Linked List

Set Data Structure

- Operations:
    - Add an element
    - Remove an element
    - Answer question about containment
- Implemented as a singly linked list

Linked List

- Adding threads should not <u>lower</u> throughput
  - Contention effects
  - Fixed by queue locks
- Should increase throughput
  - Not possible if inherently sequential
  - But surprising things are parallelizable

Linked List

- Coarse-Grained synchronization
  - Each method locks the object
    - Avoid contention using queue locks
    - Easy to reason about
    - But "Sequential Bottleneck"
      - Threads stand in line
      - So adding more threads does not improve throughput
      - In fact, could make things worse

Linked List

- Instead of using a single lock:
    - Use fine-grained synchronization
        - Split object into
            - independently synchronized components
        - Methods conflict only:
            - When they access the same component at the same time

Linked List

- Use optimistic synchronization
  - Search without locking
  - If you find it, lock, and check that it did not change
- In general, optimistic synchronization
  - Is good when it works
  - But mistakes are expensive

Linked List

- Lazy synchronization
  - Postpone hard work
  - Removing components is tricky
    - So use *logical* removal:
      - Mark the component as deleted instead of deleting it
    - Followed by *physical* removal:
      - Delete the component

Linked List

- Lock-free Synchronization
  - Don't use locks at all
    - Use Compare-And-Set and relatives
    - Needs no scheduler assumptions or support
    - But is complex and can have high overhead

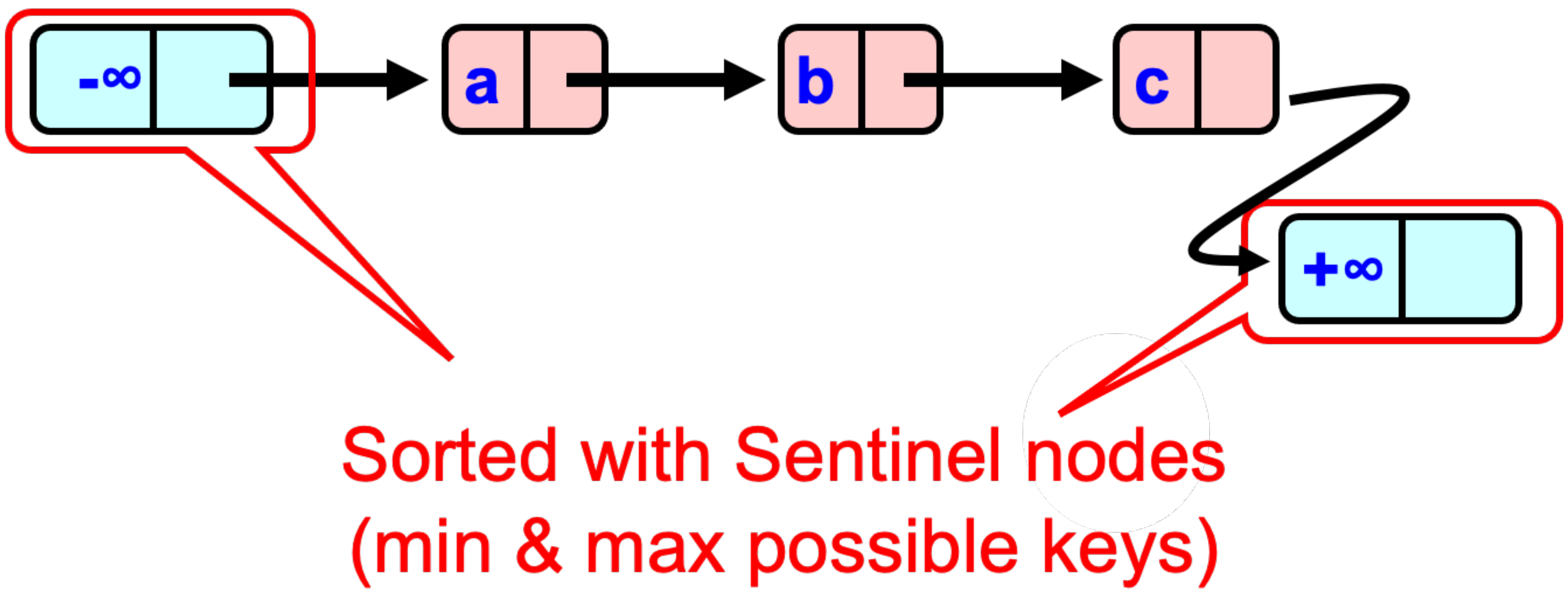
Linked List

- Singly linked list:
  - Use a List Node class

```
public class Node {
   public T item;
   public int key;
   public volatile Node next;
```

