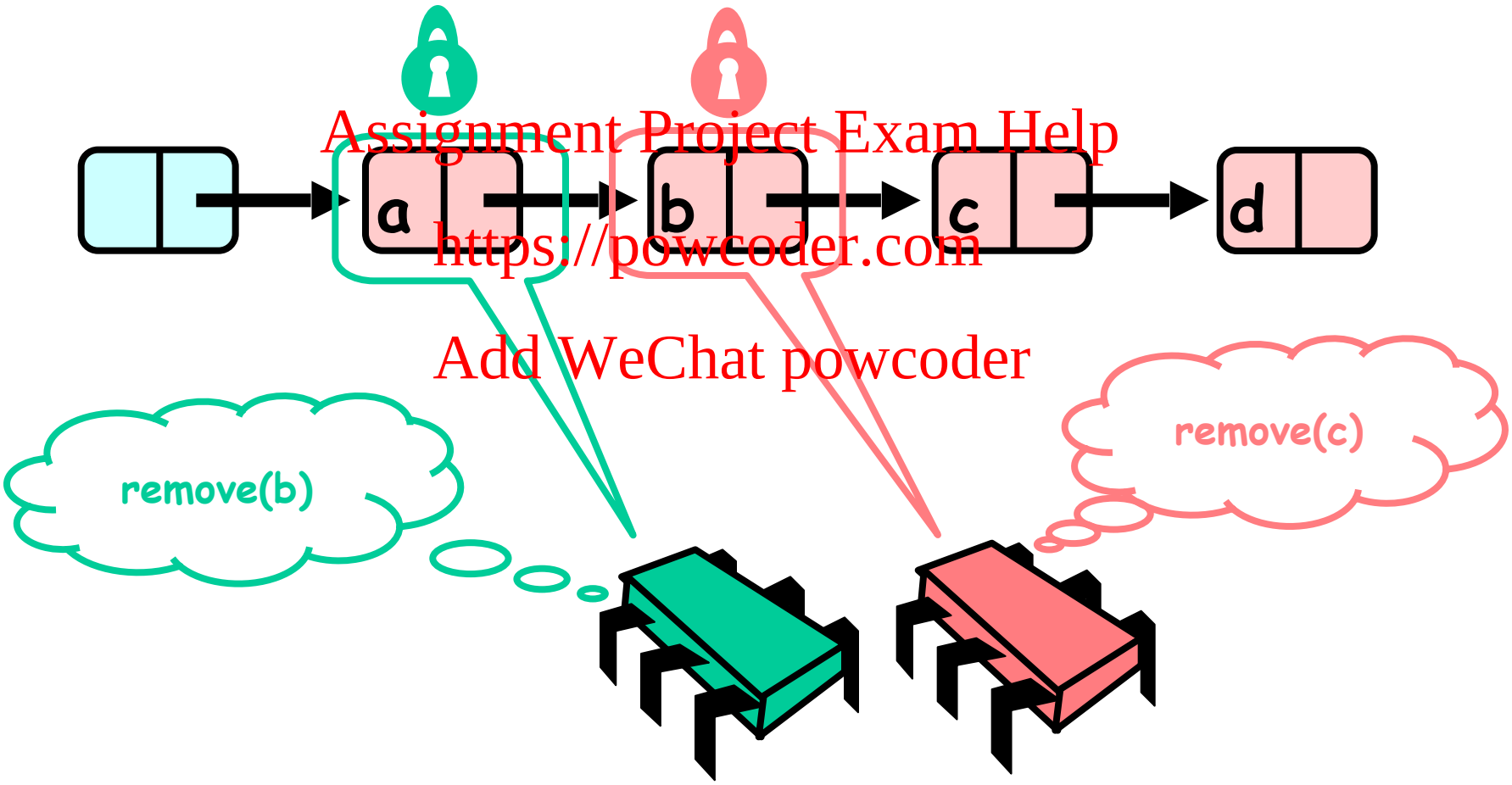
Concurrent Removes



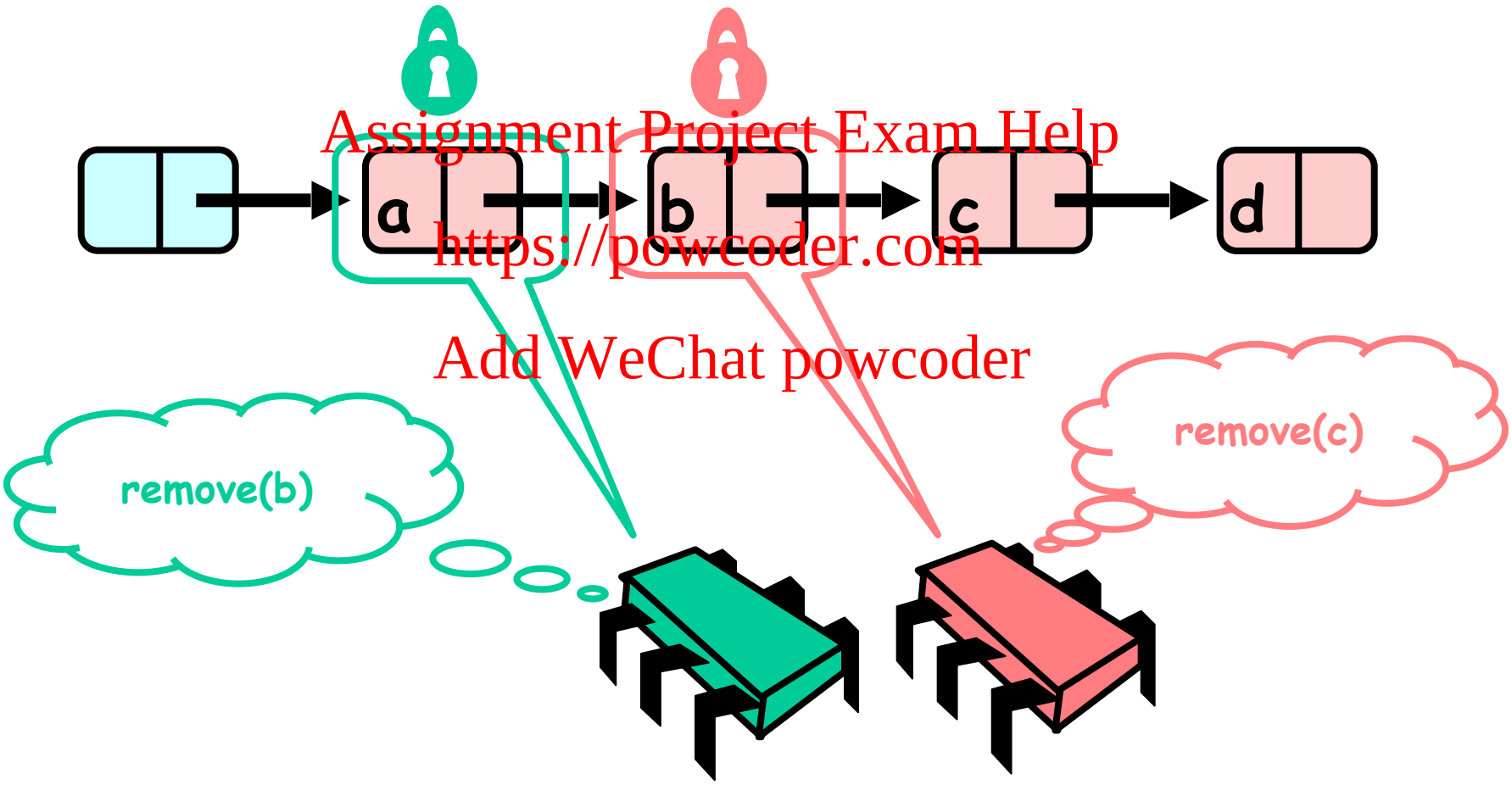
Concurrent Removes



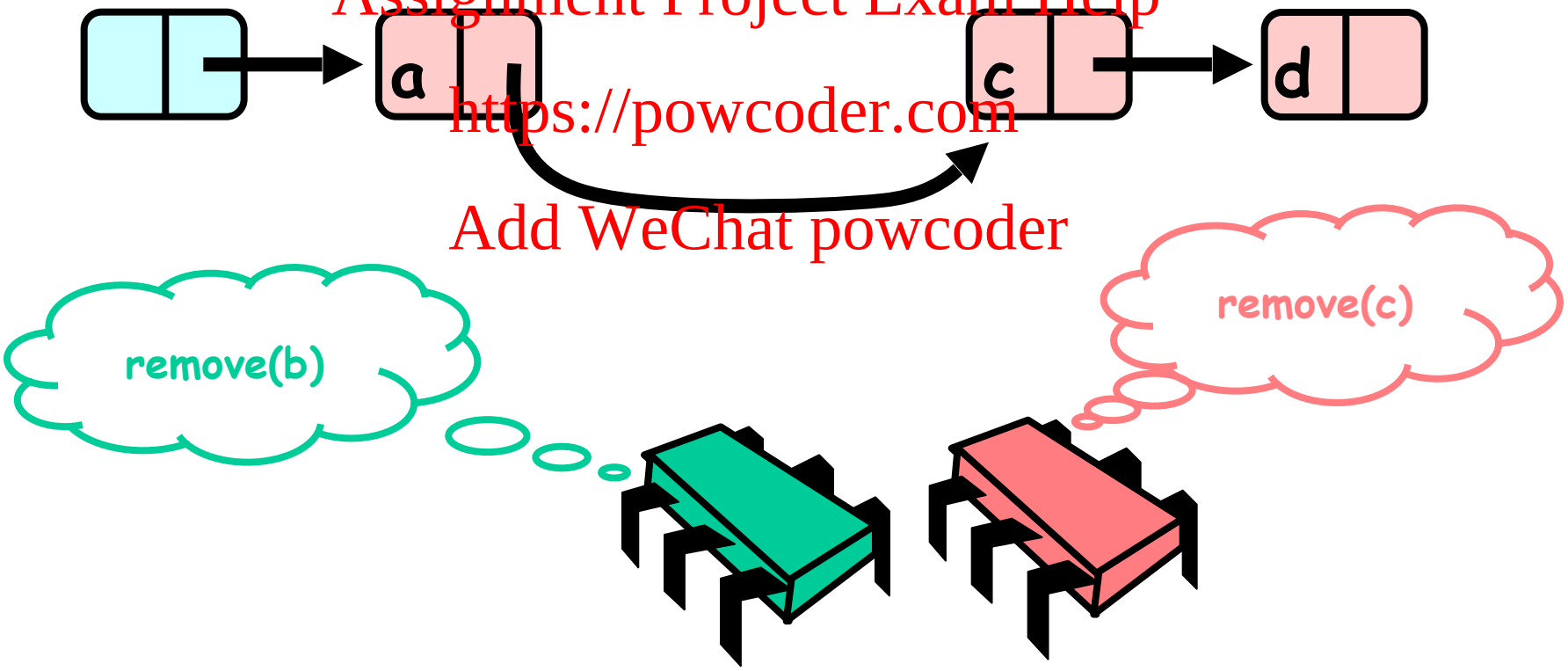
Concurrent Removes



Concurrent Removes

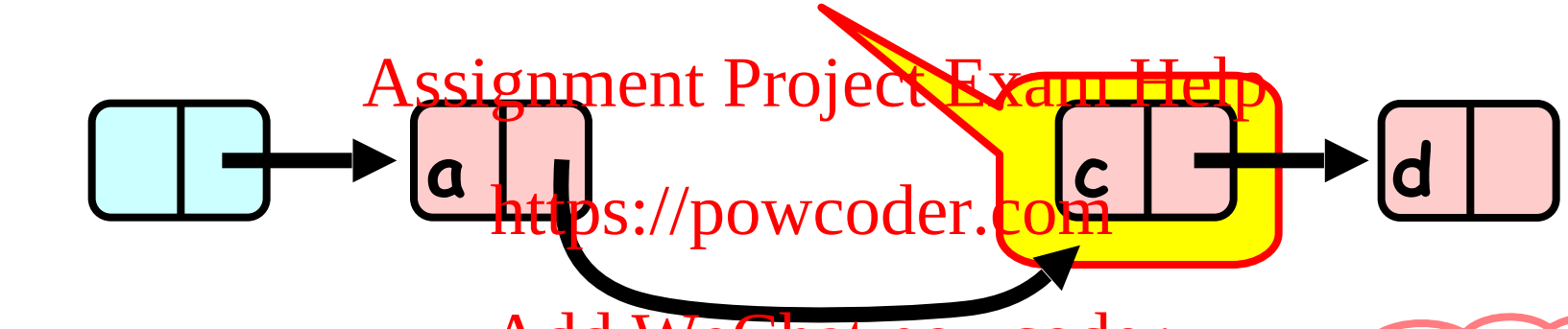


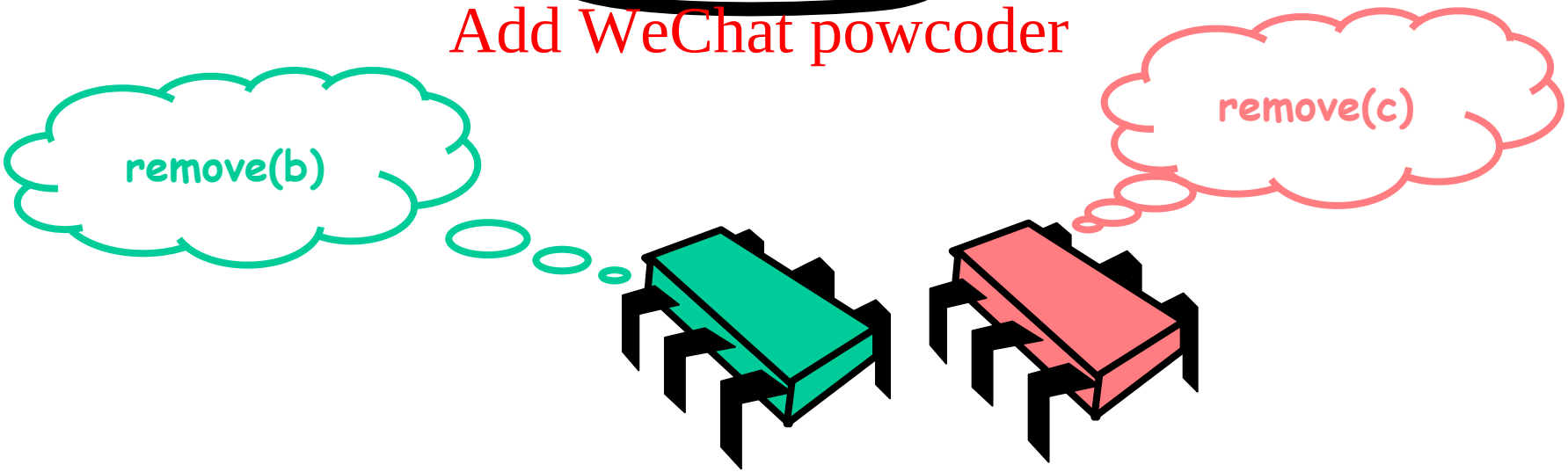
Uh, Oh



Uh, Oh

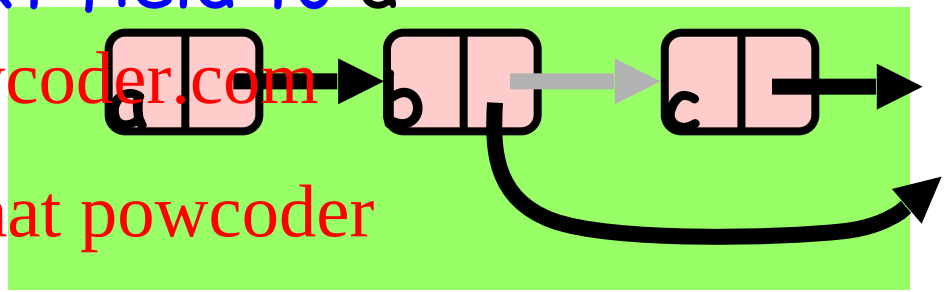
Bad news, C not removed

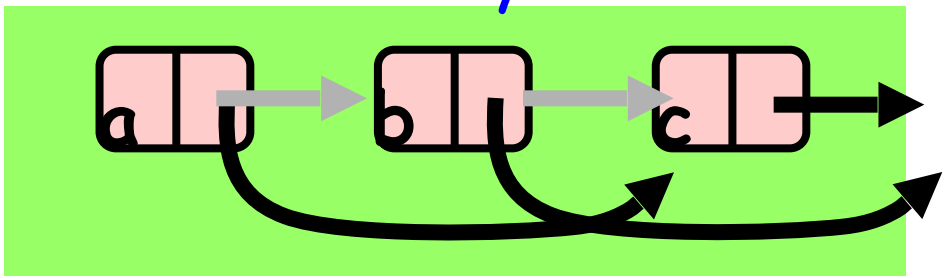




Problem

- To delete node c
  - Swing node b's next field to d
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c

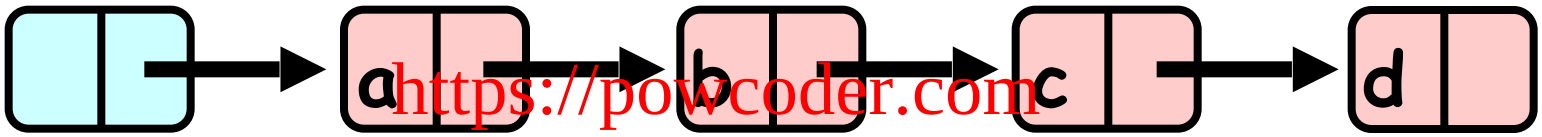


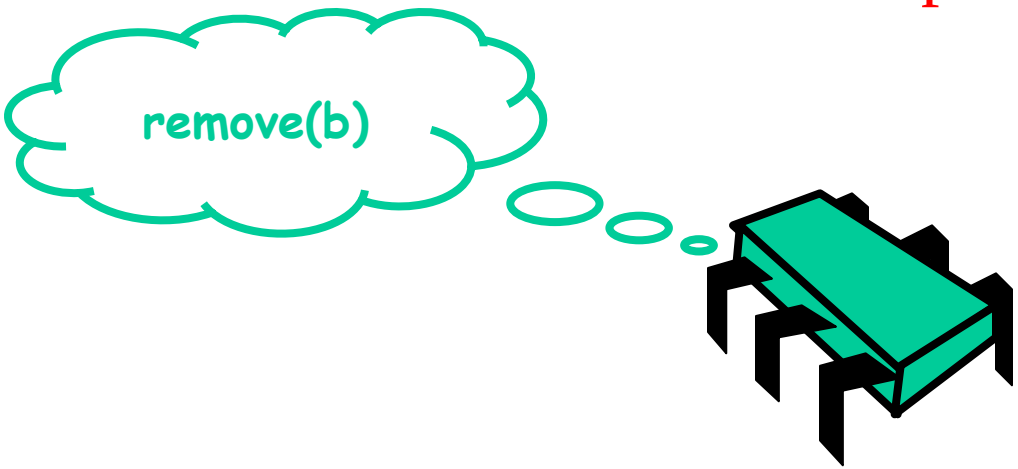


Insight

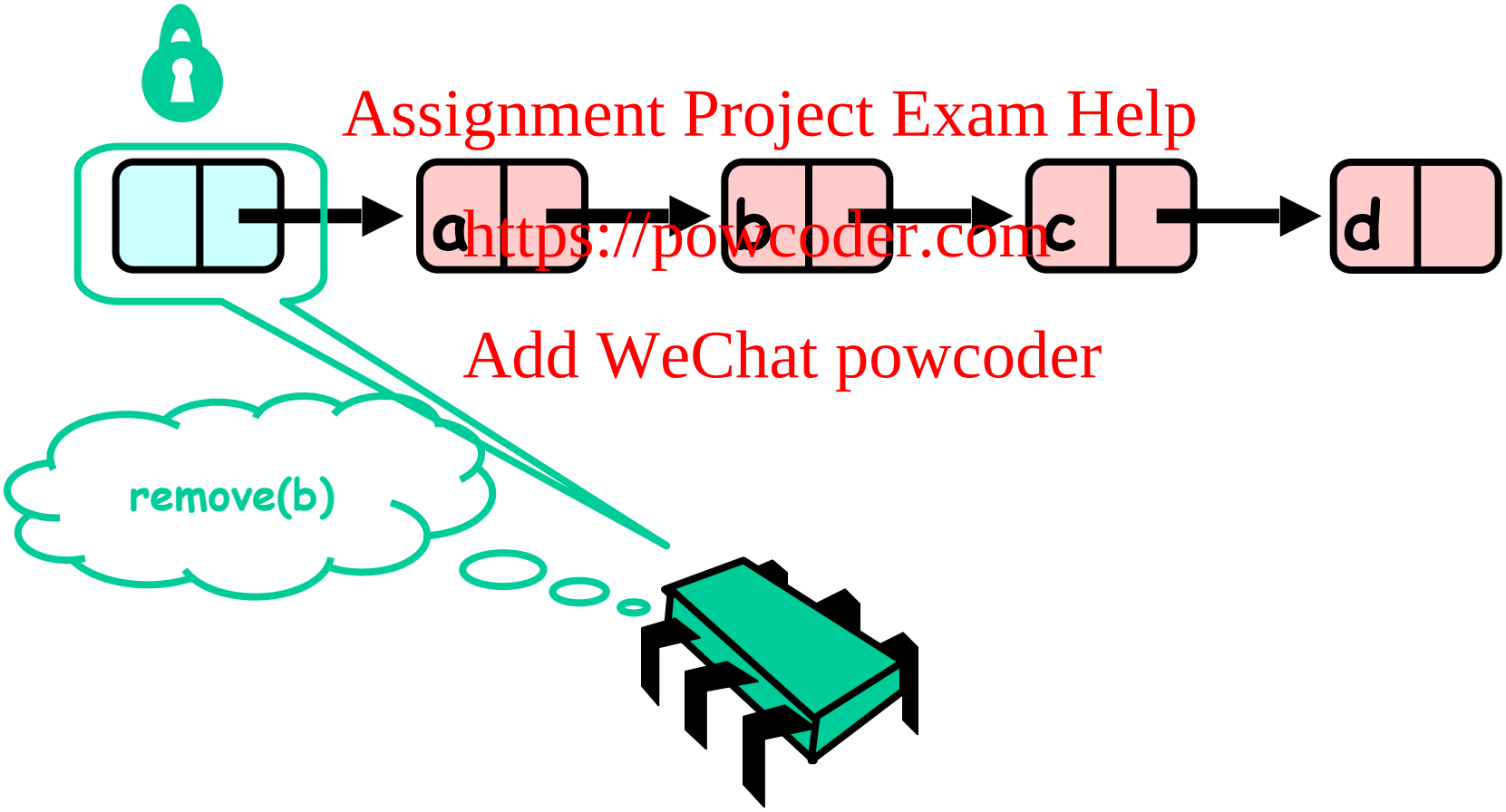
- If a node is locked
  - No one can delete node's successor
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