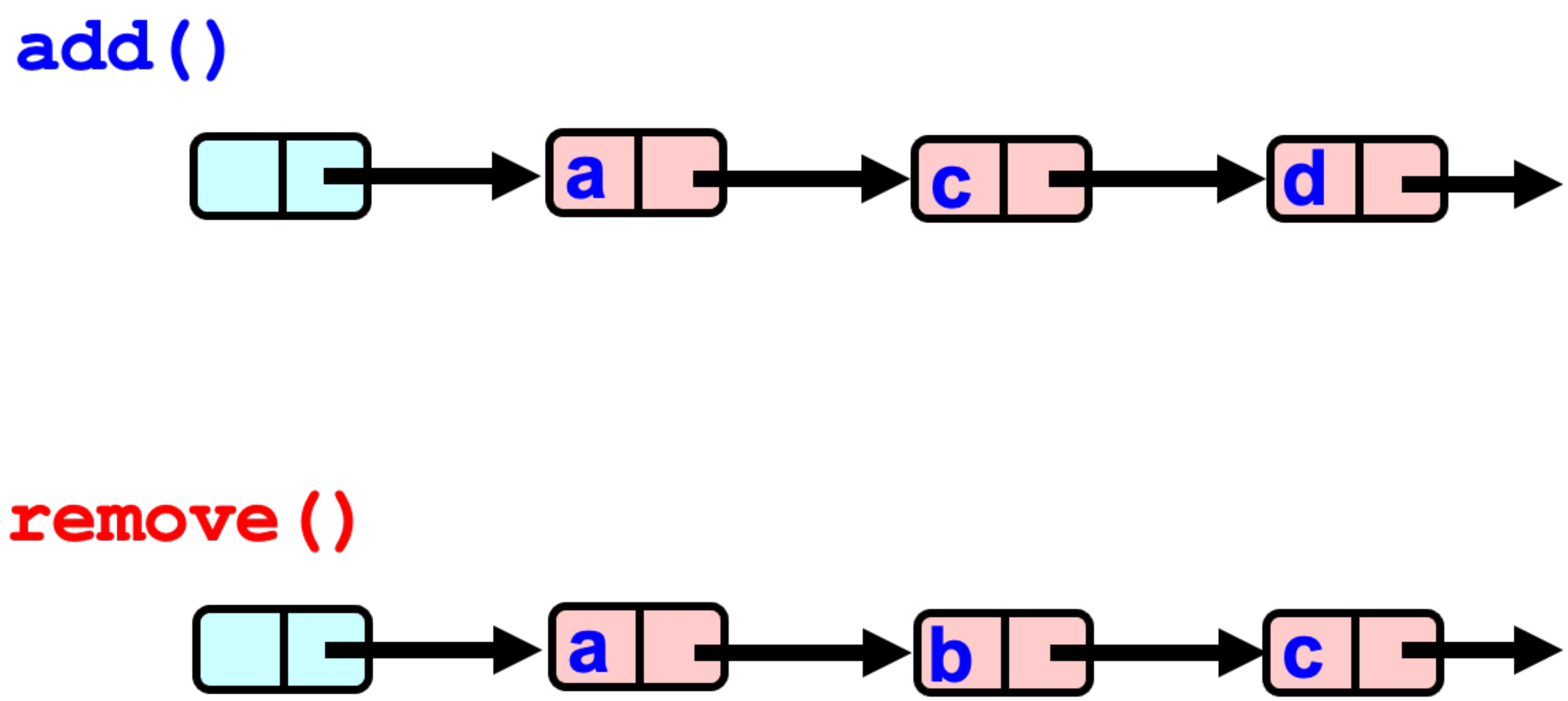
Linked List

- Use *Sentinel Nodes*



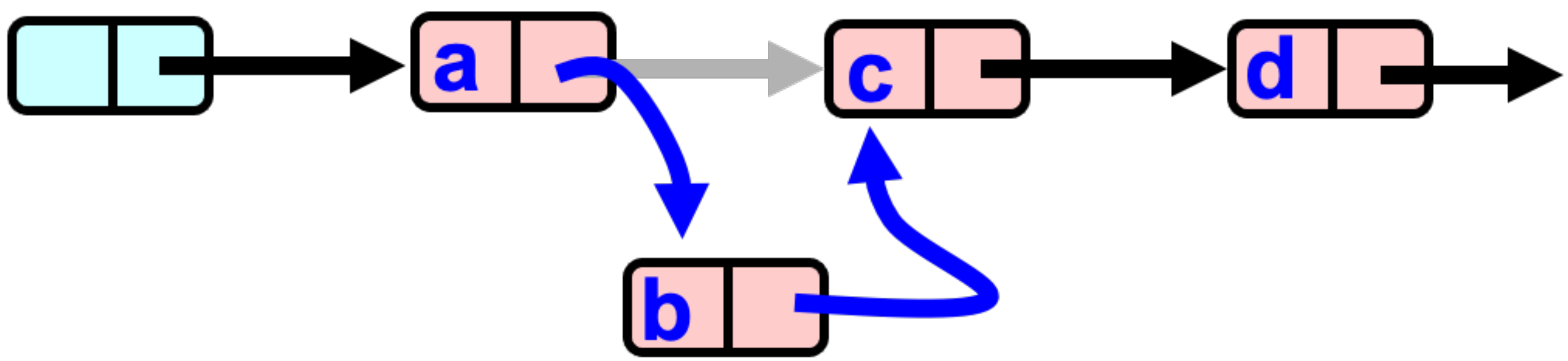
Linked List

- Operations involve pointer chasing

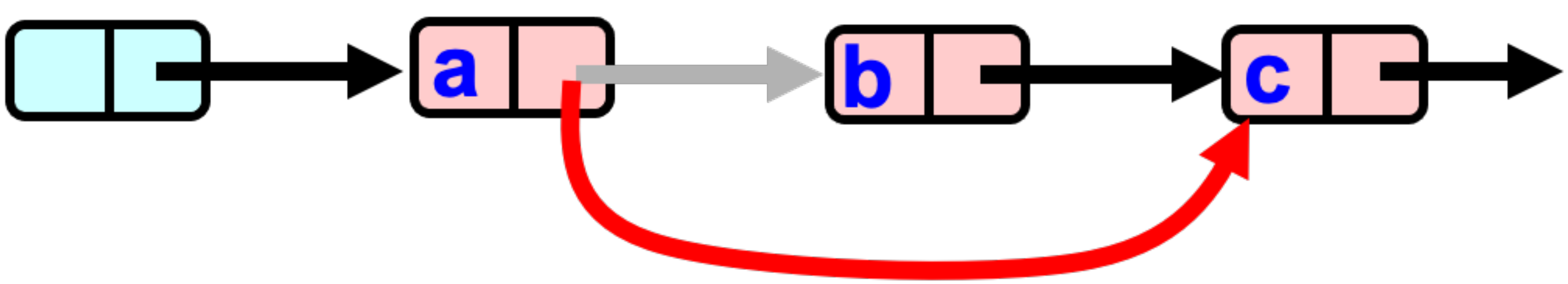


Linked List

add()