Hand-Over-Hand Again

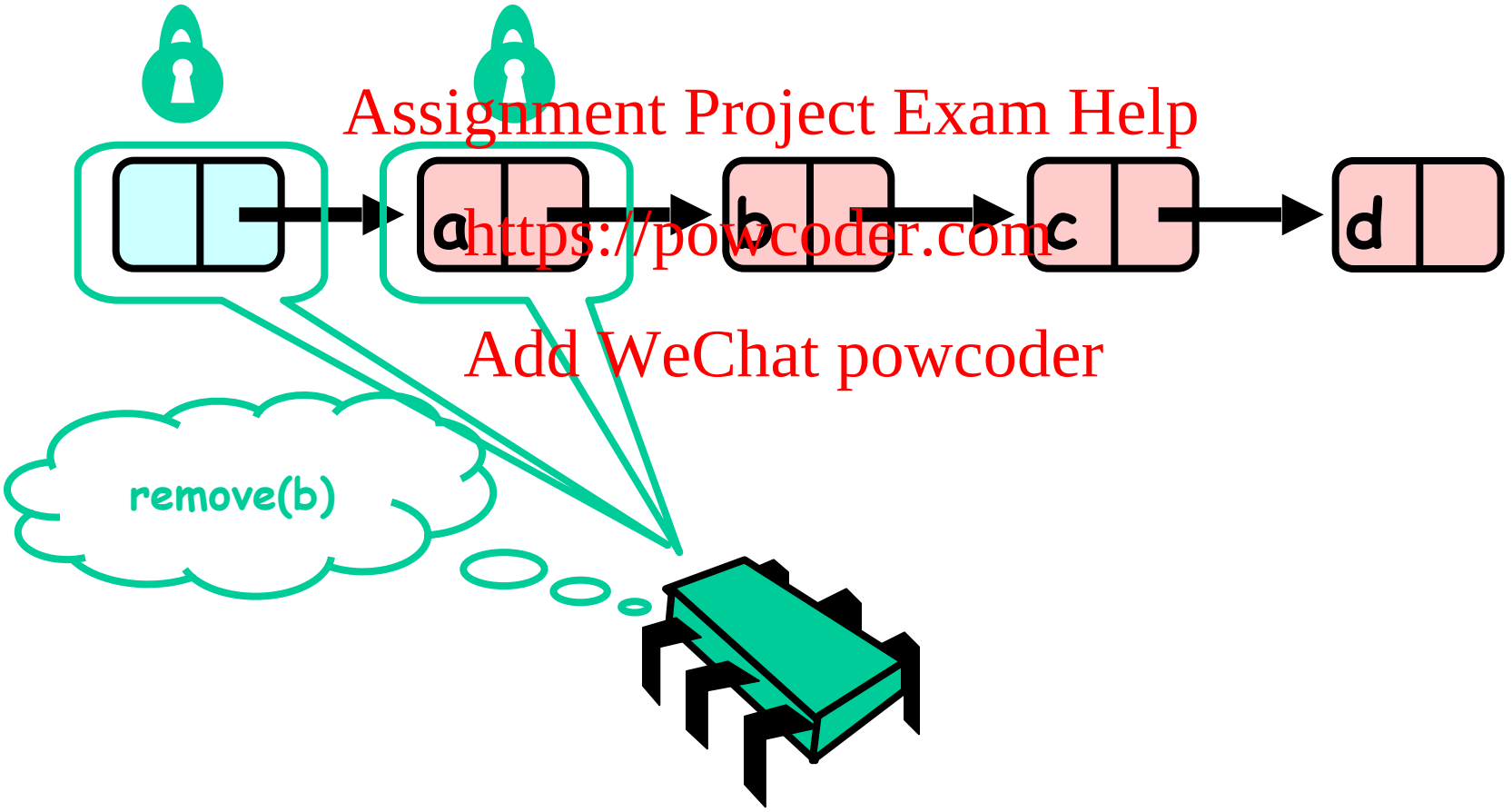




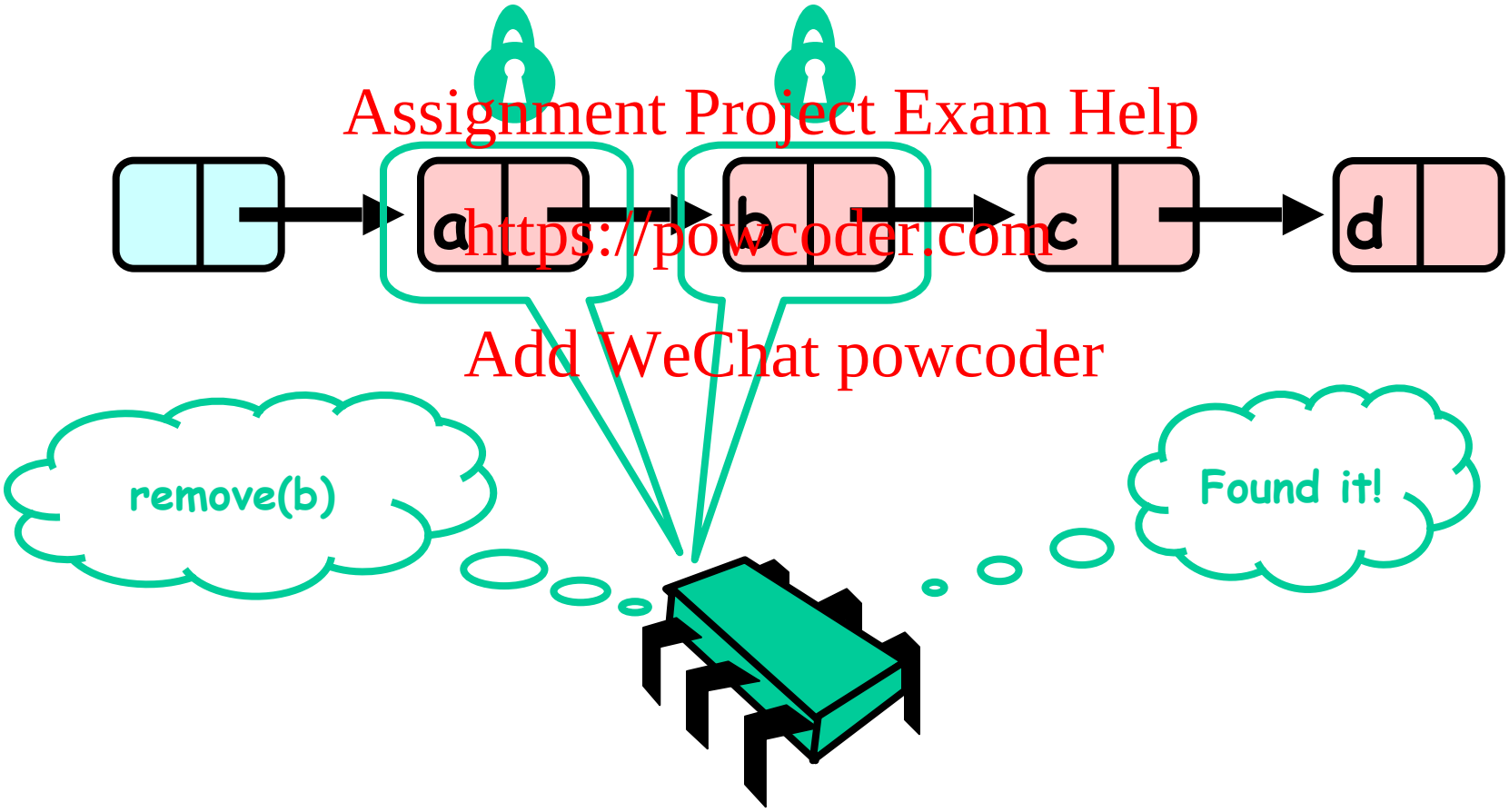
Hand-Over-Hand Again



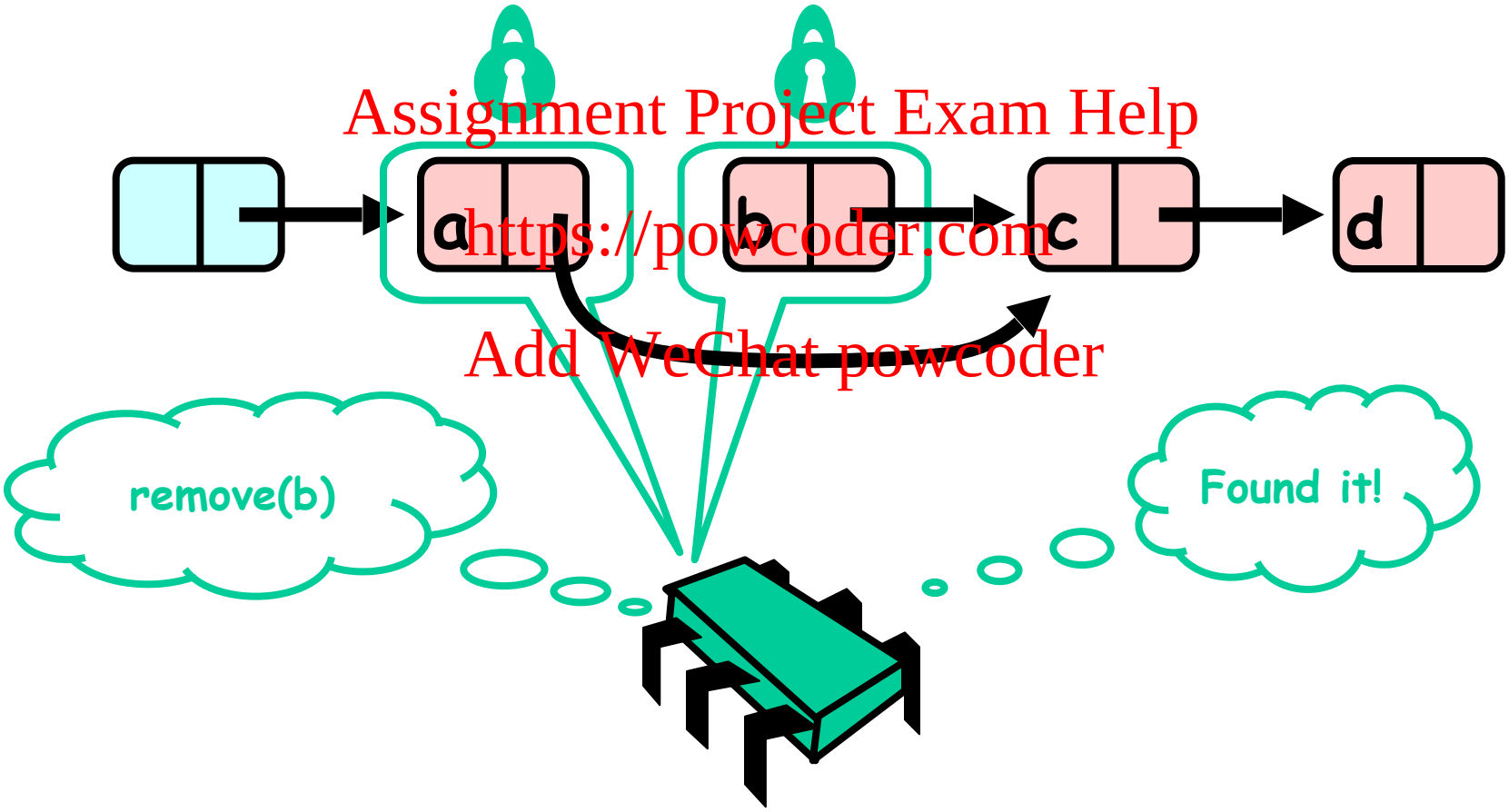
Hand-Over-Hand Again



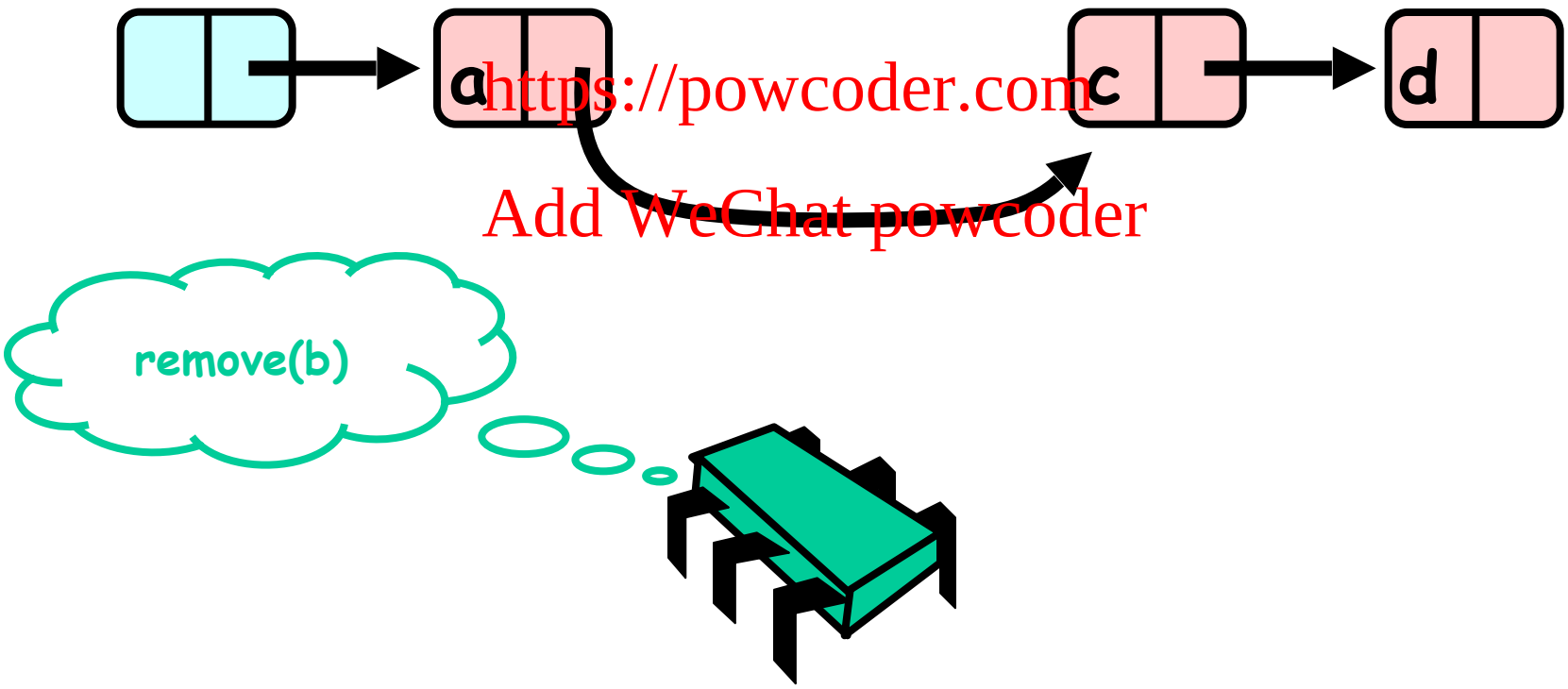
Hand-Over-Hand Again



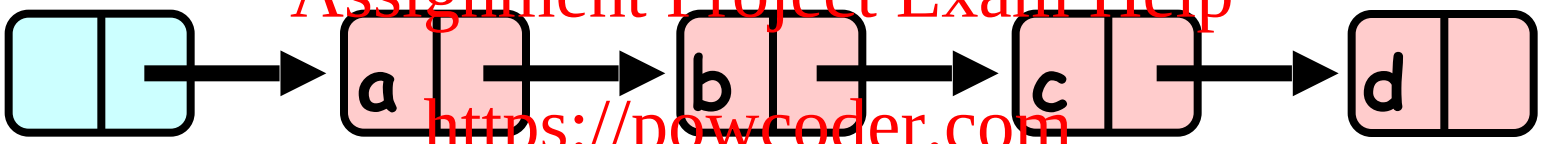
Hand-Over-Hand Again

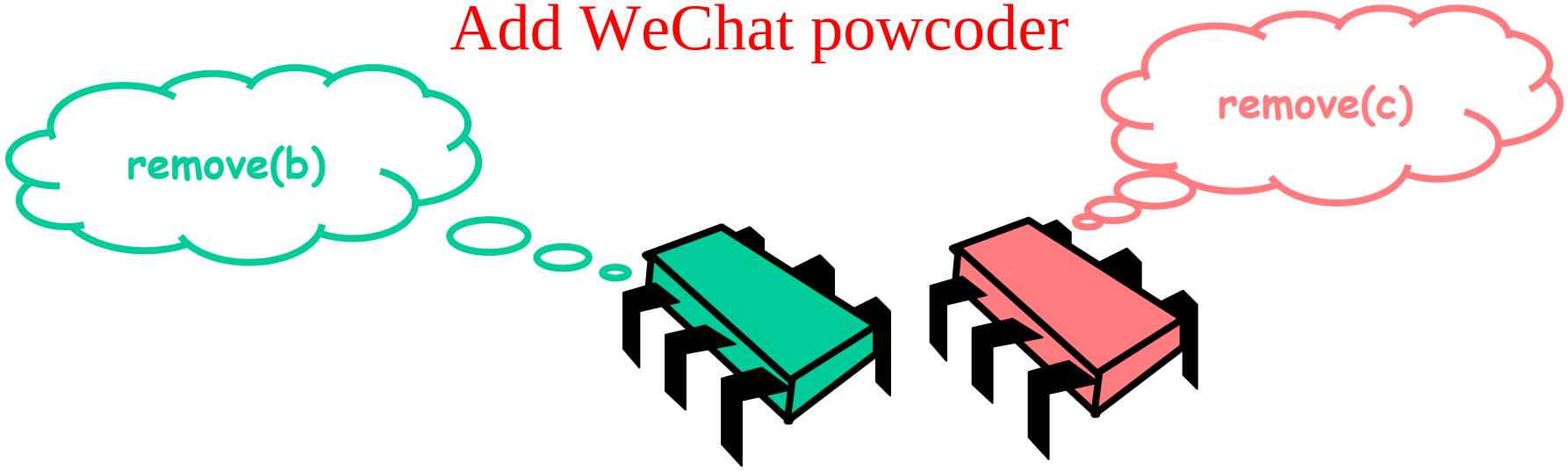


Hand-Over-Hand Again

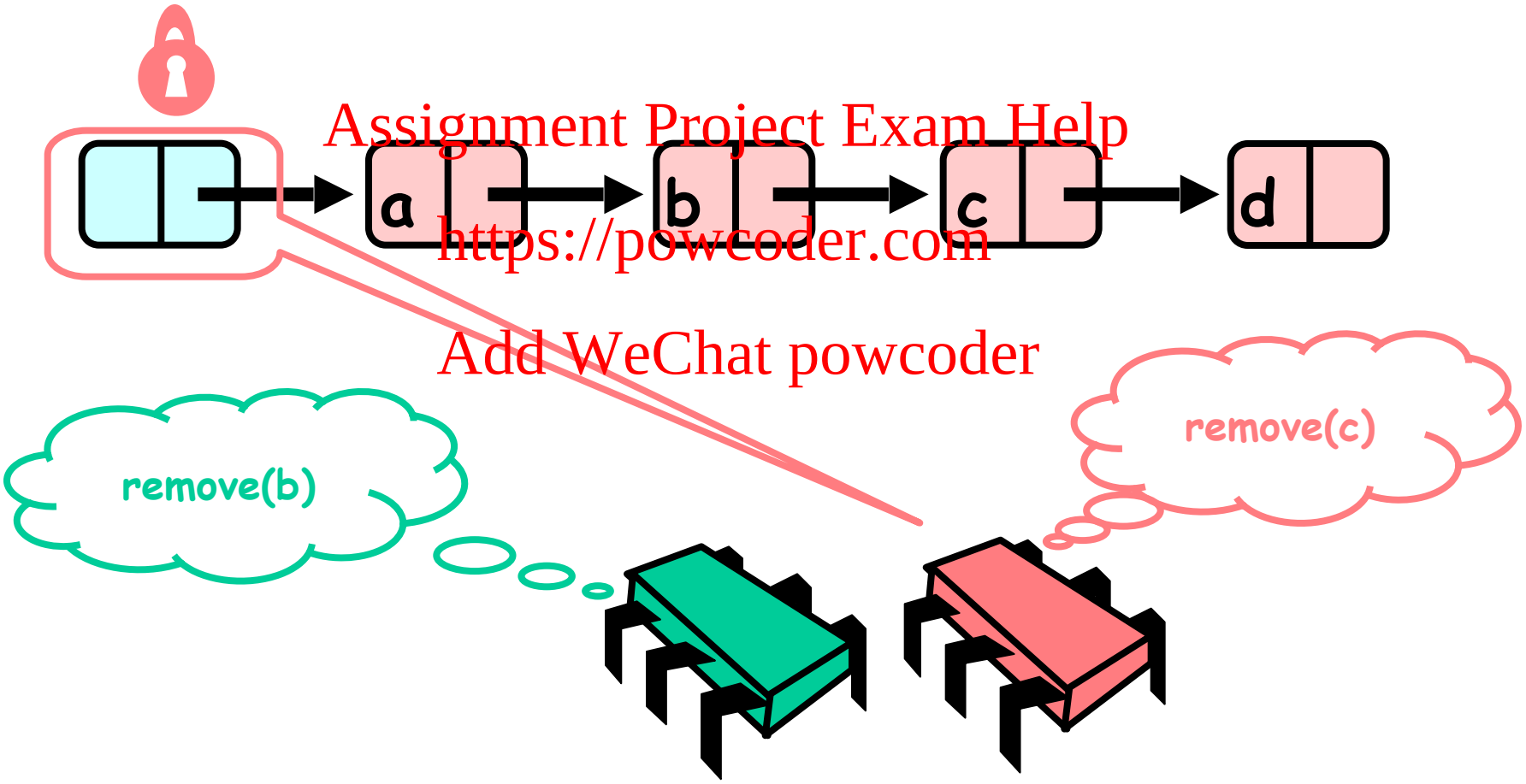


Removing a Node

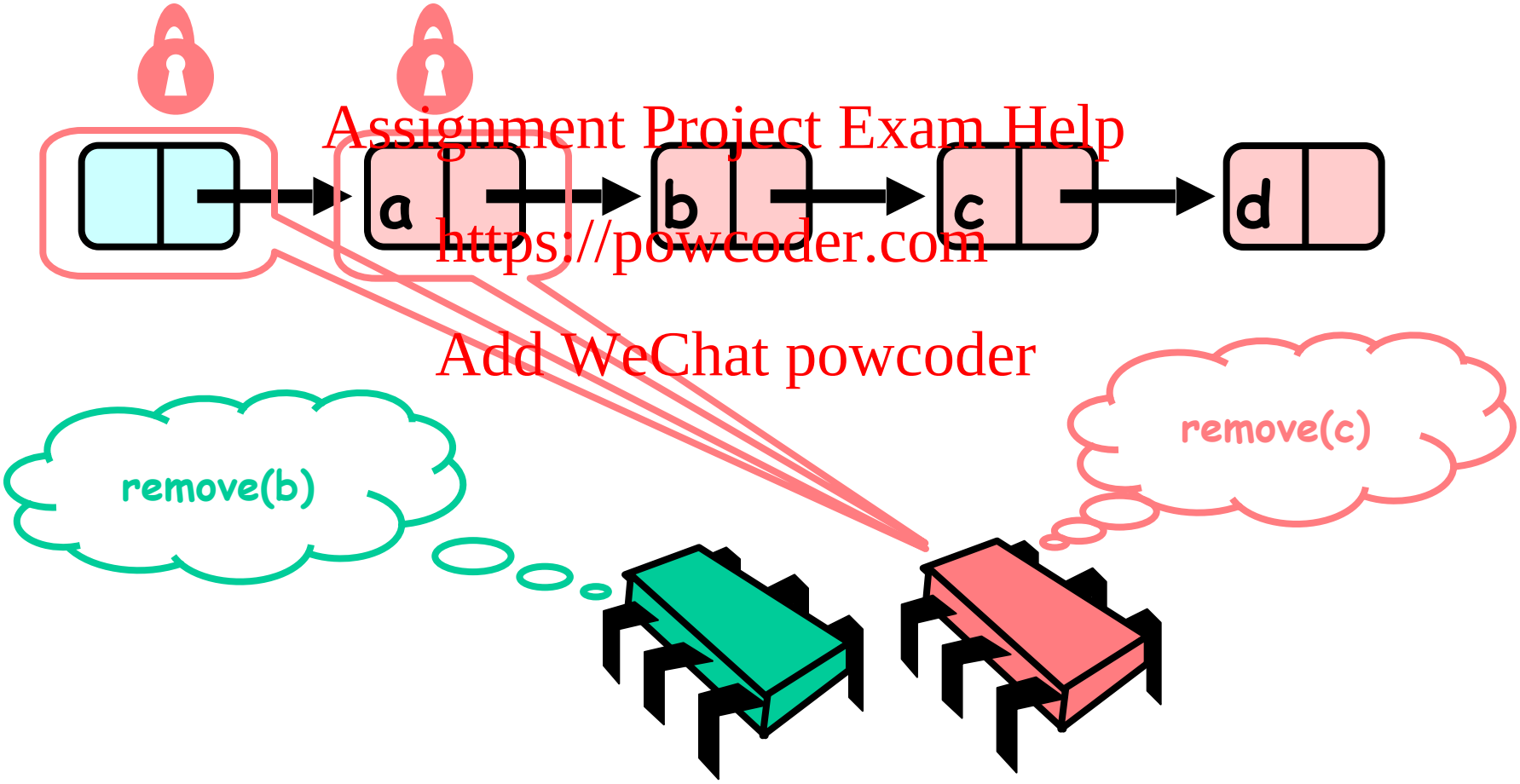




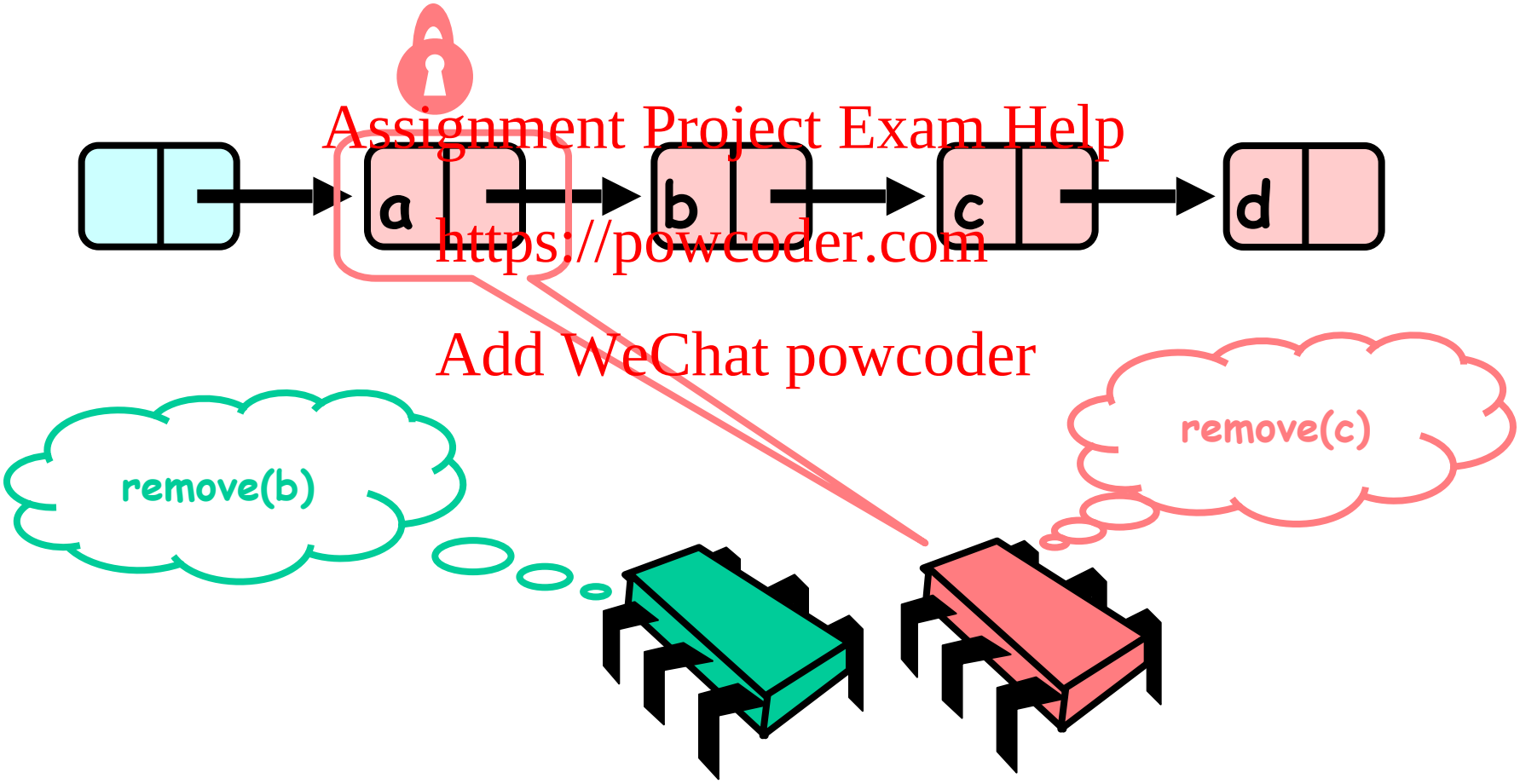
Removing a Node



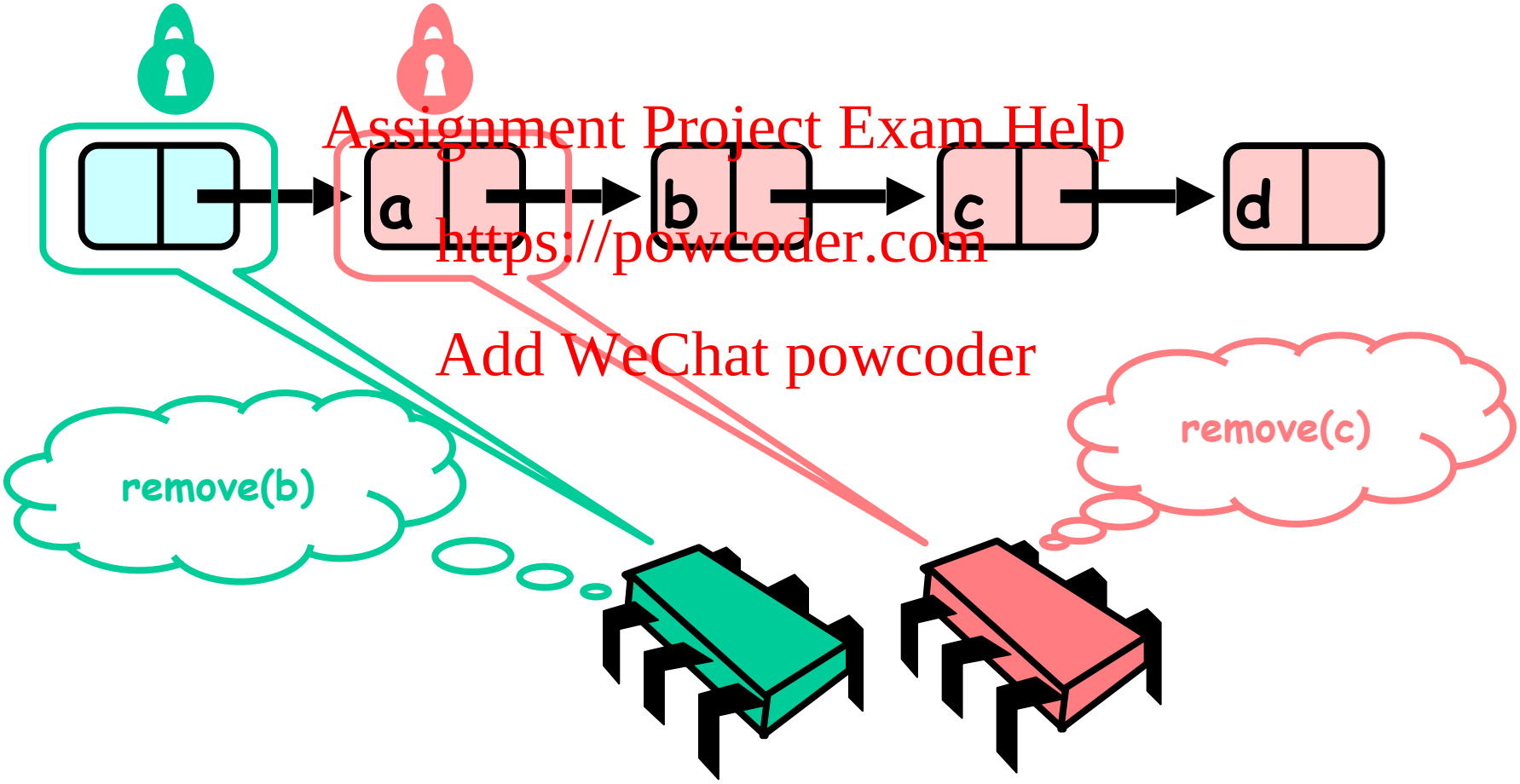