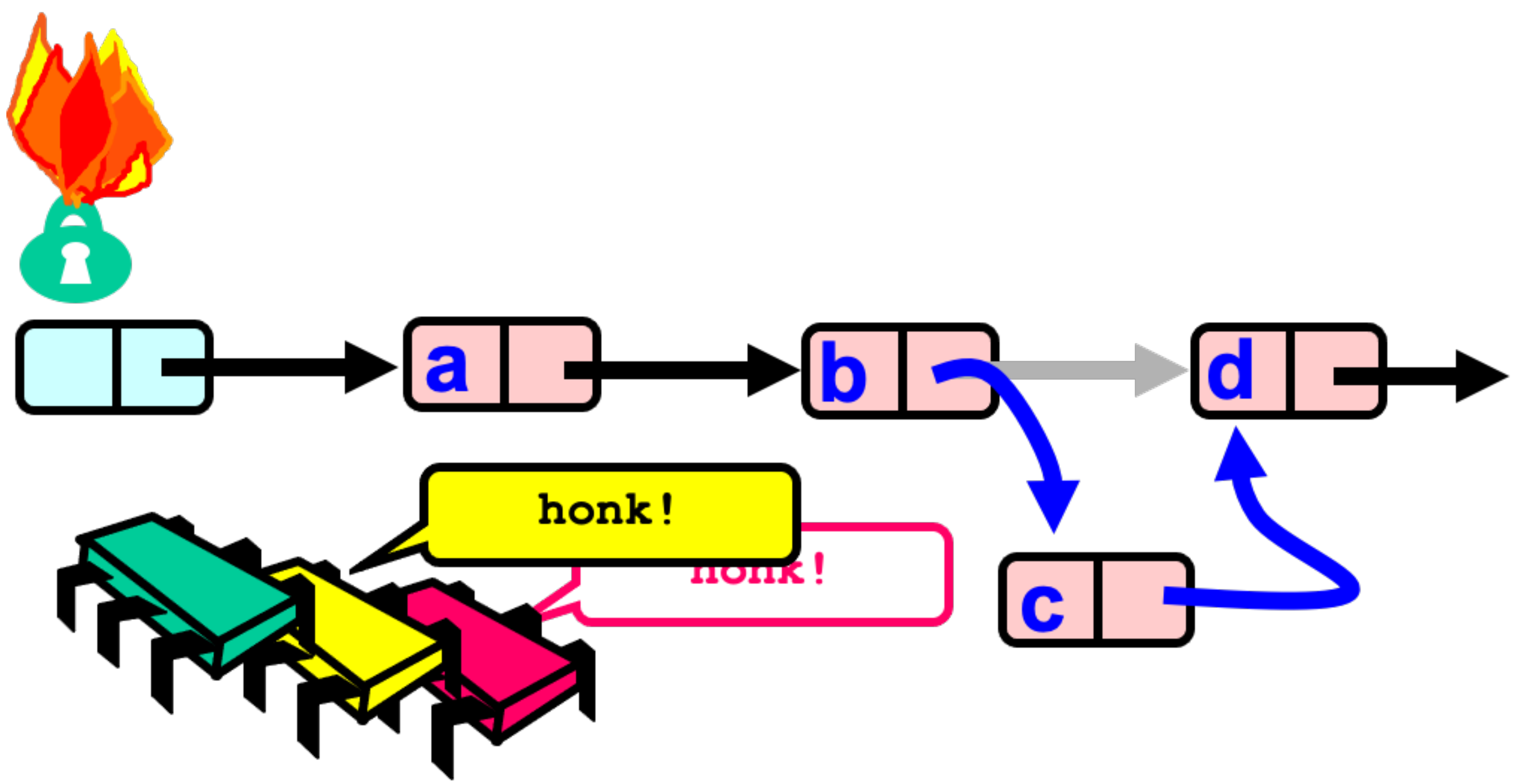


remove()



Coarse Grained Locking

- Coarse Grained Locking
  - Single hotspot + bottleneck leads to convoys

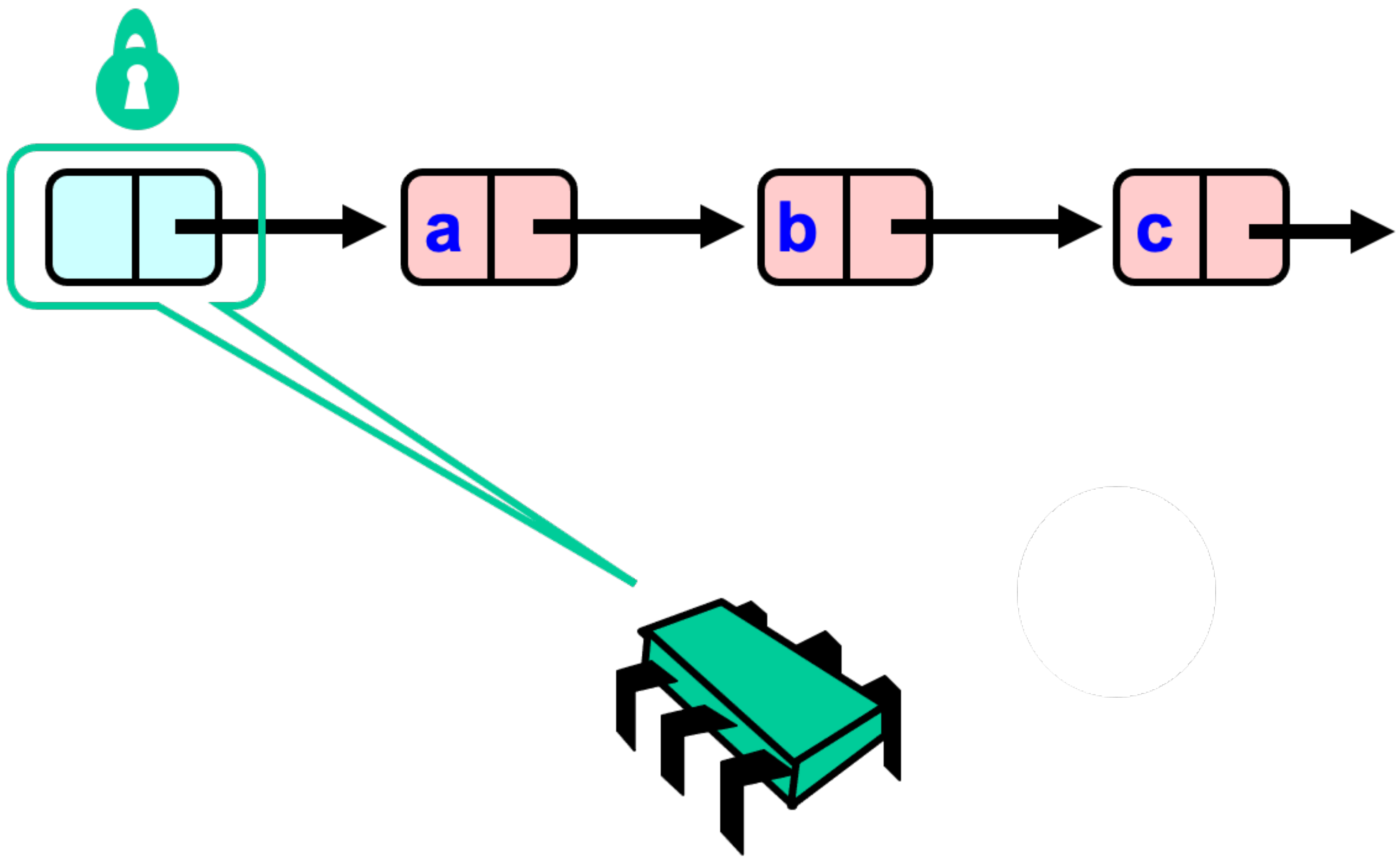


Fine Grained Locking

- Fine-grained locking
  - Requires care
  - Split object into pieces
    - Each piece has its own lock
    - Methods that work on disjoint set of pieces do not exclude each other