Removing a Node



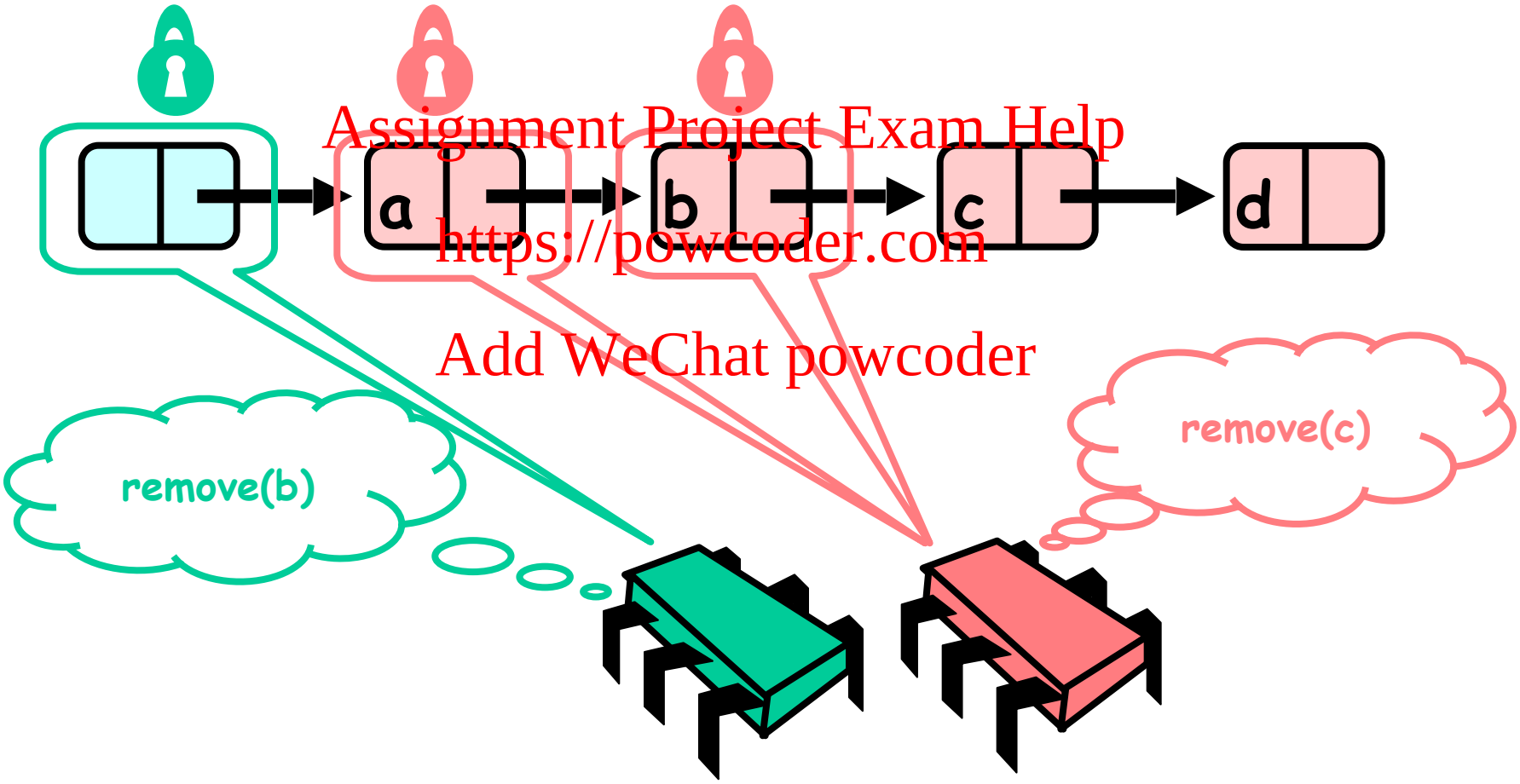
Removing a Node



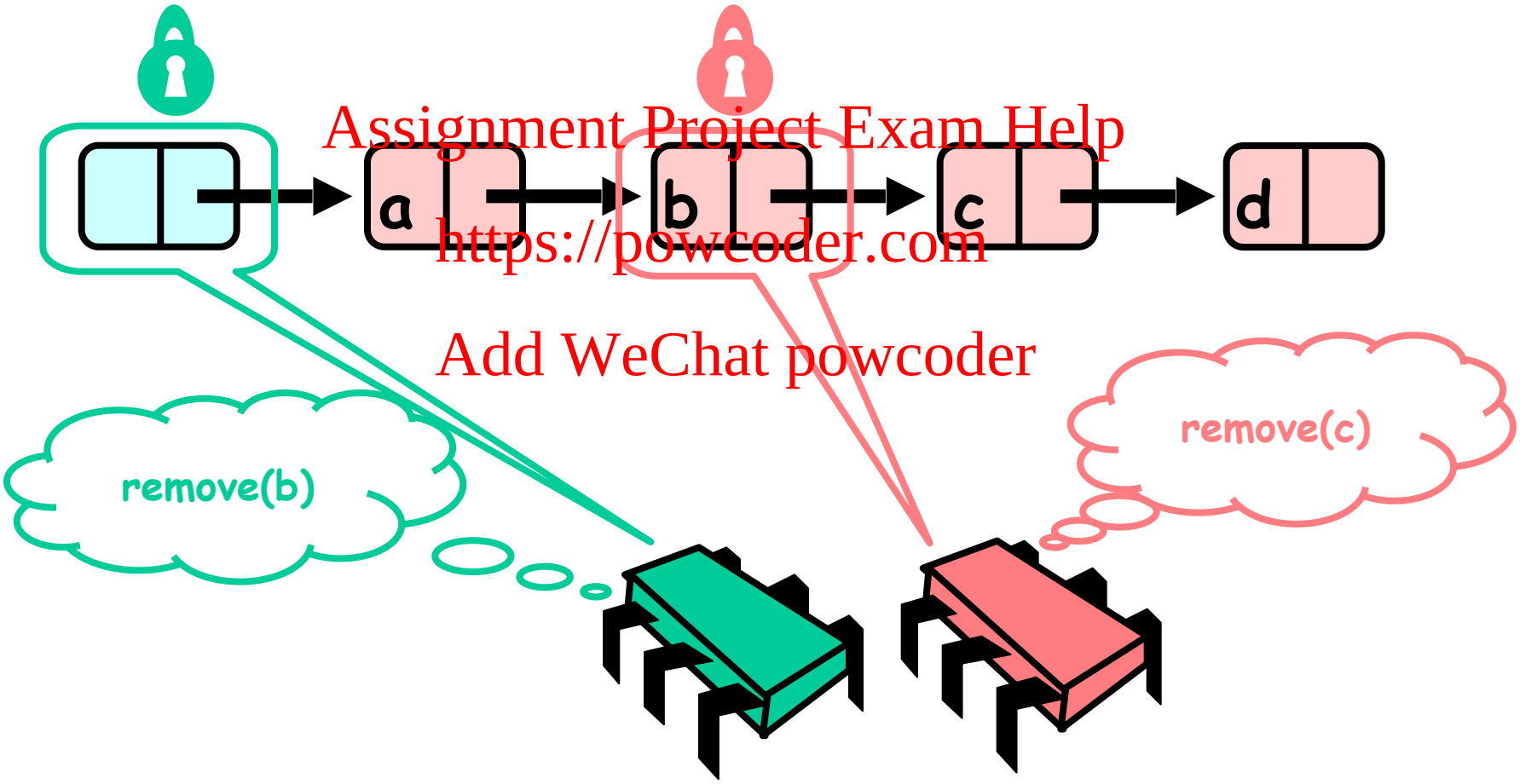
Removing a Node



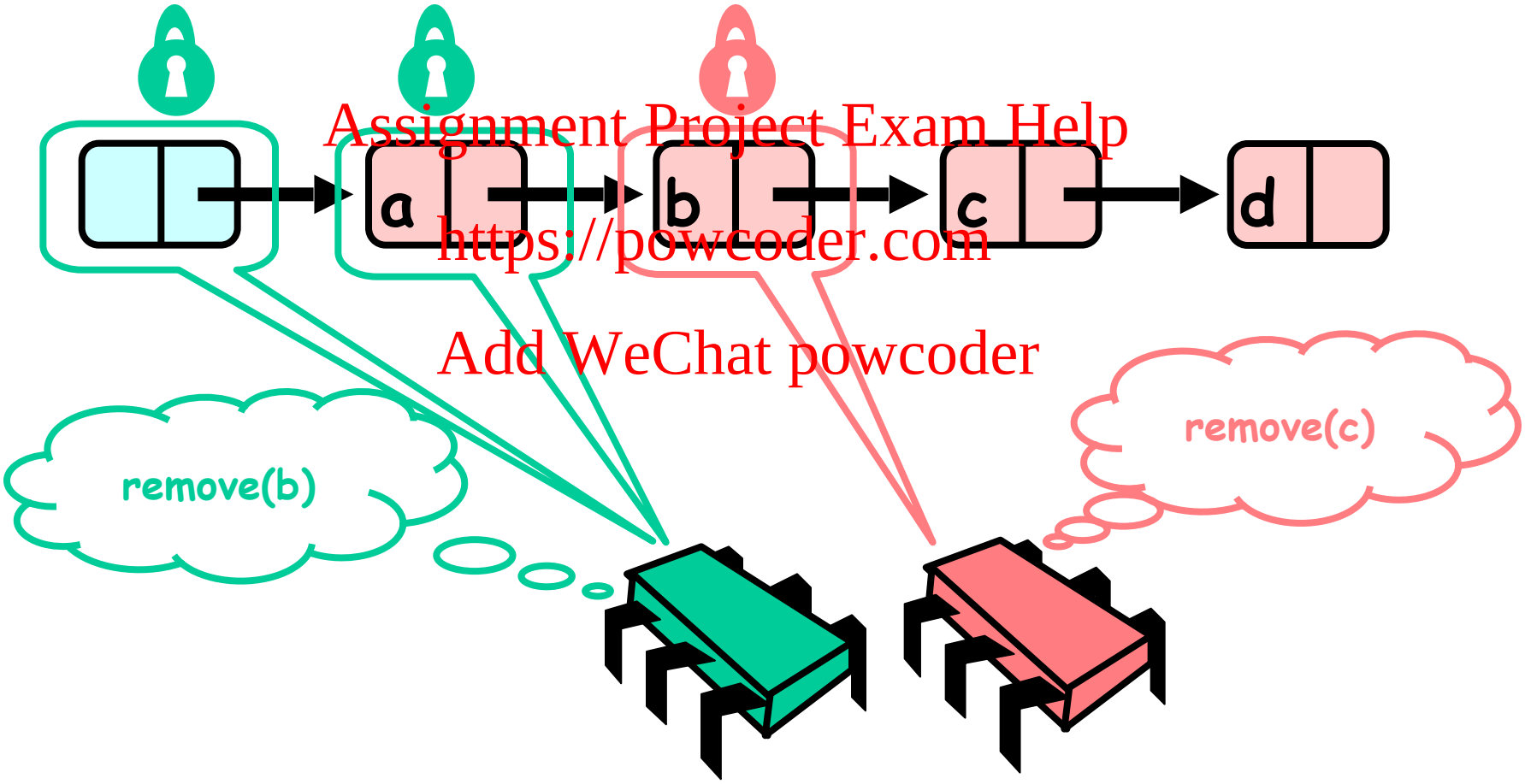
Removing a Node



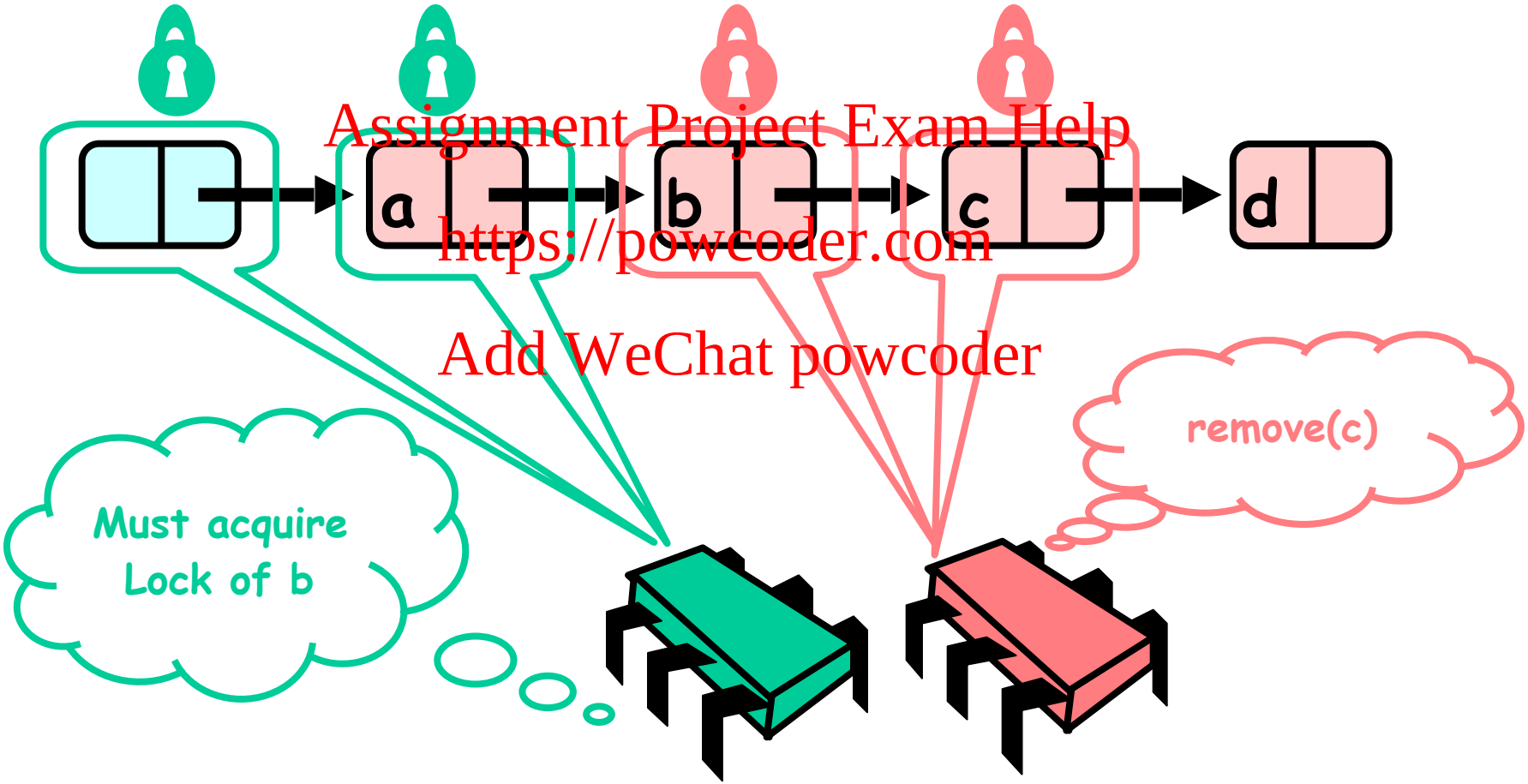
Removing a Node



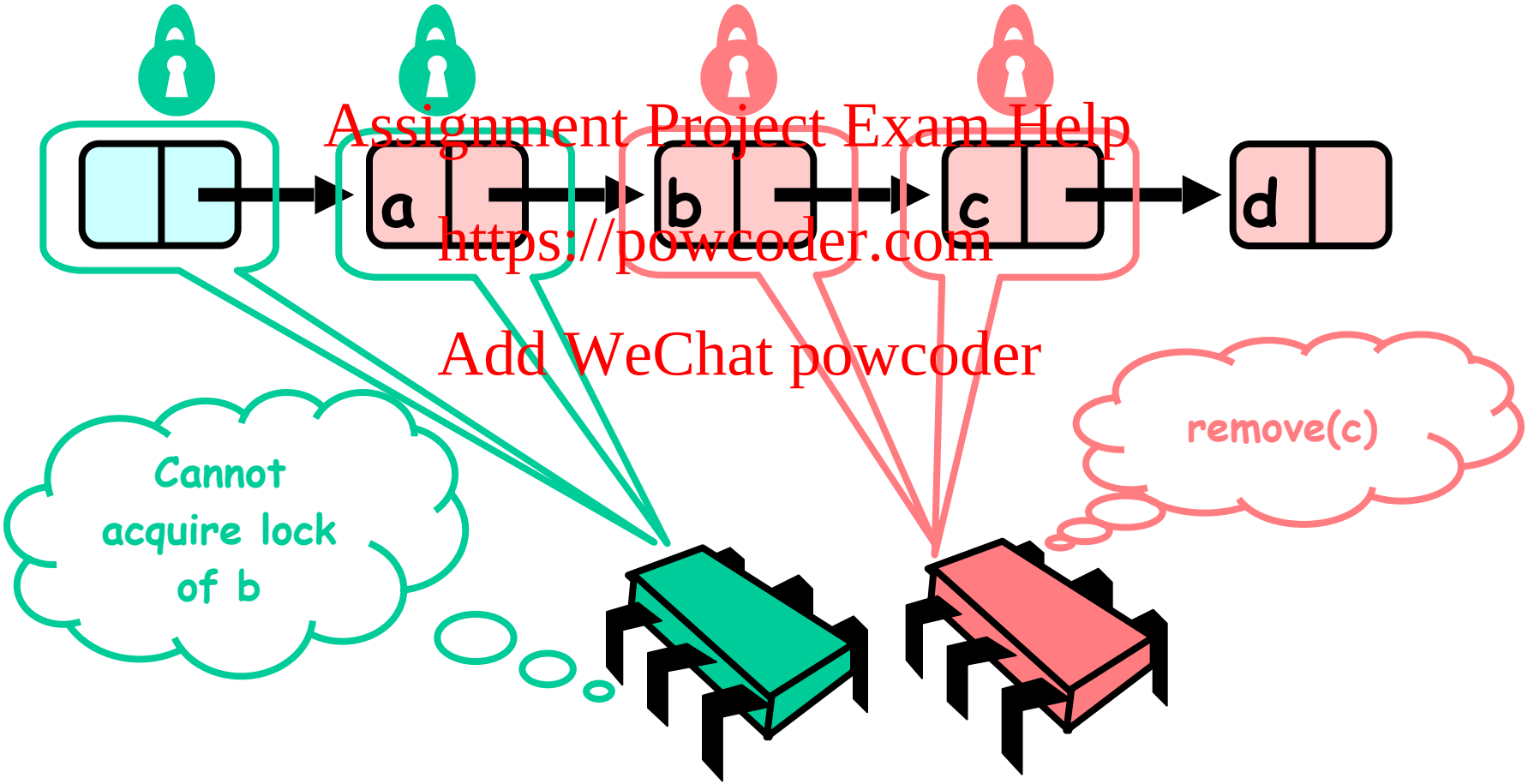
Removing a Node



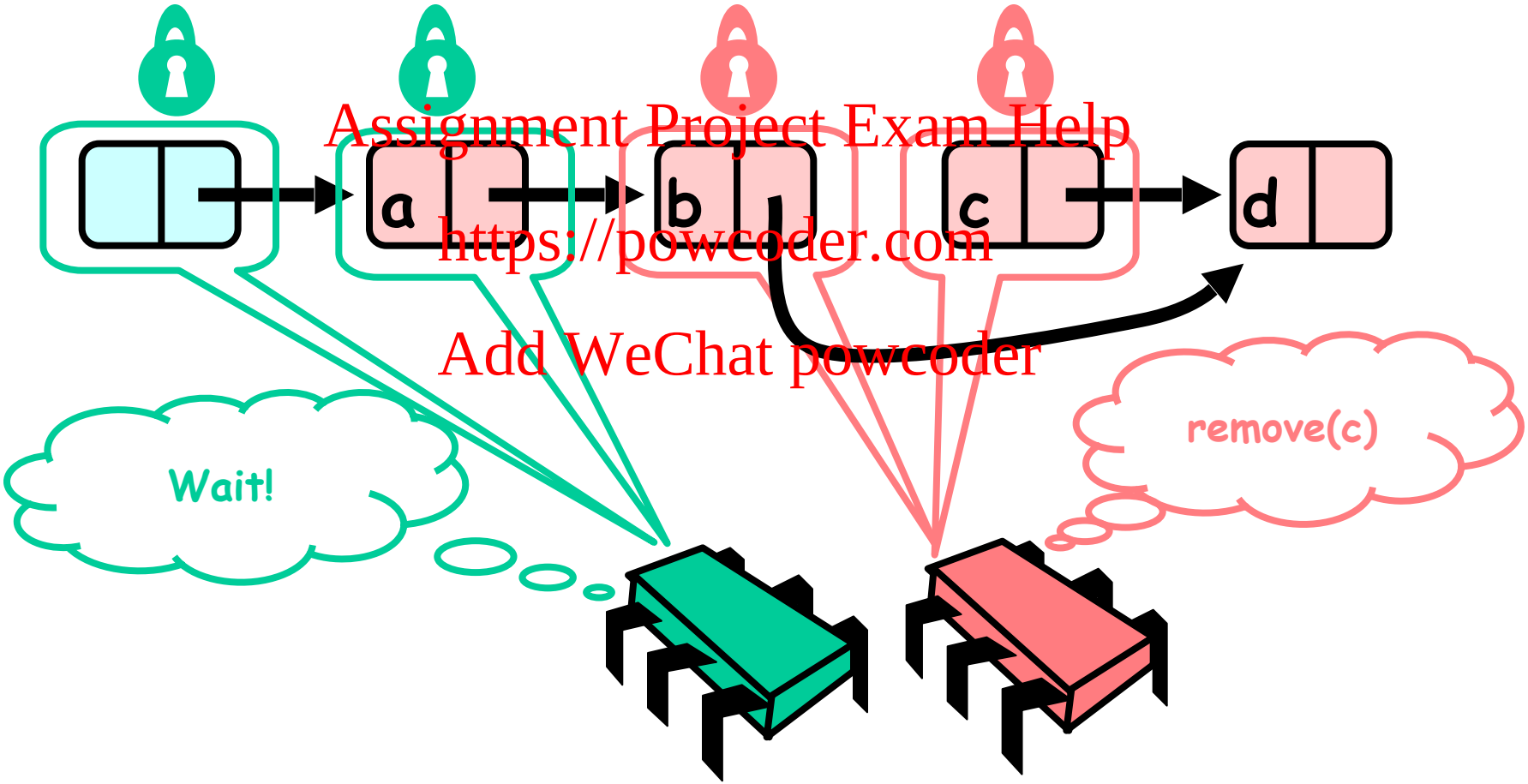
Removing a Node



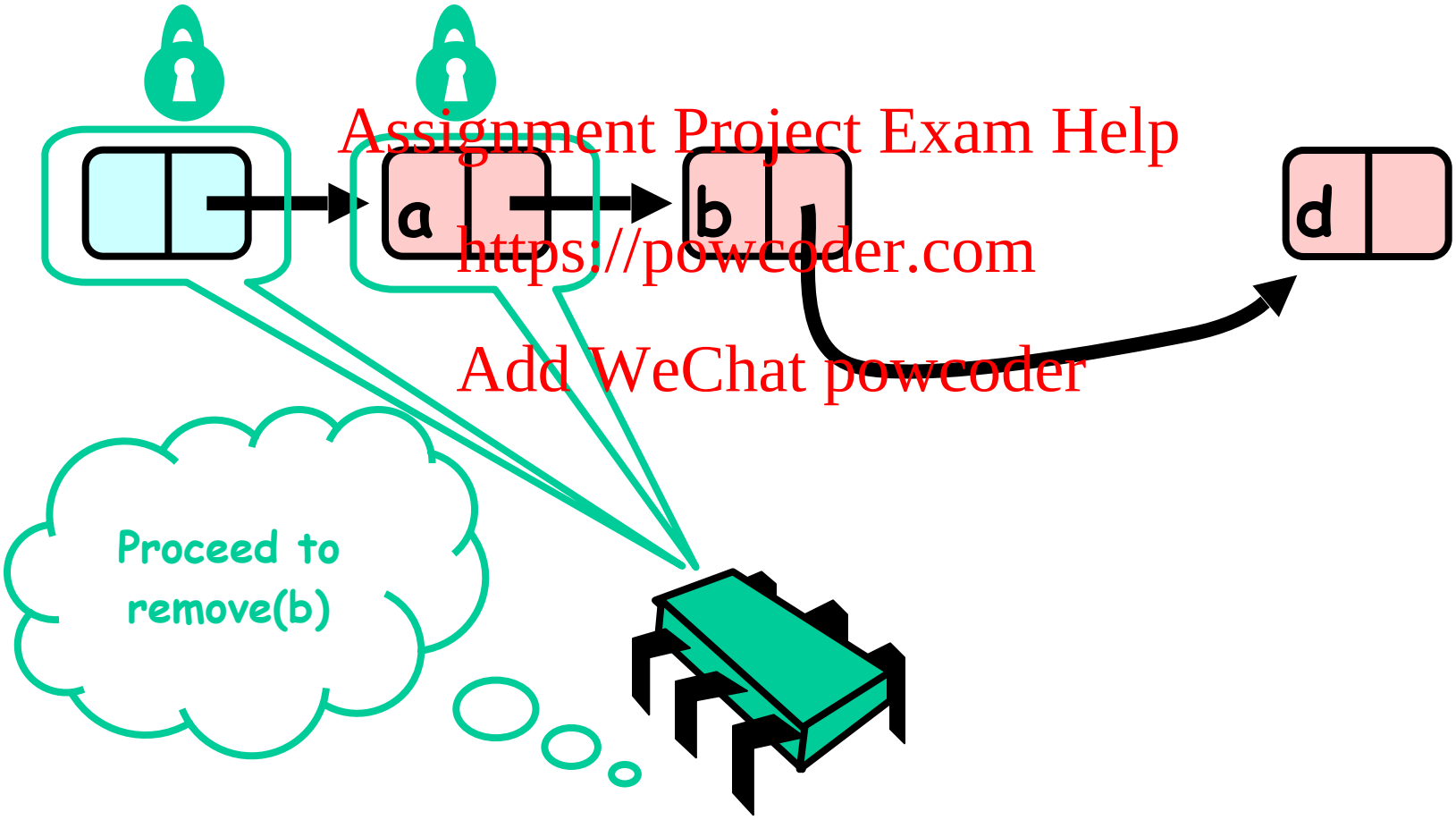
Removing a Node



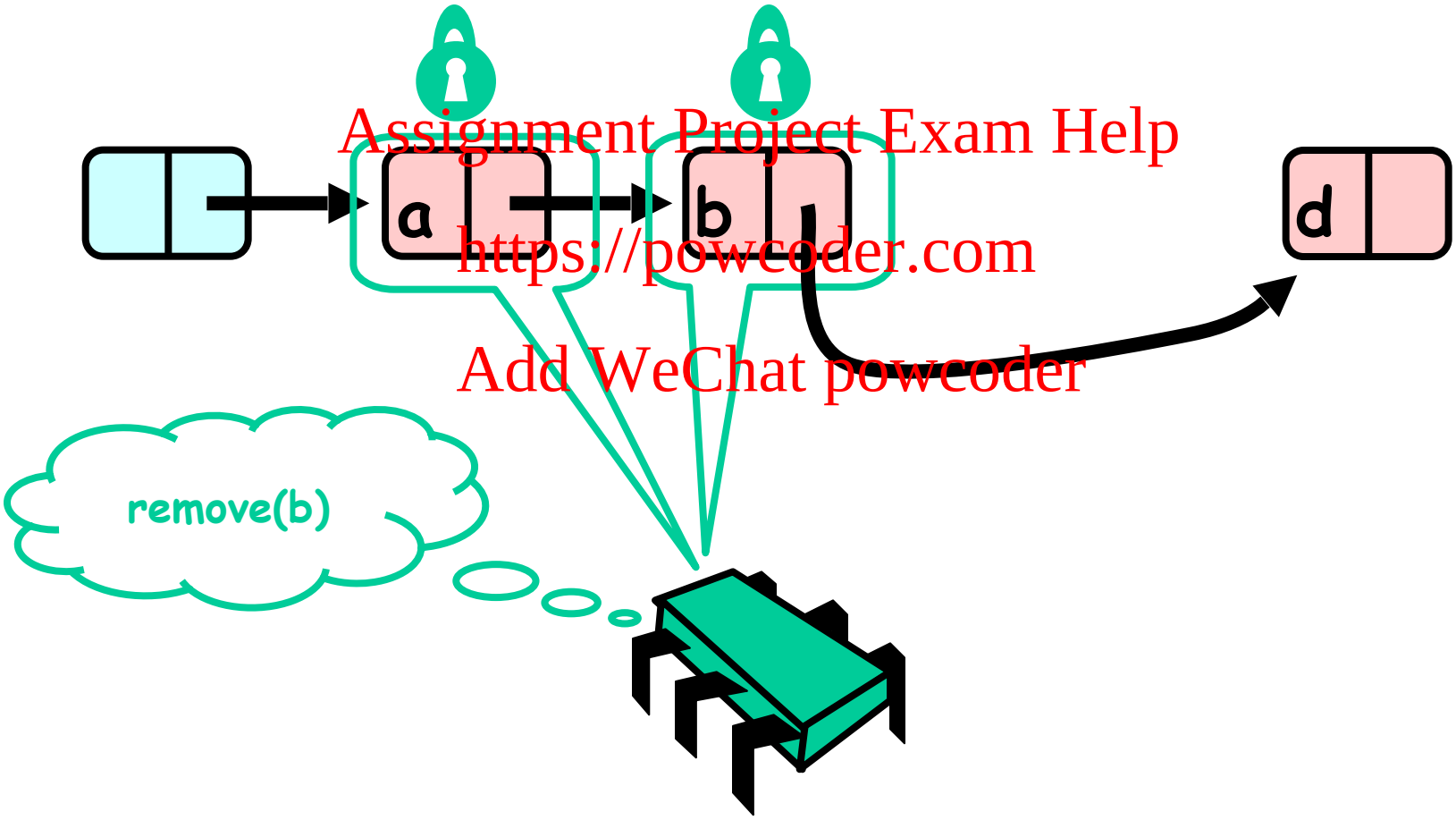
Removing a Node



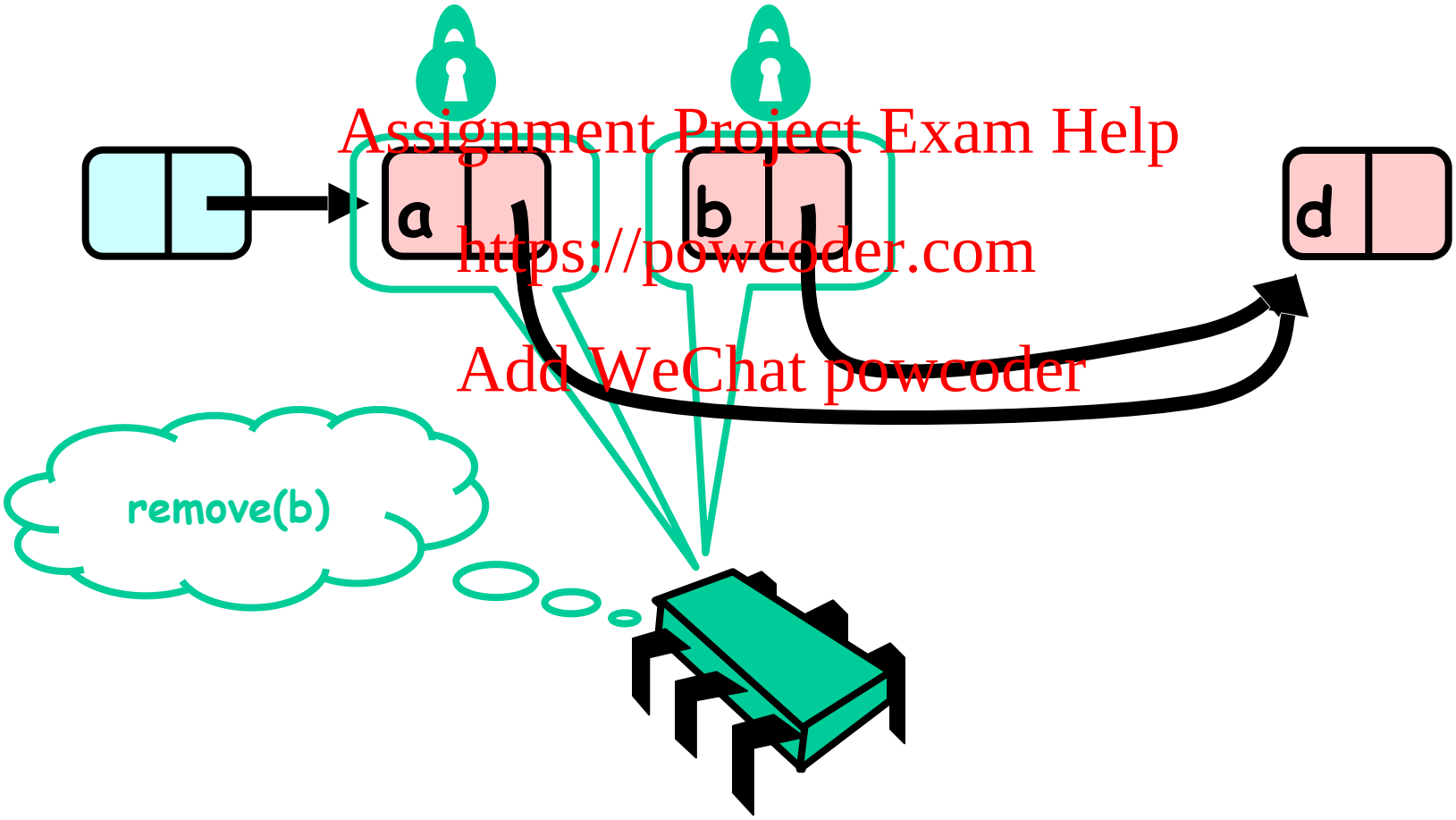
Removing a Node



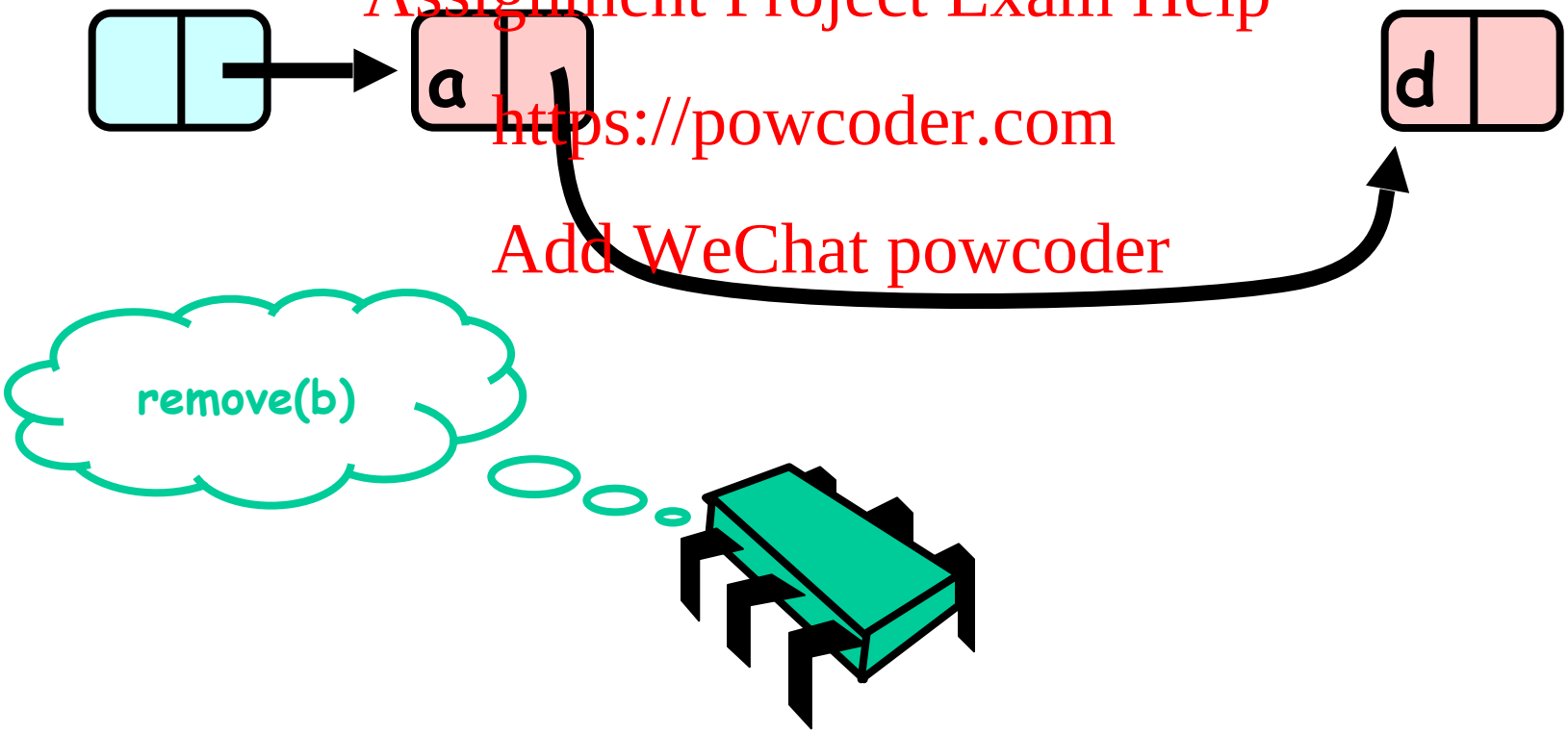
Removing a Node



Removing a Node



Removing a Node



Removing a Node

Fine-Grained Synchronization: hand-over-hand locking Linked List

```
public boolean add(T item) {
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    try {
      Node curr = pred.next;
      curr.lock();
      try {
        while (curr.key < key) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        if (curr.key == key) return false;
        Node newNode = new Node(item);
        newNode.next = curr;
        pred.next = newNode;
        return true;
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