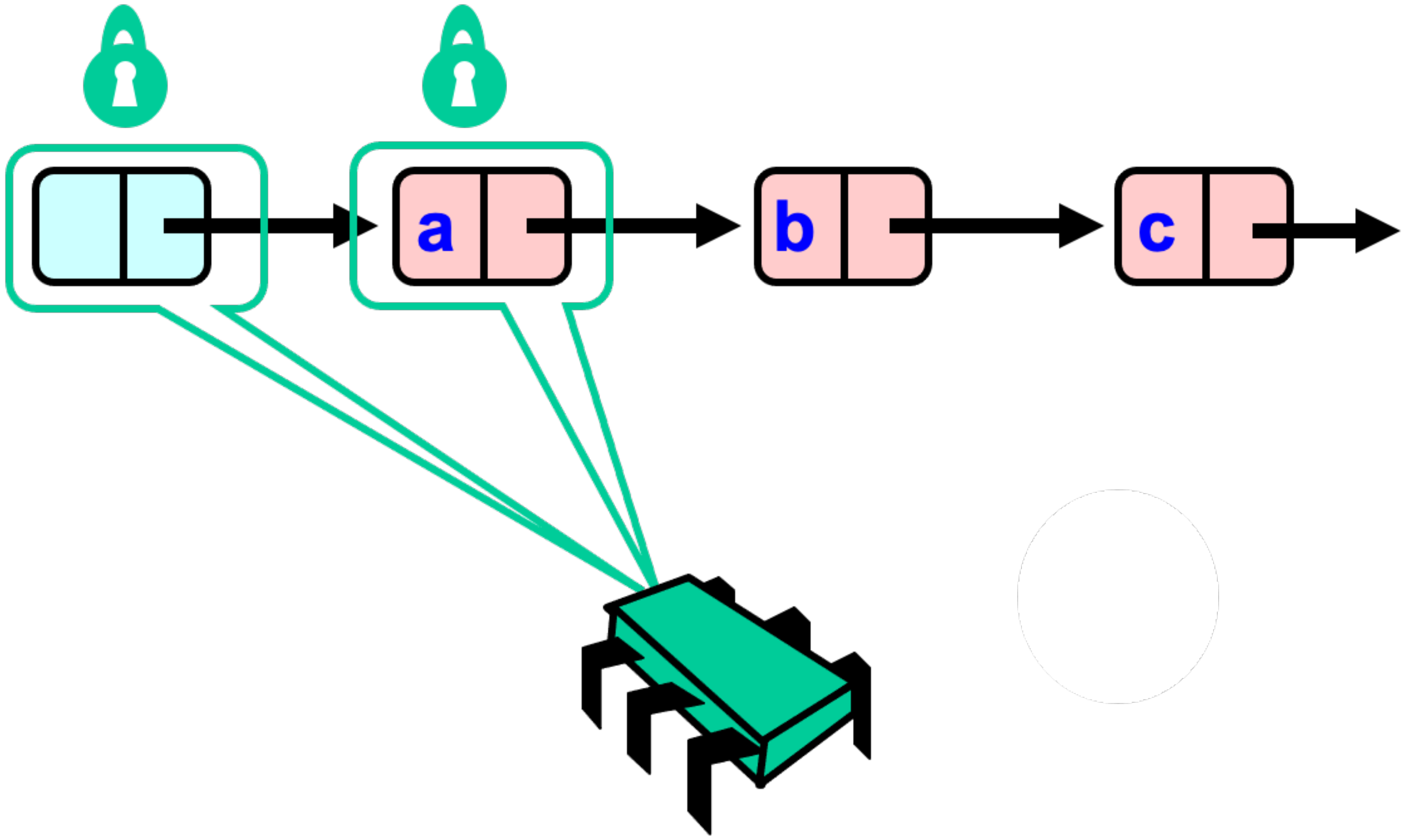
Fine Grained Locking

- Hand-over-Hand locking



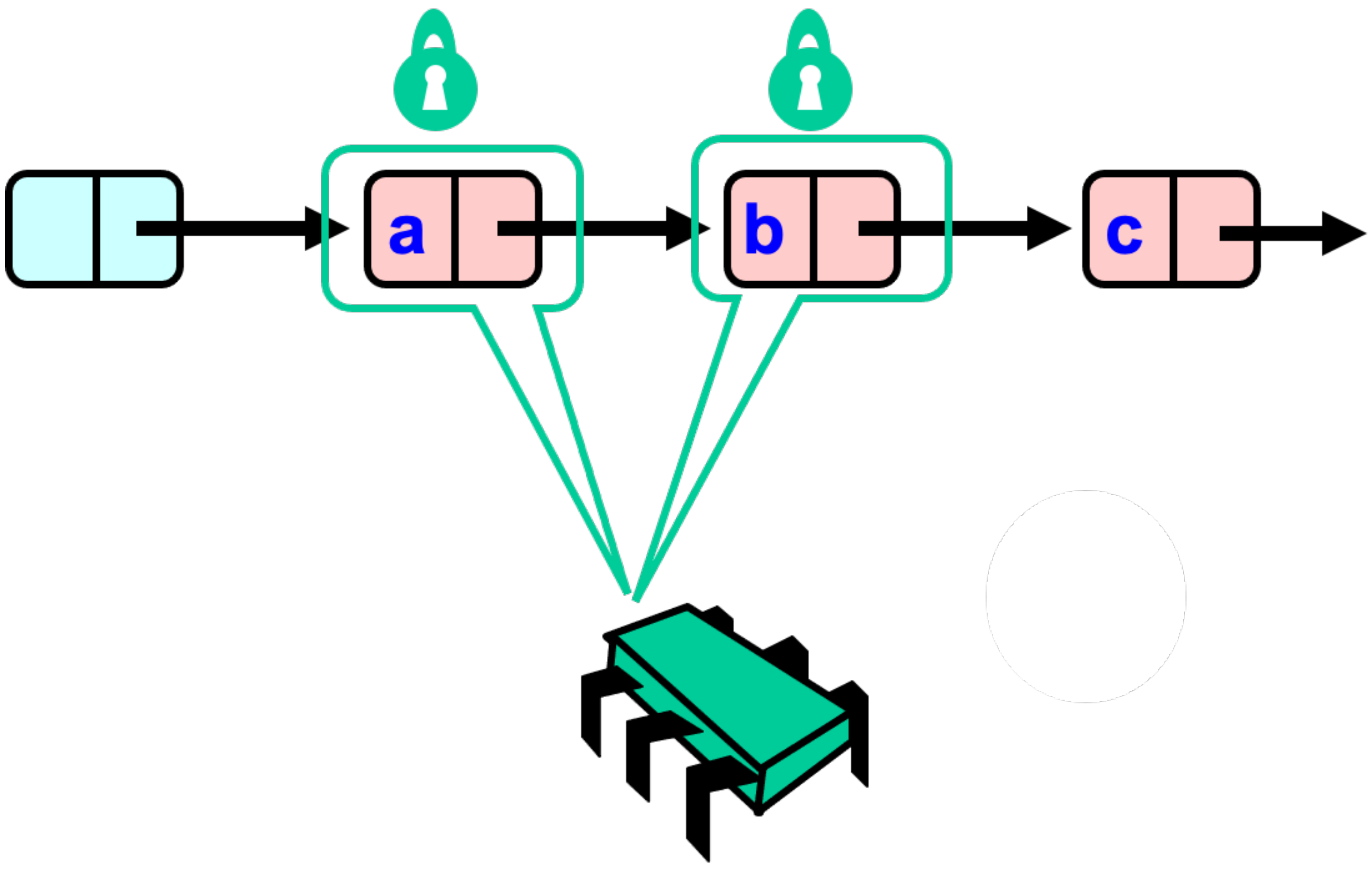