```
public boolean remove(T item) {
   Node pred = null, curr = null;
   int key = item.hashCode();
   head.lock();
   try {
     pred = head;
     curr = pred.next;
     curr.lock();
     try {
       while (curr.key < key) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (curr.key == key) {
         pred.next = curr.next;
         return true;
       }
       return false;
     } finally {
       curr.unlock();
     }
   } finally {
     pred.unlock();
   }
 }
```

```
public boolean contains(T item) {
   Node last = null, pred = null, curr
  = null;
   int key = item.hashCode();
   head.lock();
   try {
     pred = head;
     curr = pred.next;
     curr.lock();
     try {
       while (curr.key < key) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       return (curr.key == key);
     } finally {
       curr.unlock();
     }
   } finally {
     pred.unlock();
   }
  }
}
```

Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted

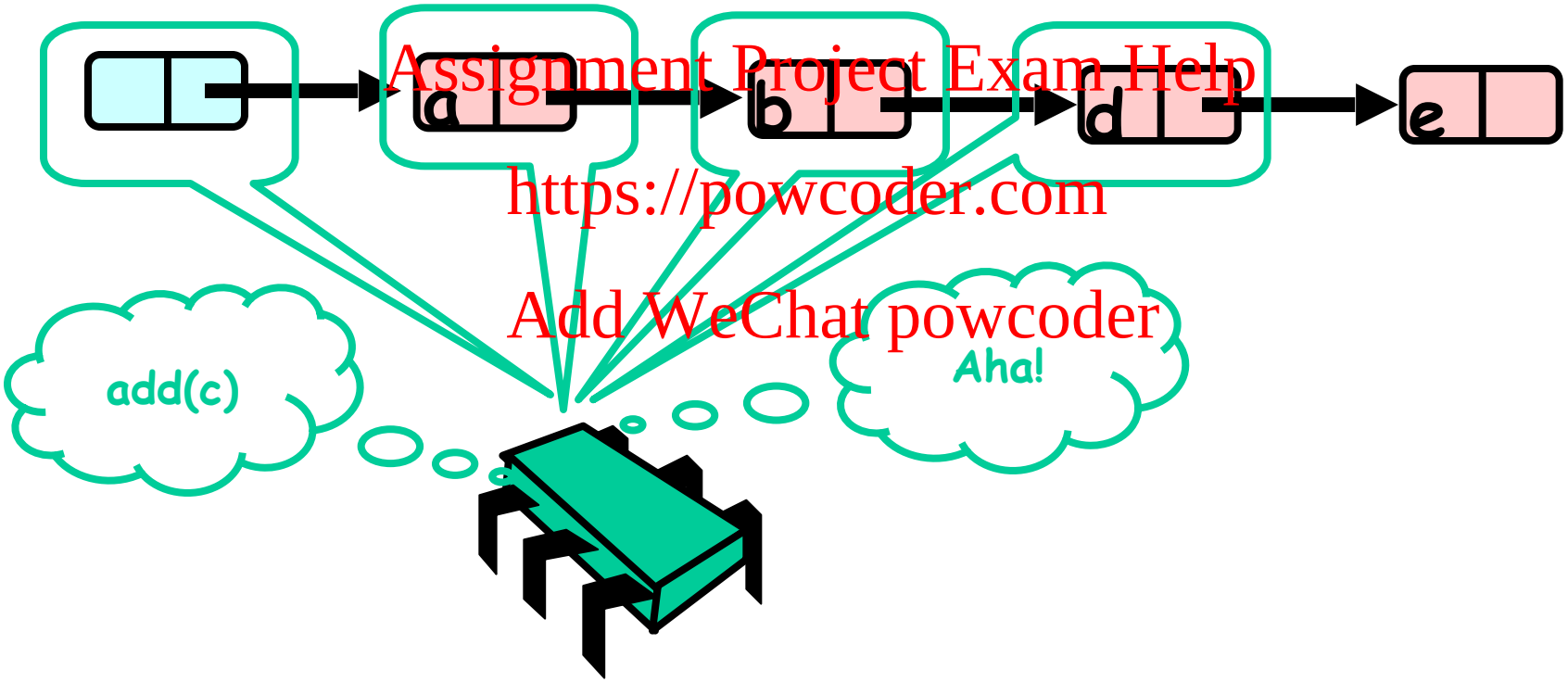
Drawbacks

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

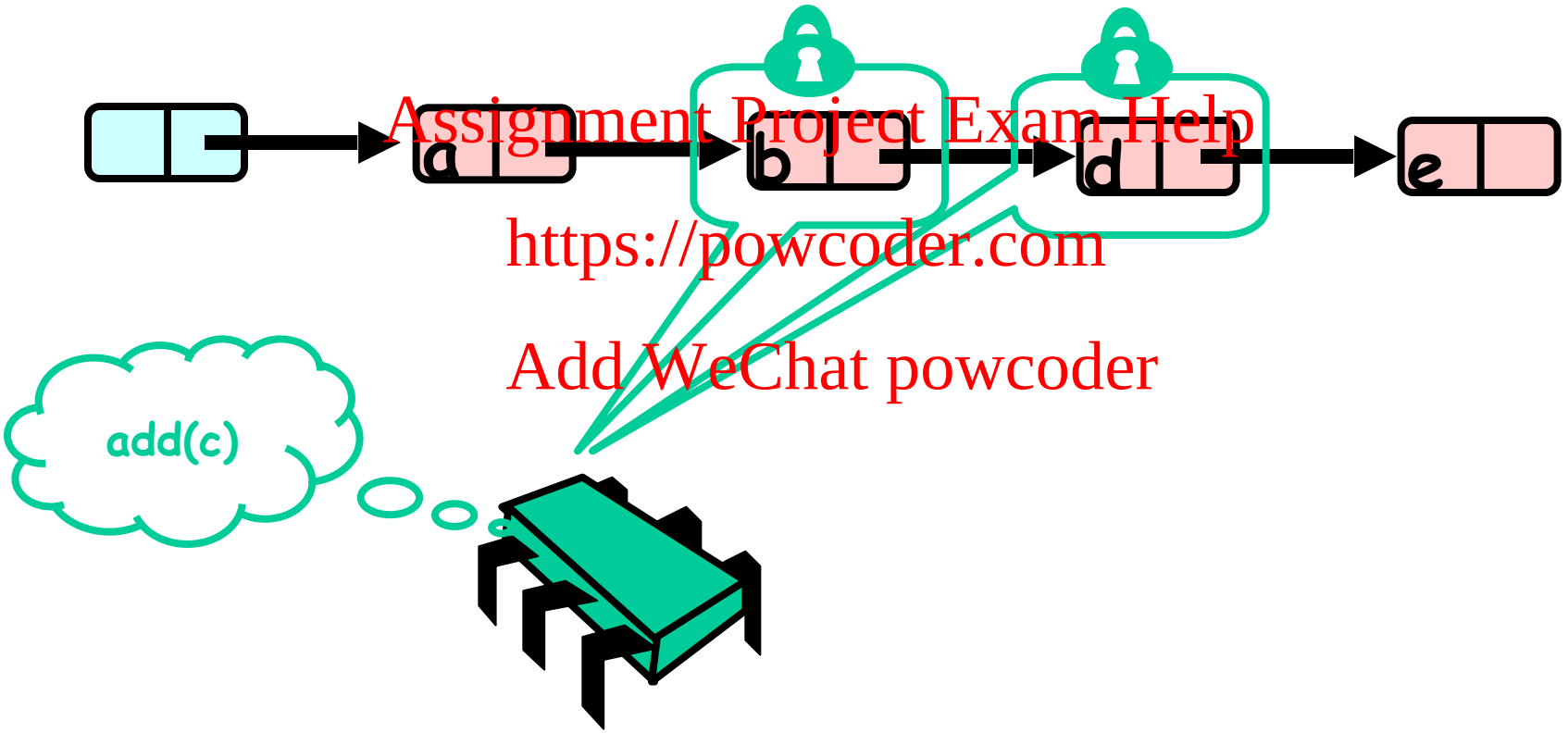
Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK

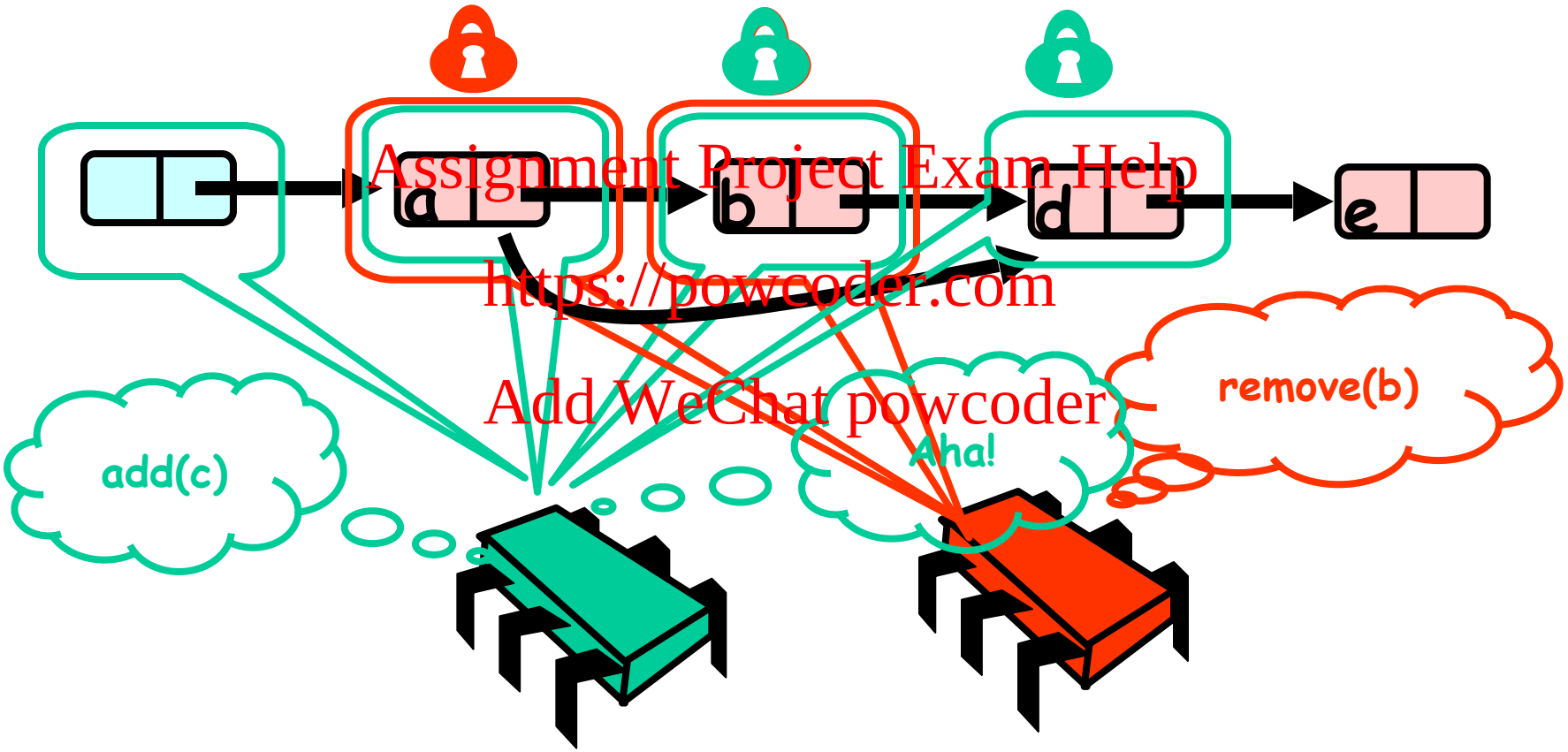
Optimistic: Traverse without Locking



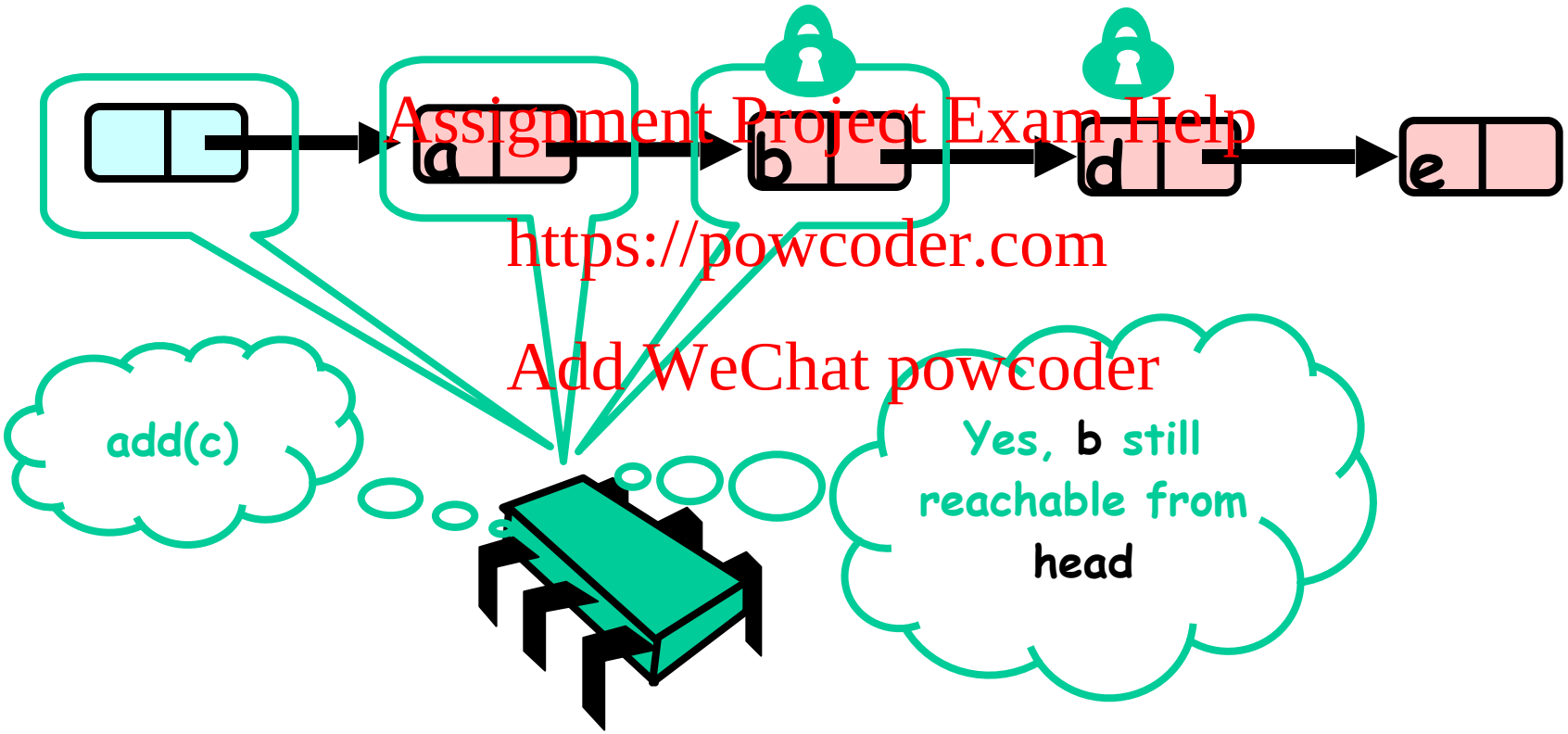
Optimistic: Lock and Load



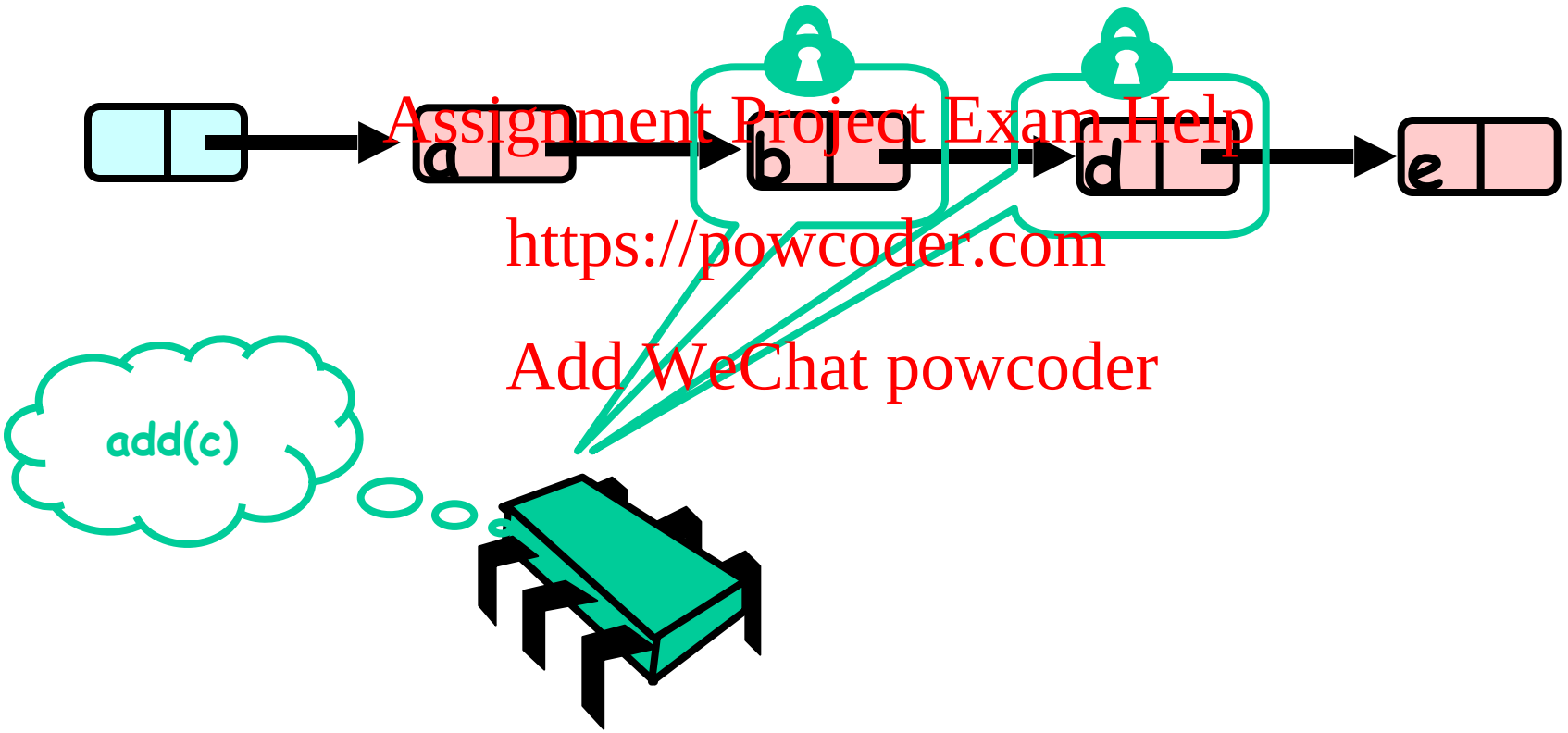
What could go wrong?



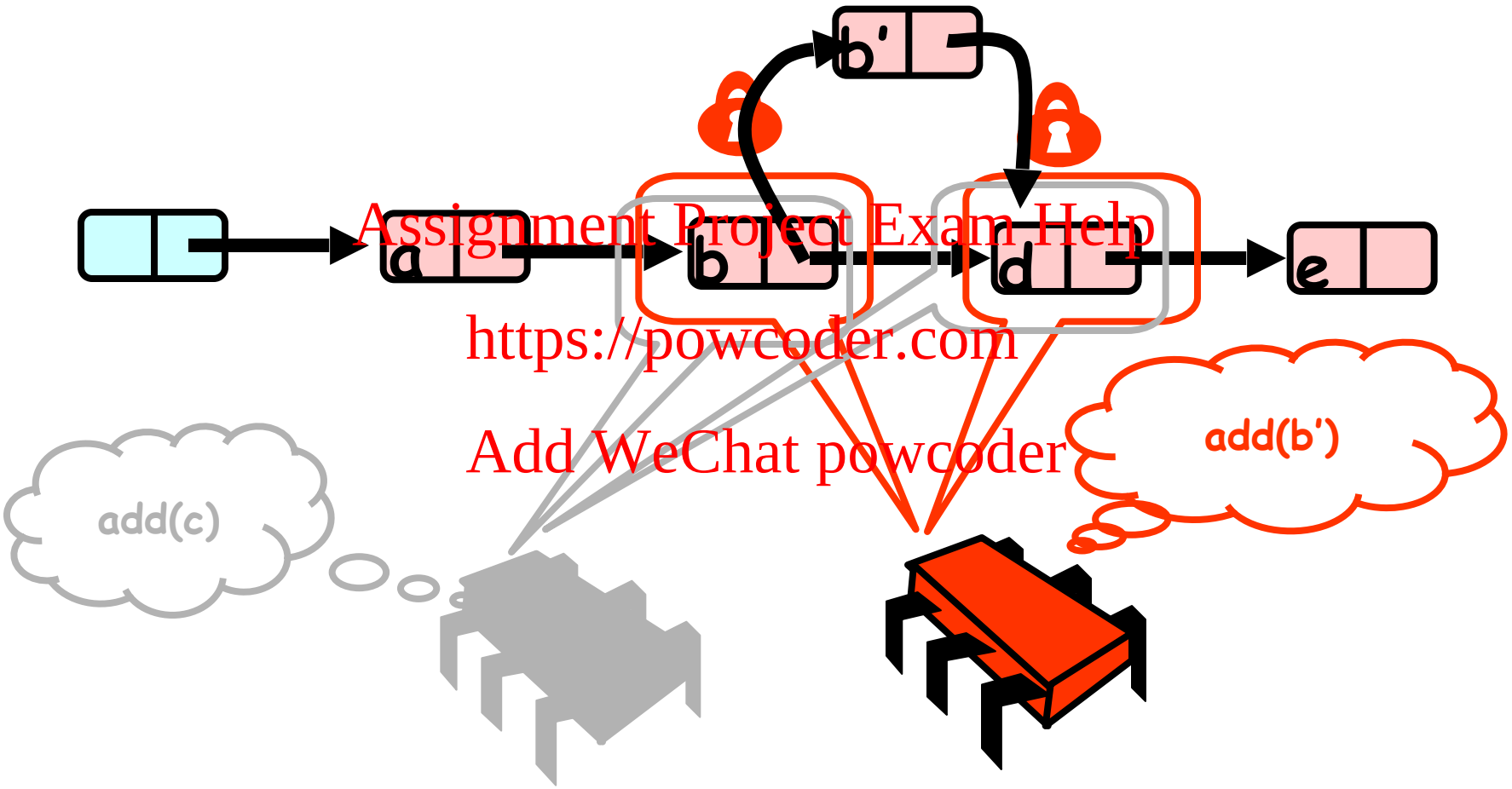
Validate – Part 1 (while holding locks)



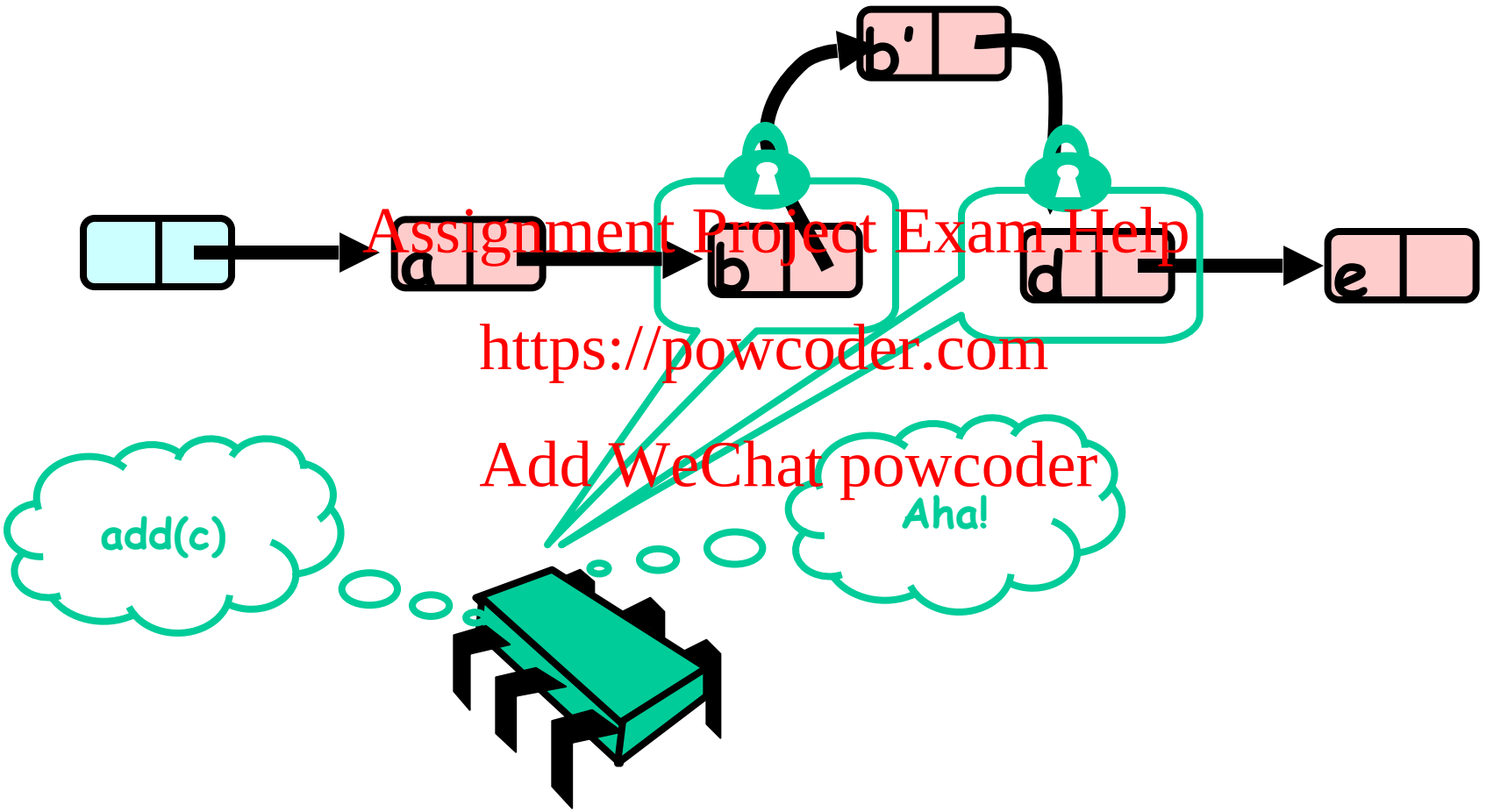
What Else Can Go Wrong?



What Else Can Go Wrong?

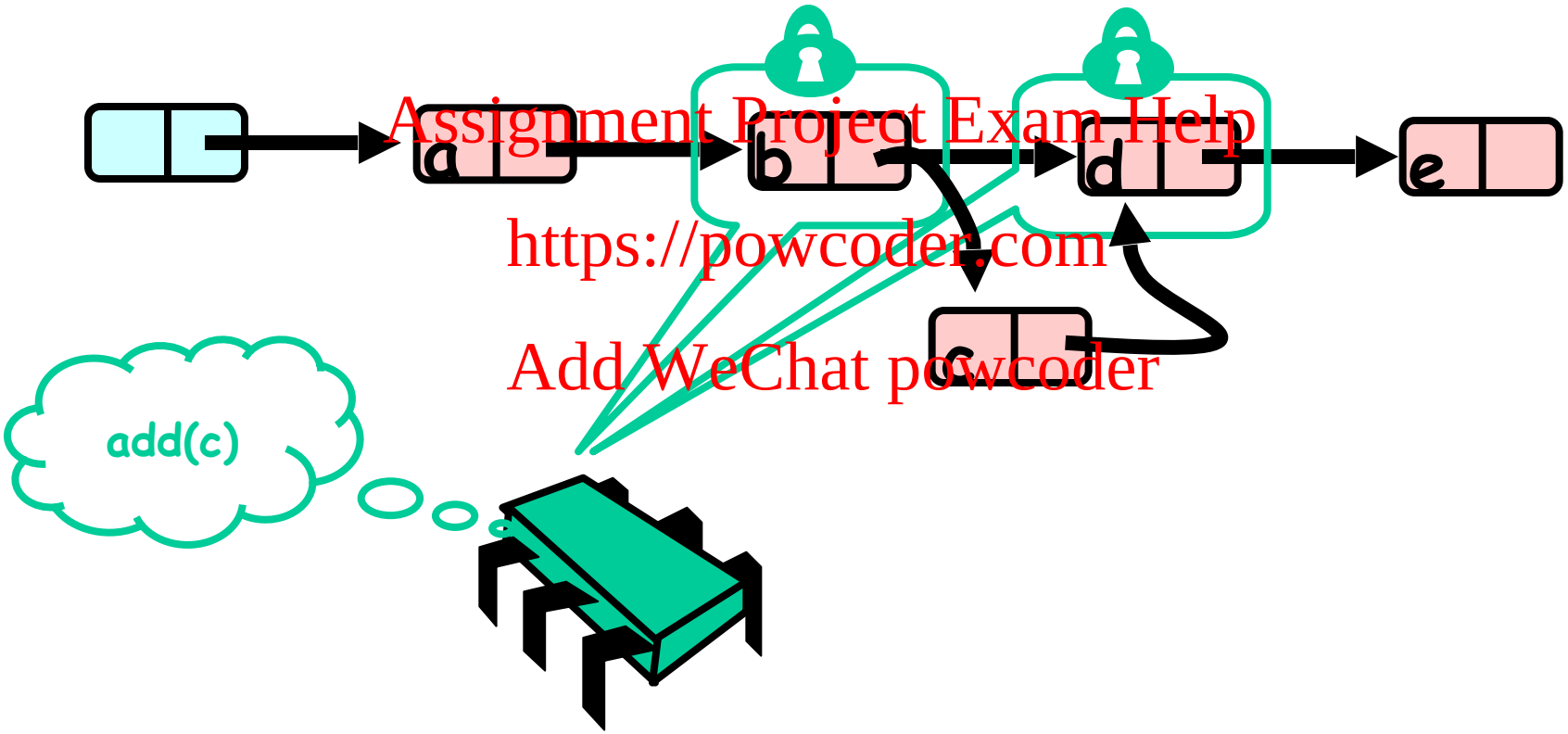


What Else Can Go Wrong?



Validate Part 2 (while holding locks)

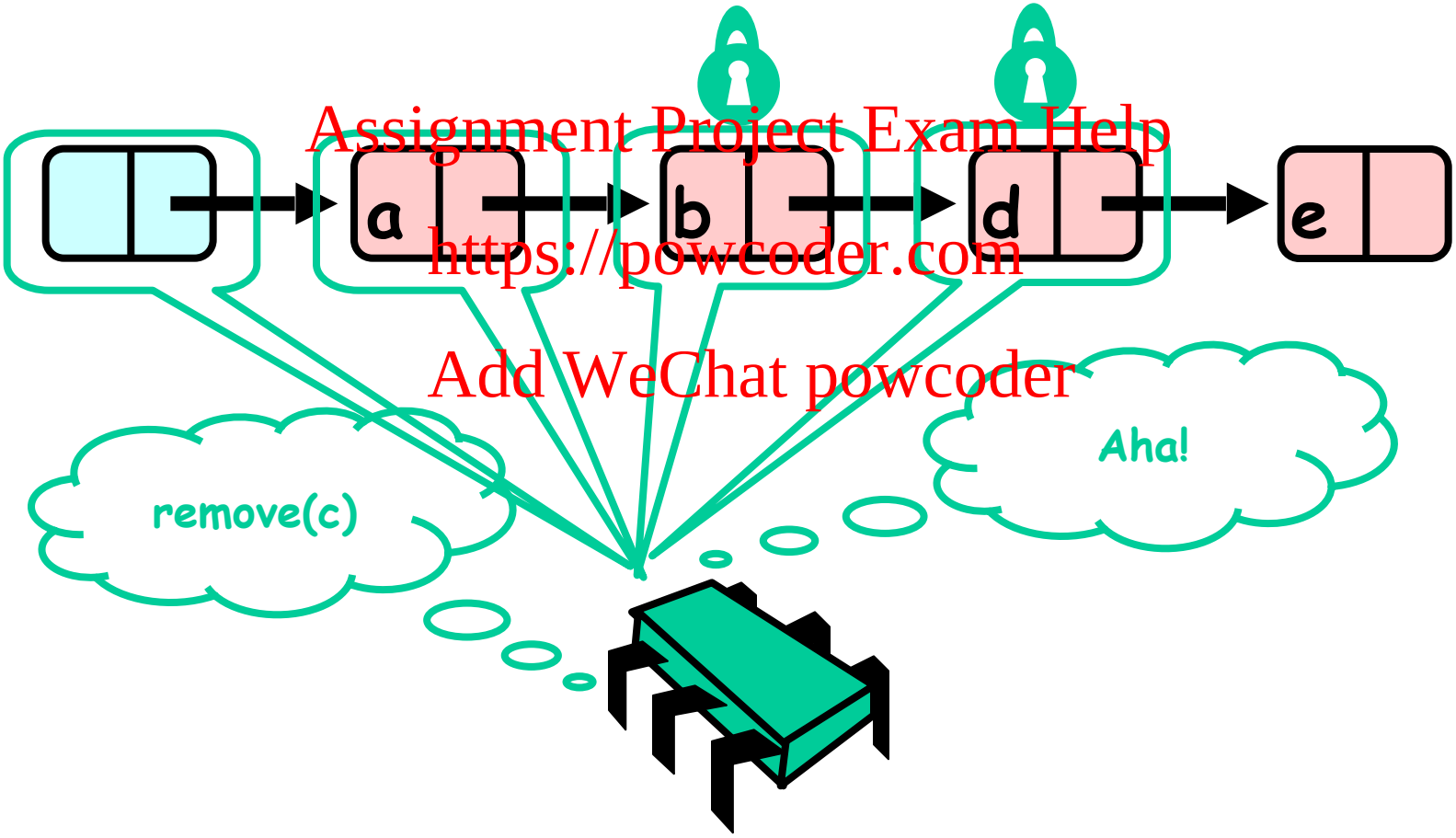
Optimistic: Linearization Point



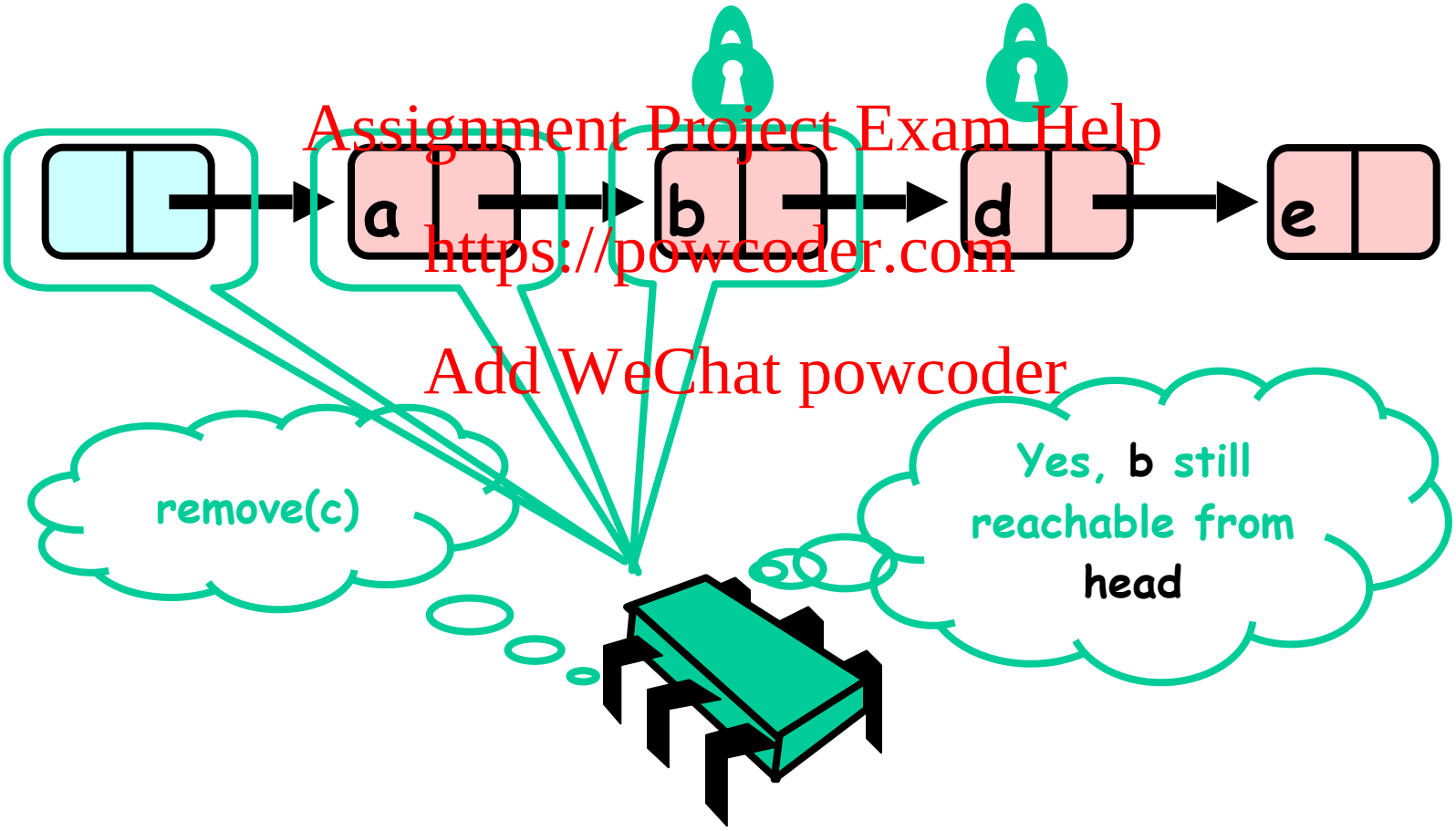
Correctness

- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

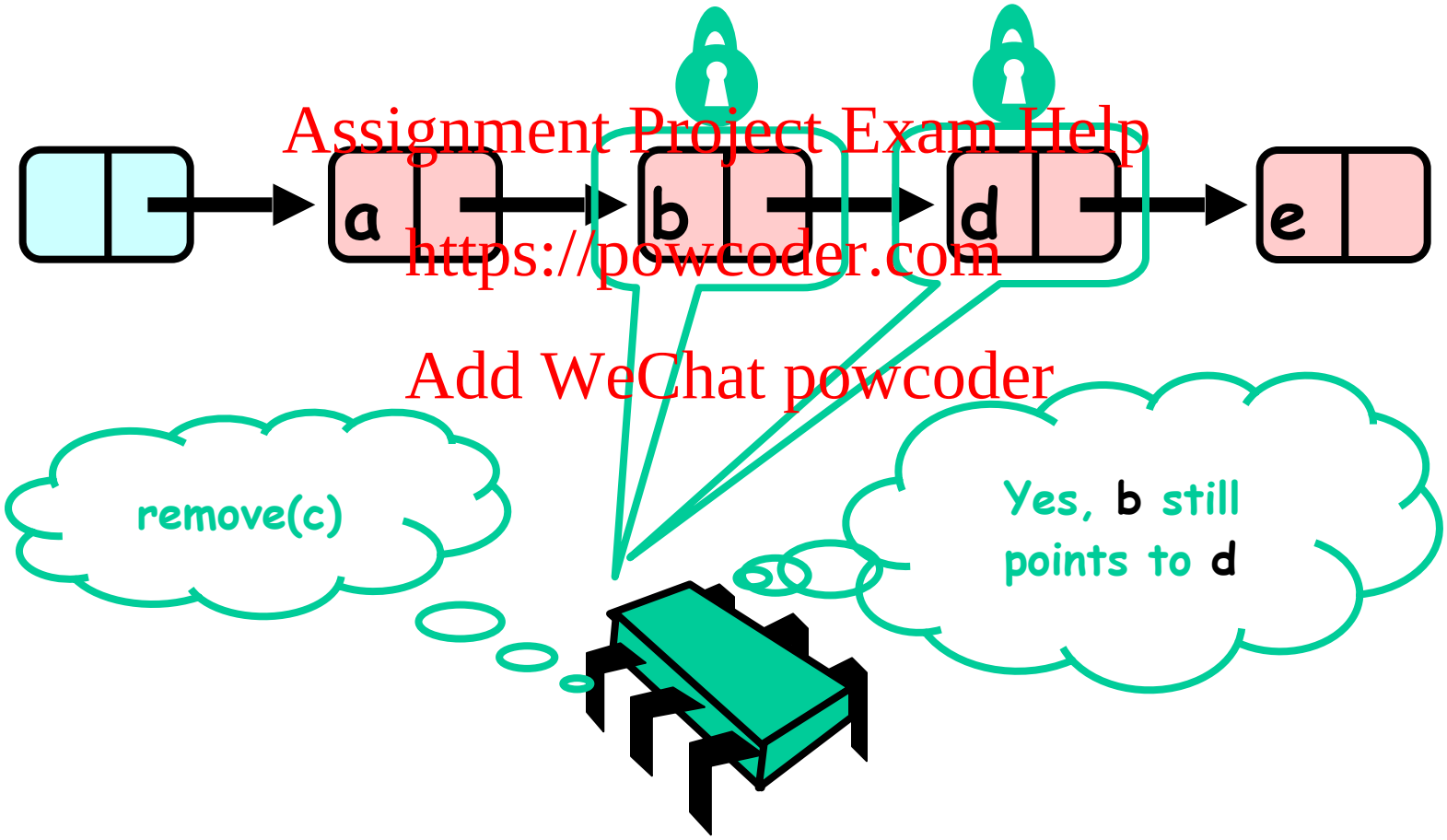
Unsuccessful Remove



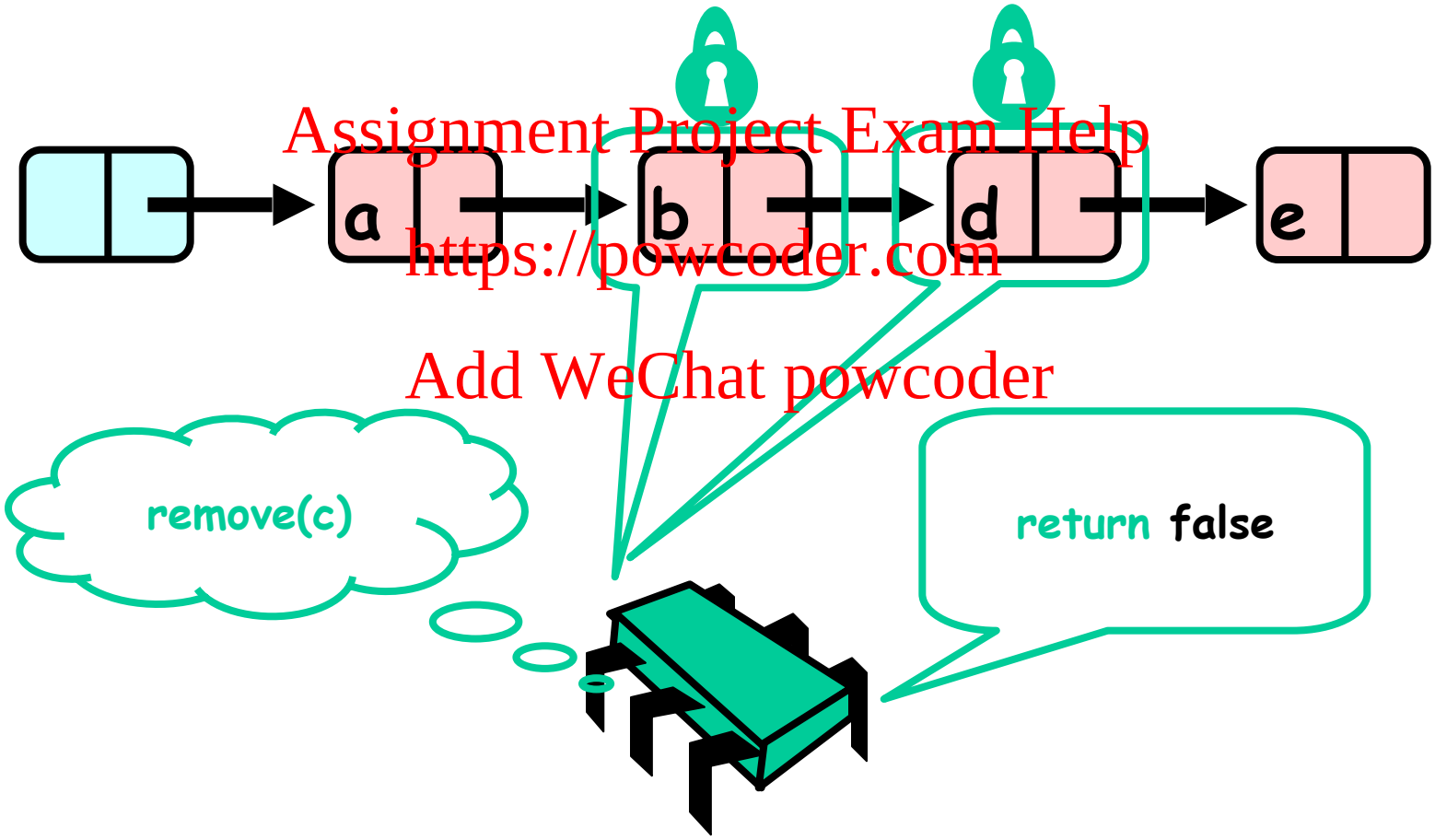
Validate (1)



Validate (2)



OK Computer



Correctness

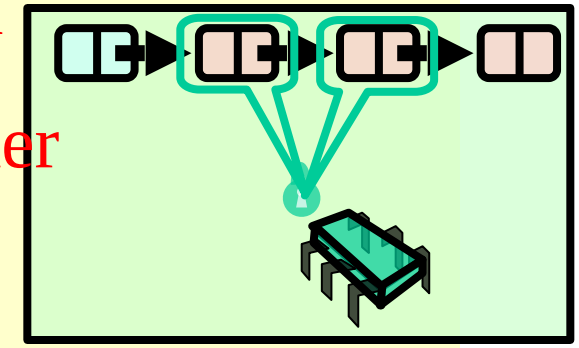
- If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false

Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

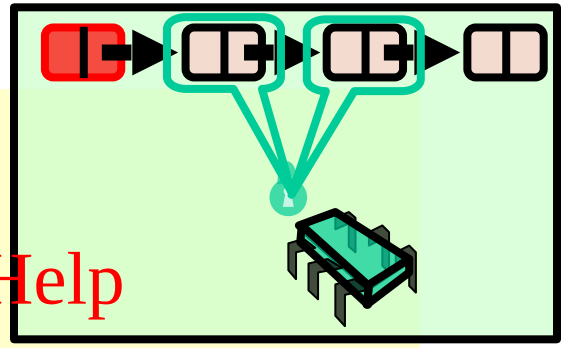
Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor & current nodes

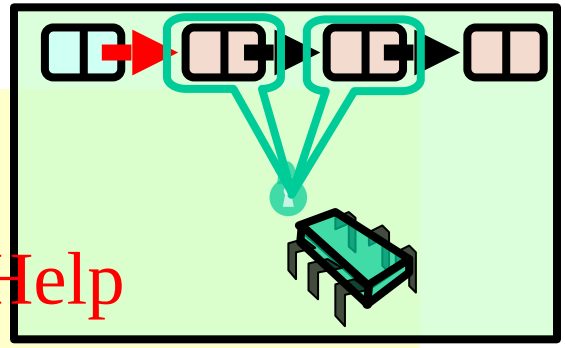
Validation



```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```


Begin at the beginning

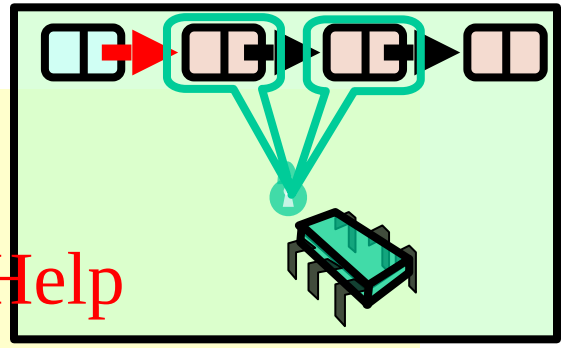
Validation



```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
    return pred.next == curr;
   node = node.next;
 }
 return false;
}
```


Search range of keys

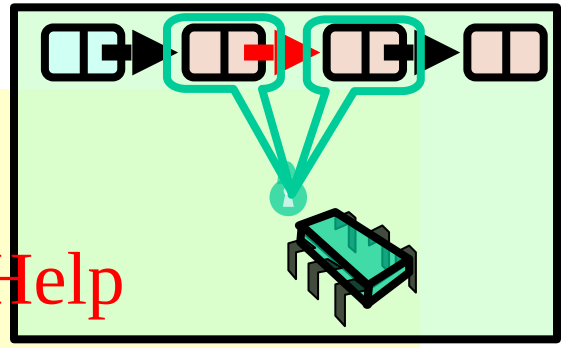
Validation



```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```


Predecessor reachable

Validation



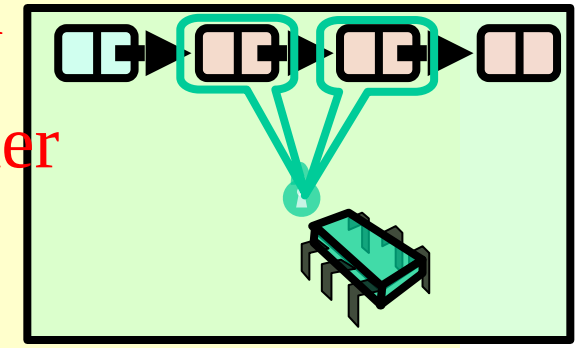
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
    return pred.next == curr;
  node = node.next;
 }
 return false;
}
```


Is current node next?

Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

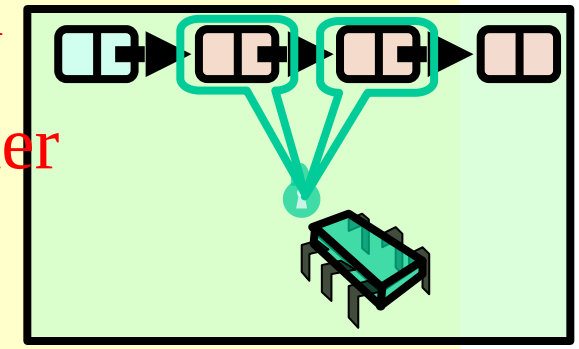
Otherwise move on



Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor not reachable



Optimistic Synchronization

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Entry(item);
          entry.next = curr;
          pred.next = node;
          return true;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

```
public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = pred.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock(); curr.lock();
     try {
       if (validate(pred, curr)) {
         if (curr.key == key) {
           pred.next = curr.next;
           return true;
         } else {
           return false;
         }
       }
     } finally {
       pred.unlock(); curr.unlock();
     }
   }
 }
```

```
public boolean contains(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = pred.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     try {
       pred.lock(); curr.lock();
       if (validate(pred, curr)) {
         return (curr.key == key);
       }
     } finally {
       pred.unlock(); curr.unlock();
     }
   }
 }
```

```
private boolean validate(Node pred, Node
  curr) {
  Node node = head;
  while (node.key <= pred.key) {
    if (node == pred)
      return pred.next == curr;
    Node = node.next;
  }
  return false;
}
```

Optimistic List

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …

So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() method acquires locks

Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks
- Drawback
  - contains() acquires locks
  - 90% of calls in many apps

Lazy List

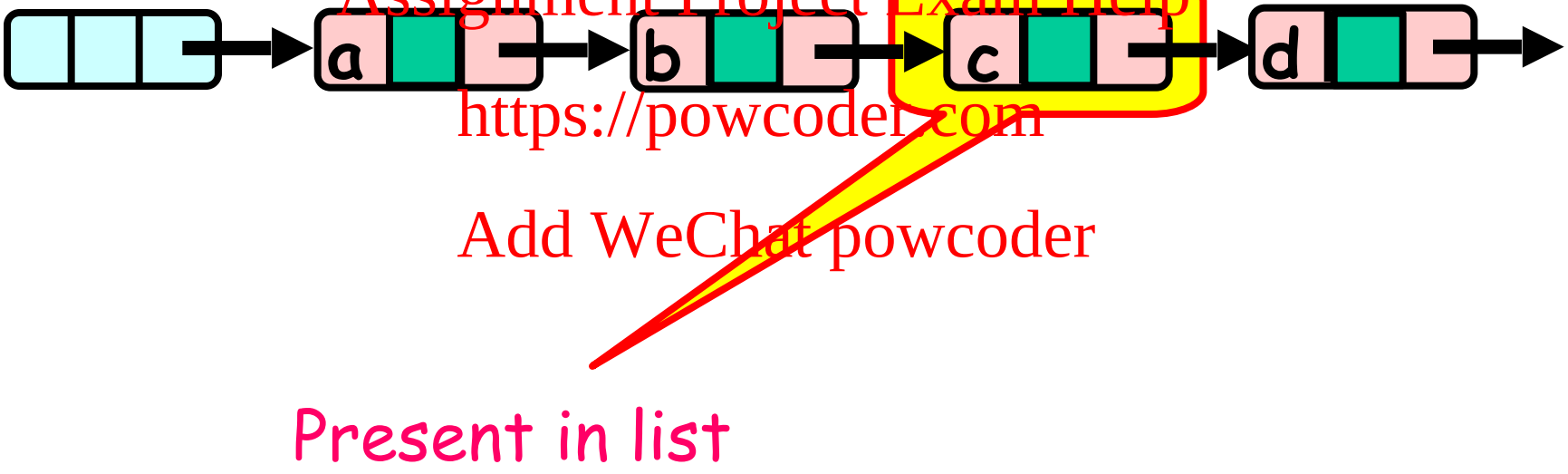
- Like optimistic, except
  - Scan once
  - contains(x) never locks …
- Key insight
  - Removing nodes causes trouble
  - Do it "lazily"

Lazy List

- remove()
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

Lazy Removal

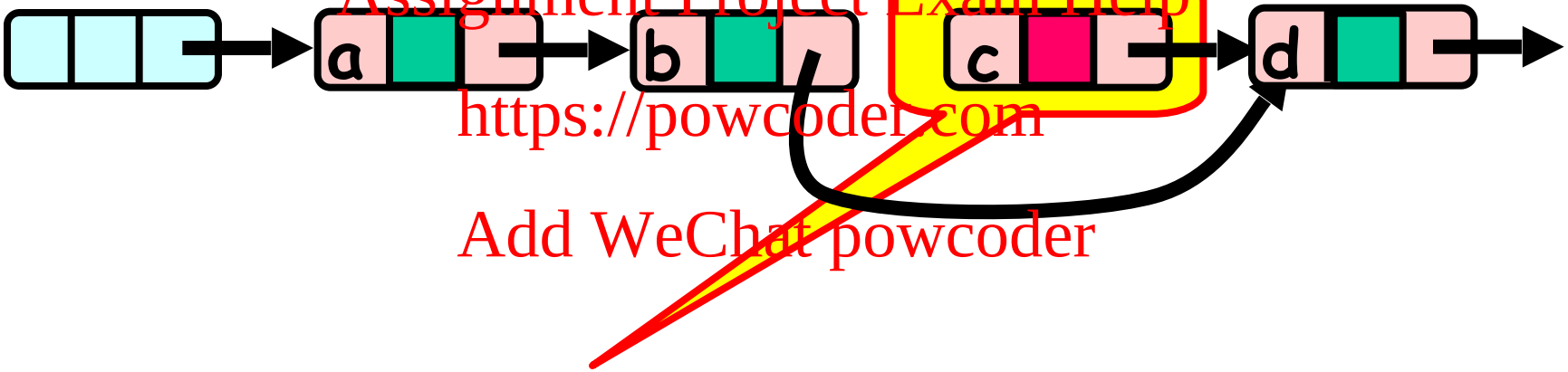
Lazy Removal



Lazy Removal

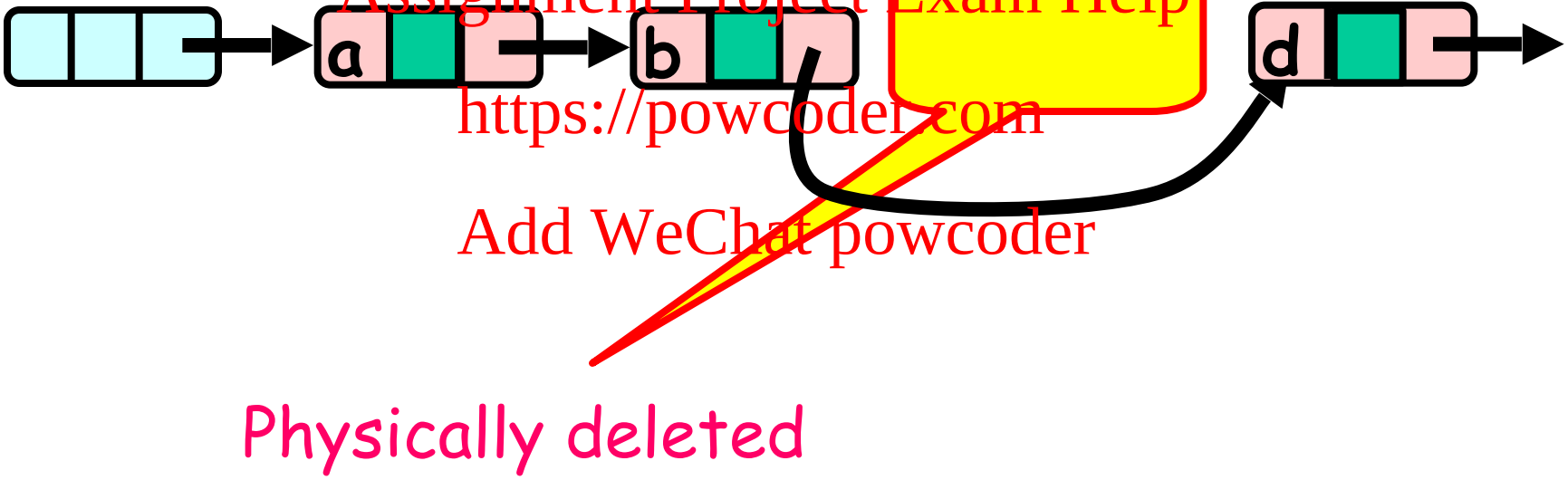
Logically deleted

Lazy Removal



Physically deleted

Lazy Removal



Physically deleted

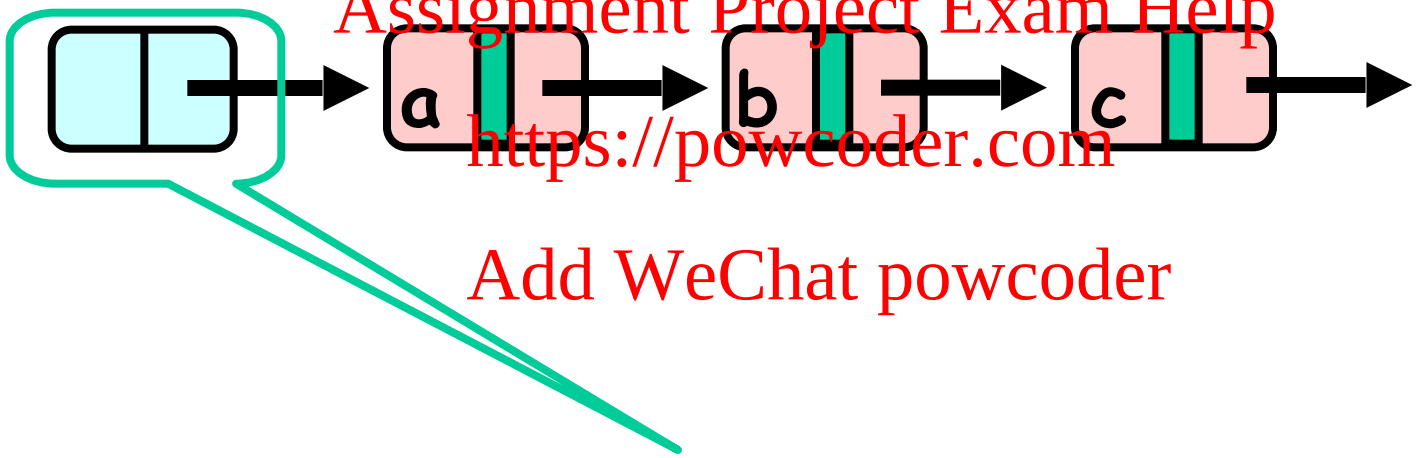
Lazy List

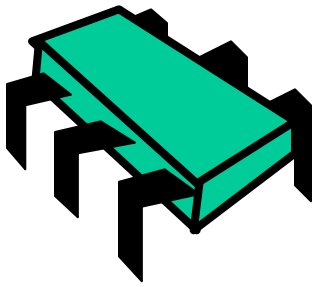
- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
- Must still lock pred and curr nodes.

Validation

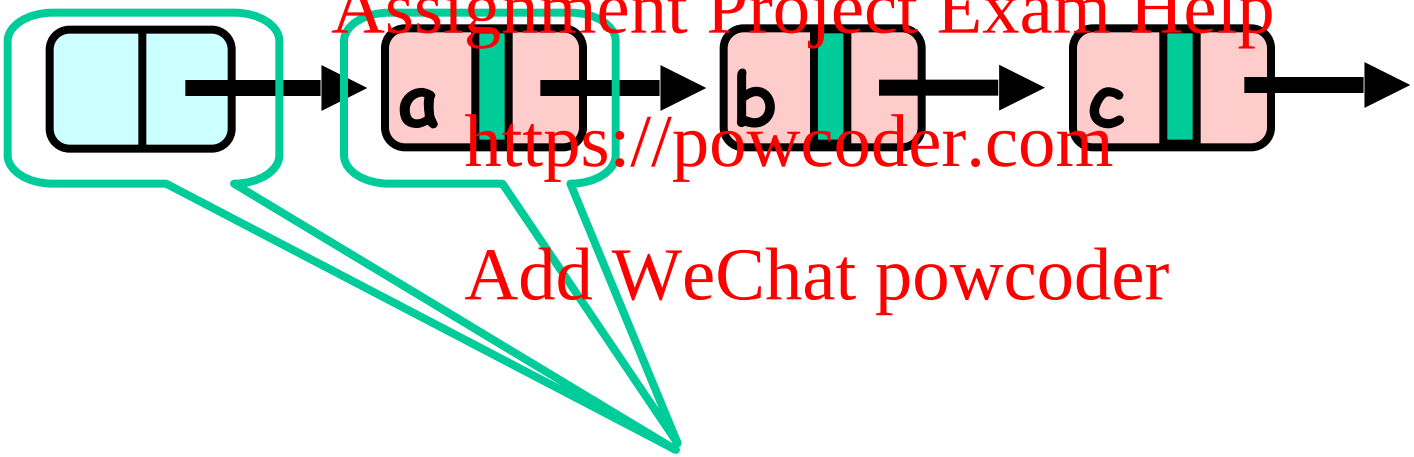
- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

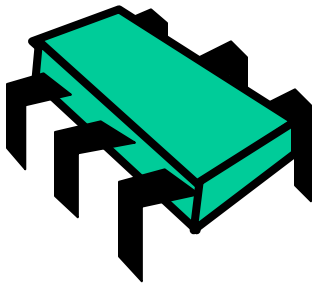
Business as Usual



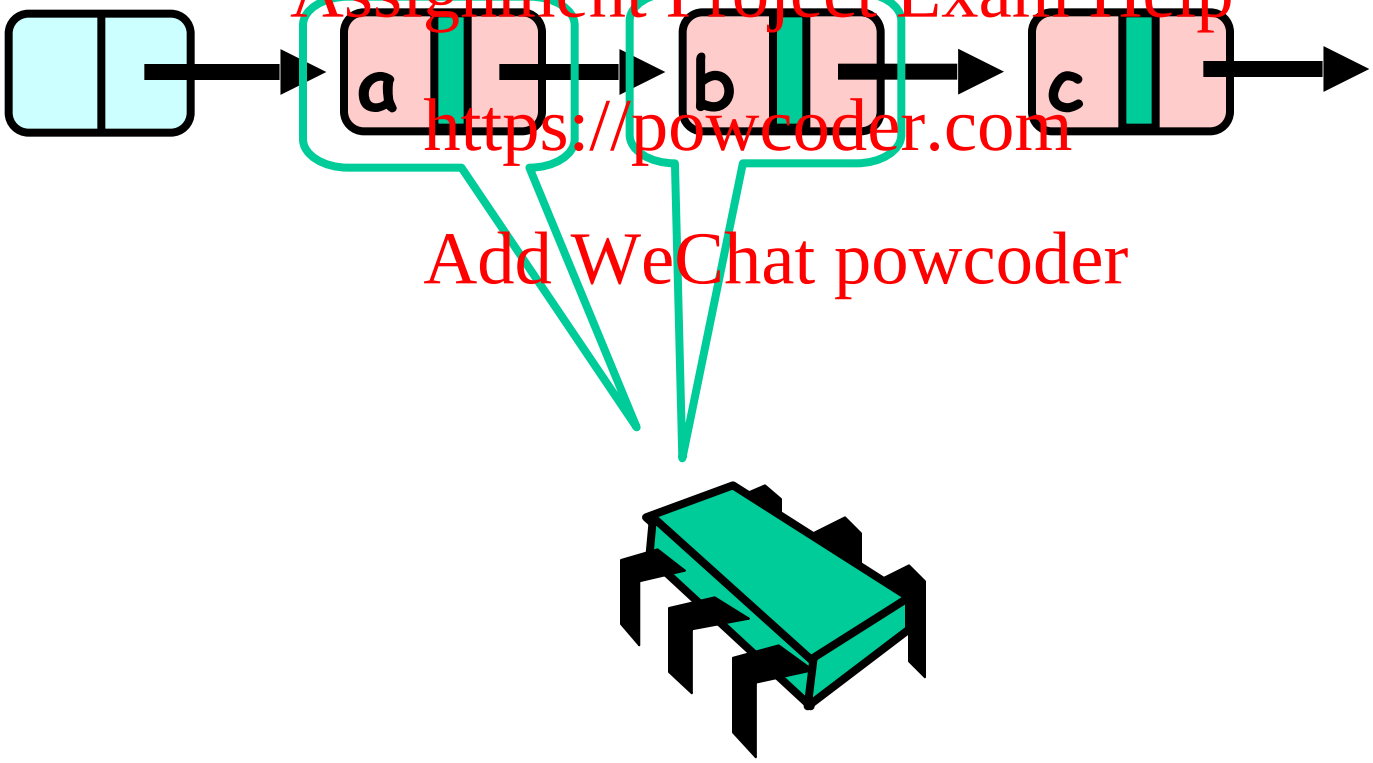


Business as Usual

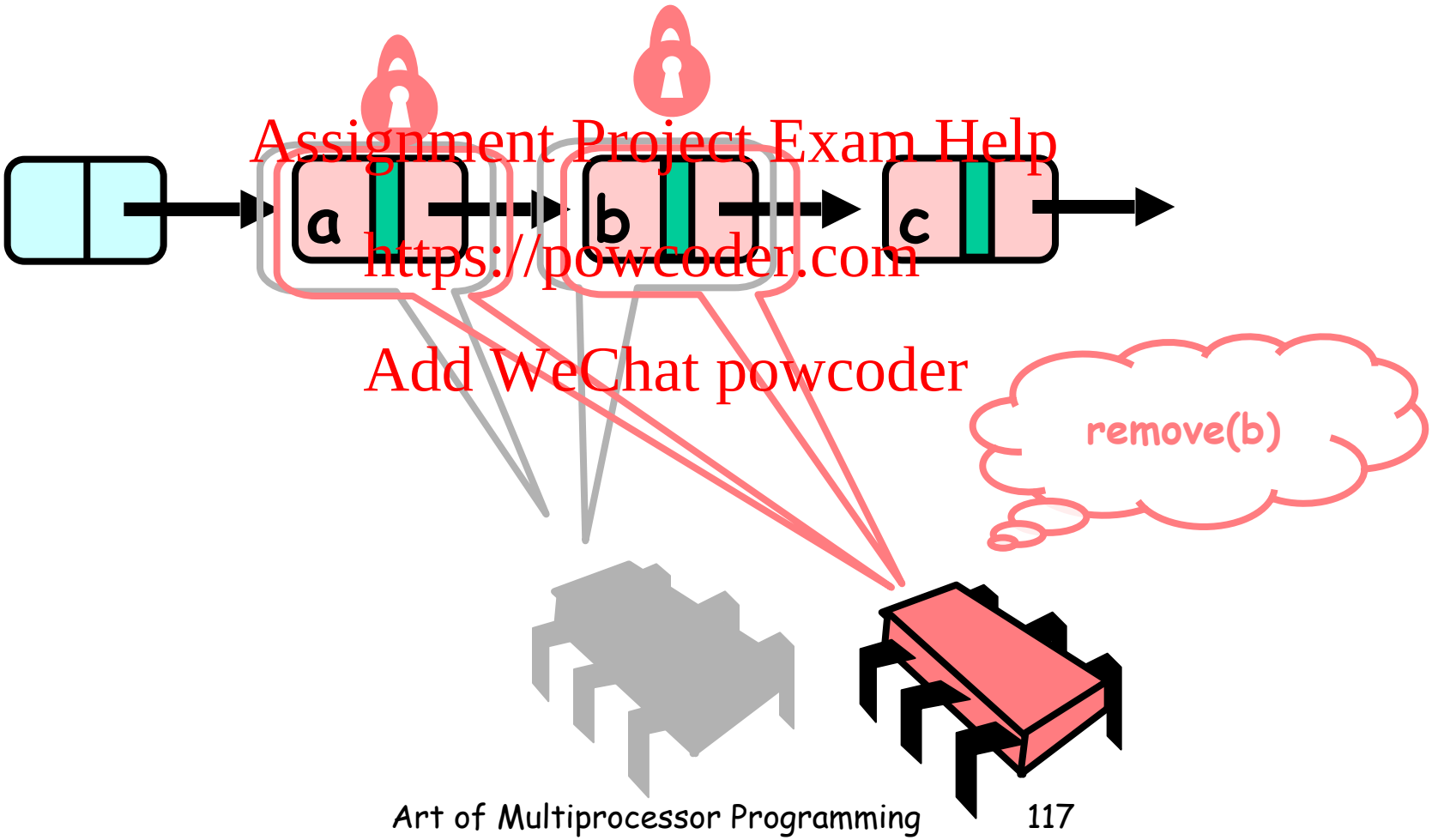




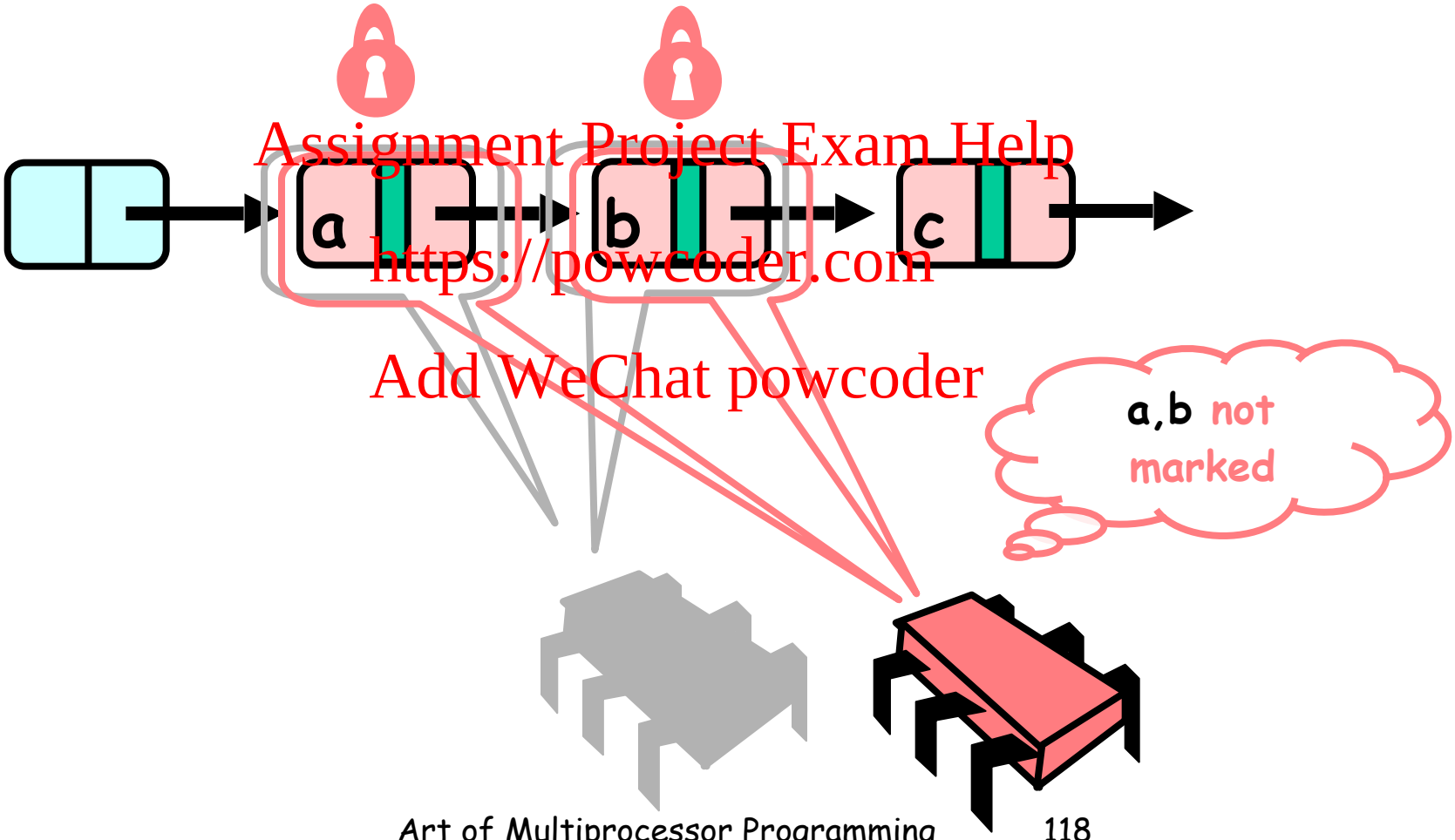
Business as Usual



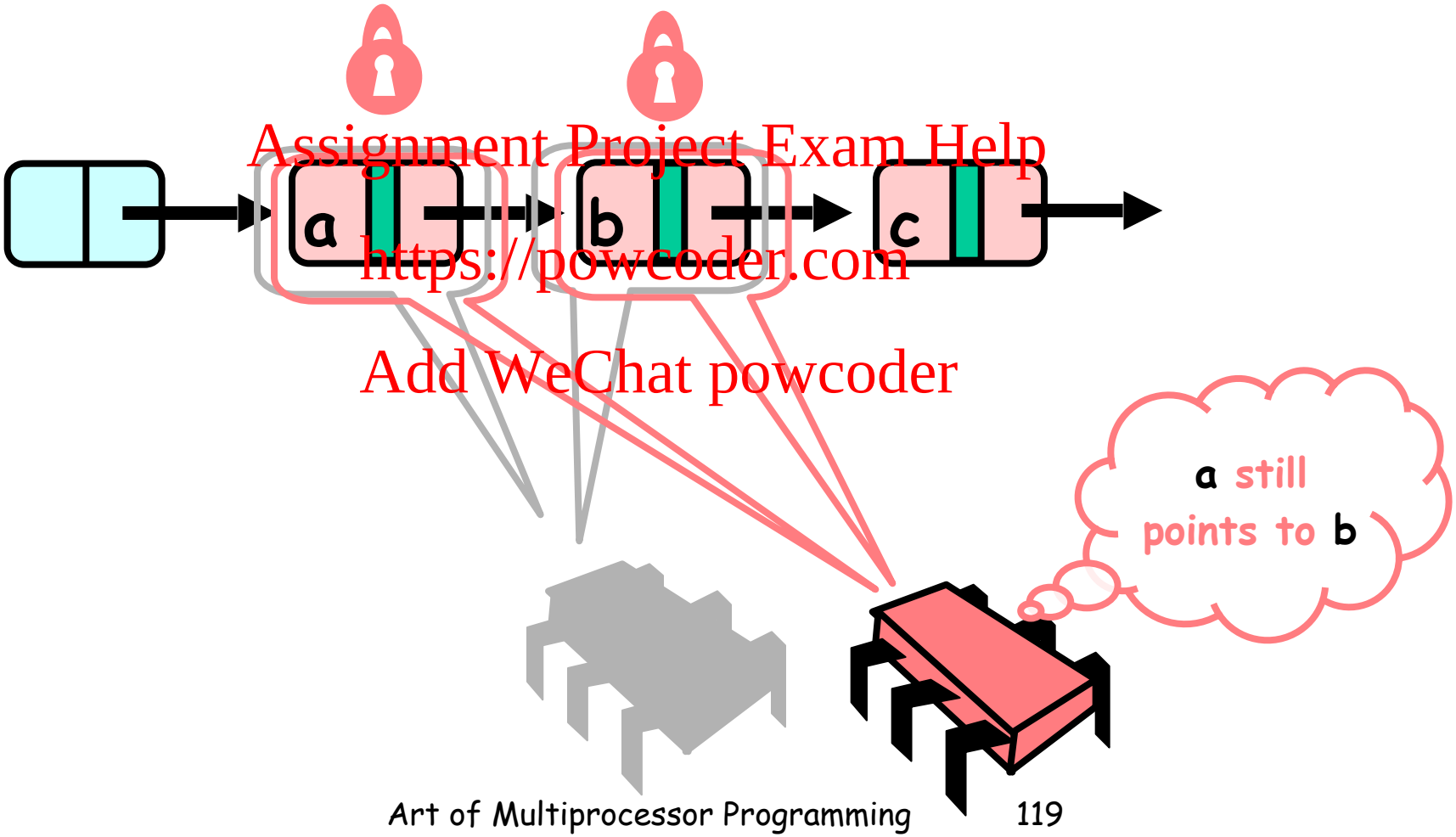
Business as Usual



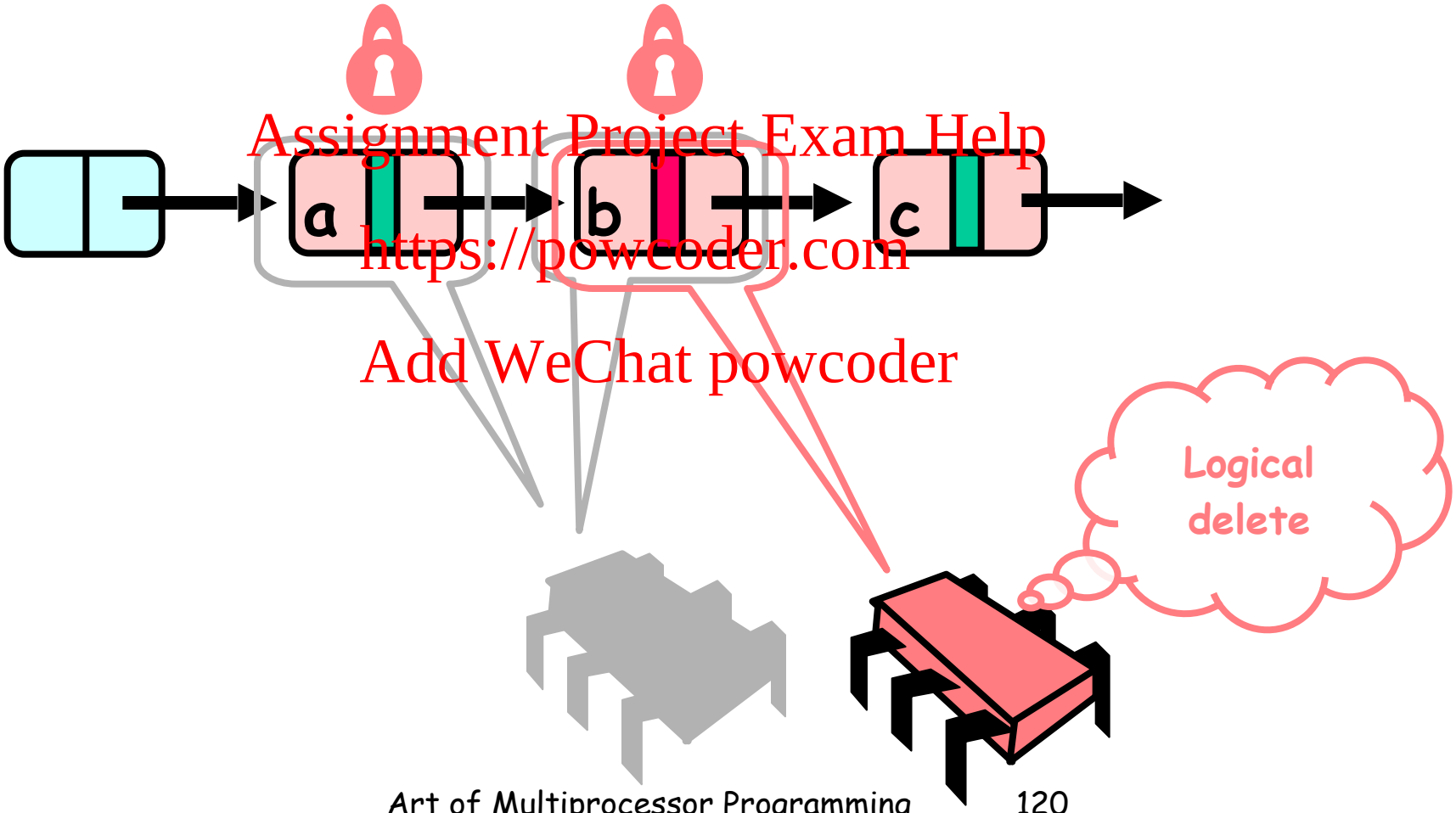
Business as Usual



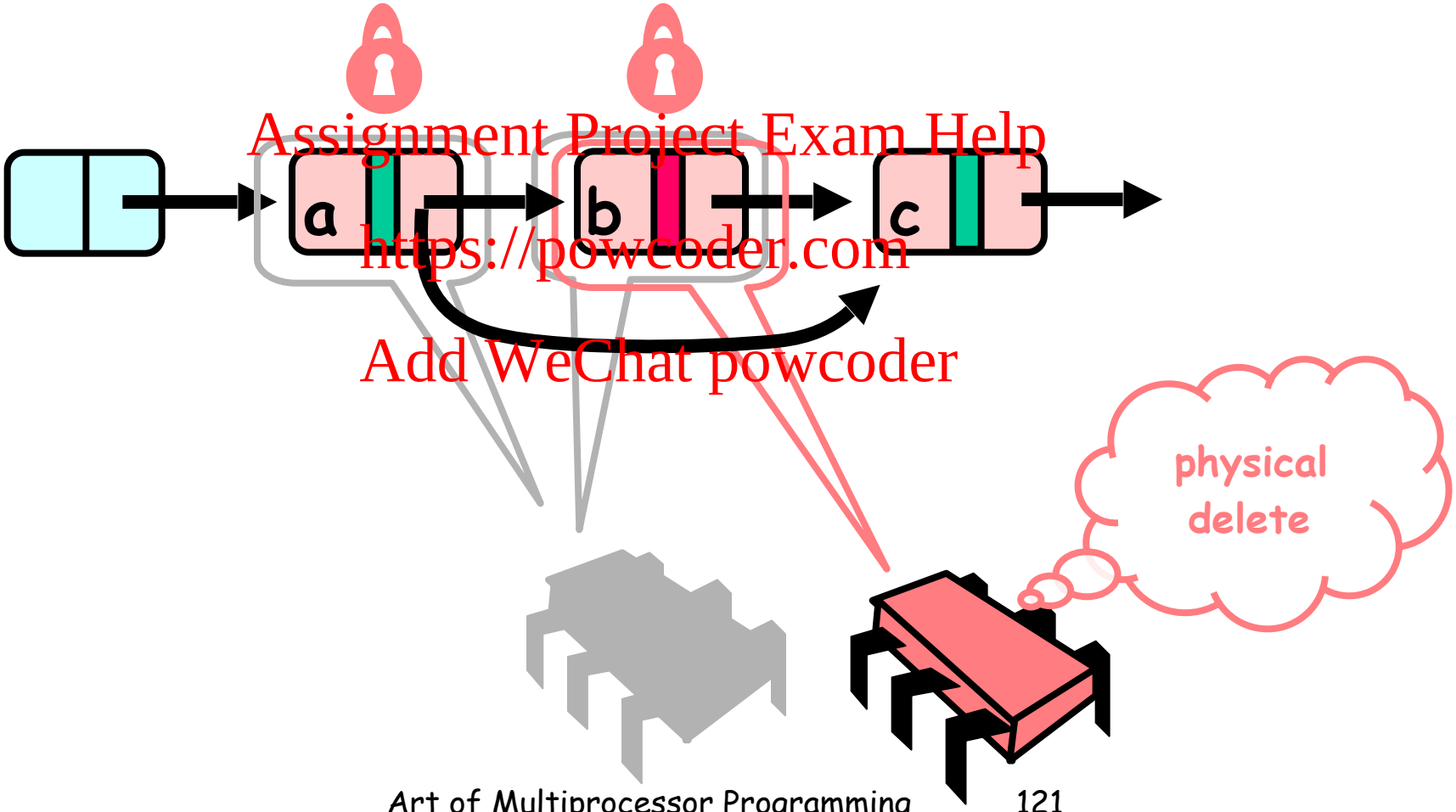
Business as Usual



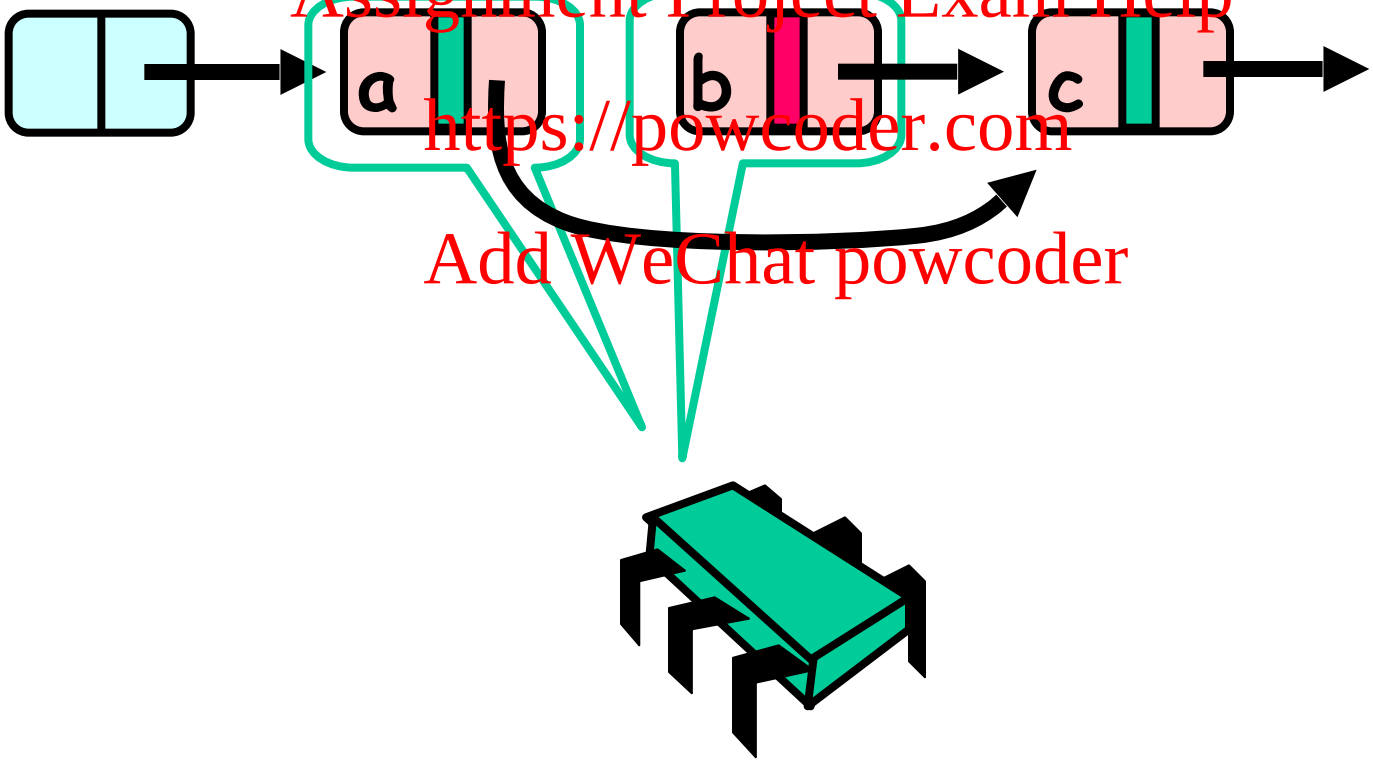
Business as Usual



Business as Usual



Business as Usual



Invariant

- If not marked then item in the set
- and reachable from head
- and if not yet traversed it is reachable from pred

Validation

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```


Predecessor not Logically removed

List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```


Current not Logically removed

List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
}
```


Predecessor still Points to current

Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```


Start at the head

Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```


Search key range

Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```


Traverse without locking

(nodes may have been removed)

Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```


Present and undeleted?

Lazy Synchronization

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = head.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock();
     try {
       curr.lock();
       try {
         if (validate(pred, curr)) {
           if (curr.key == key) {
             return false;
           } else {
             Node Node = new Node(item);
             Node.next = curr;
             pred.next = Node;
             return true;
           }
         }
       } finally { // always unlock
         curr.unlock();
       }
     } finally { // always unlock
       pred.unlock();
     }
   }
 }
```

```
public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = head.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock();
     try {
       curr.lock();
       try {
         if (validate(pred, curr)) {
           if (curr.key != key) {
             return false;
           } else {
             curr.marked = true;
             pred.next = curr.next;
             return true;
           }
         }
       } finally {
         curr.unlock();
       }
     } finally {
       pred.unlock();
     }
   }
 }
```

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

```
 private boolean validate(Node pred, Node
   curr) {
   return !pred.marked && !curr.marked &&
   pred.next == curr;
 }
```

Evaluation

- Good:
  - contains() doesn't lock
  - 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