Fine Grained Locking

- Hand-over-Hand locking



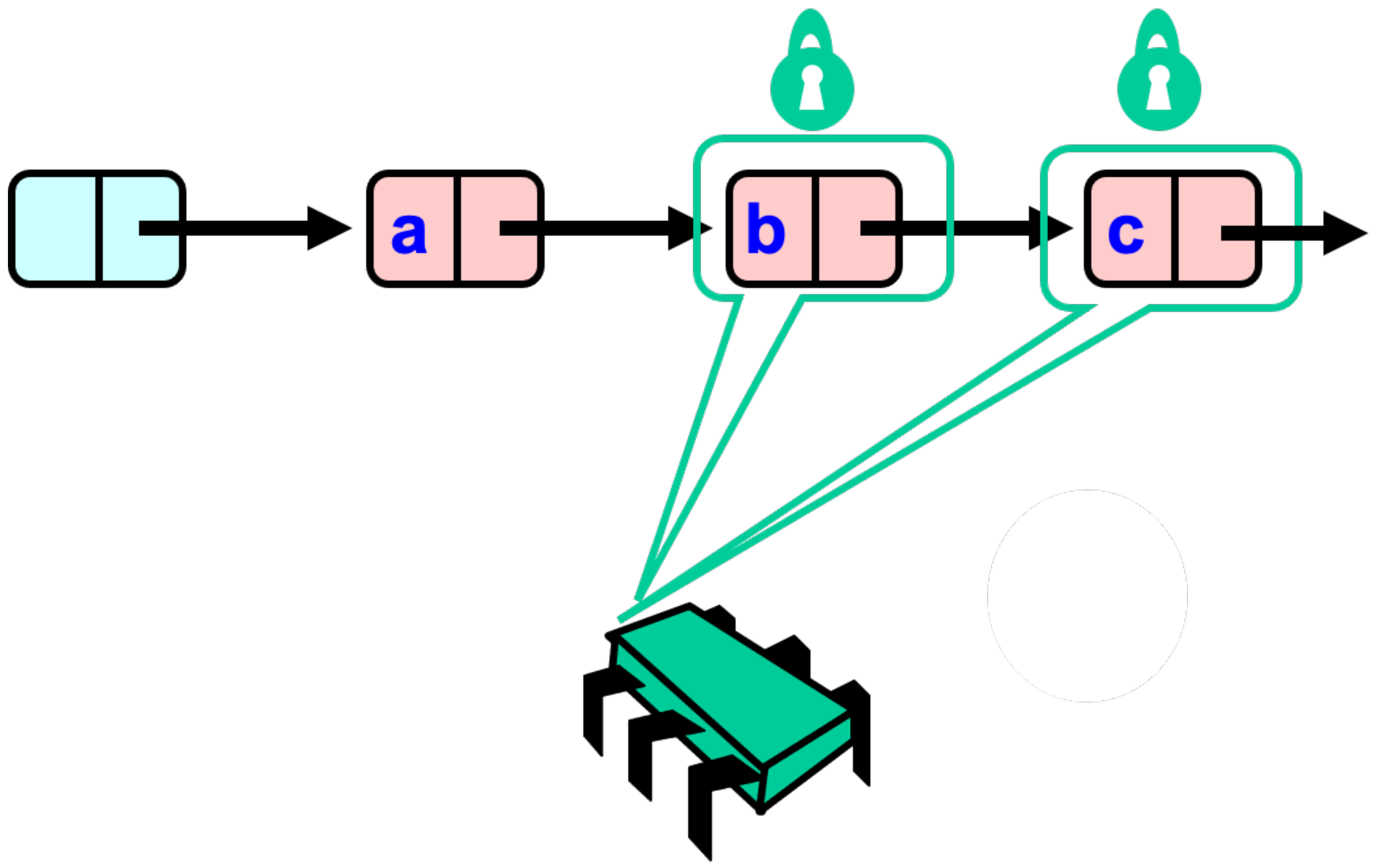
Fine Grained Locking

- Hand-over-Hand locking



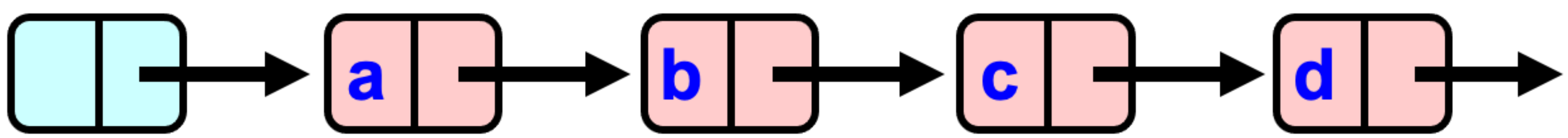
Fine Grained Locking

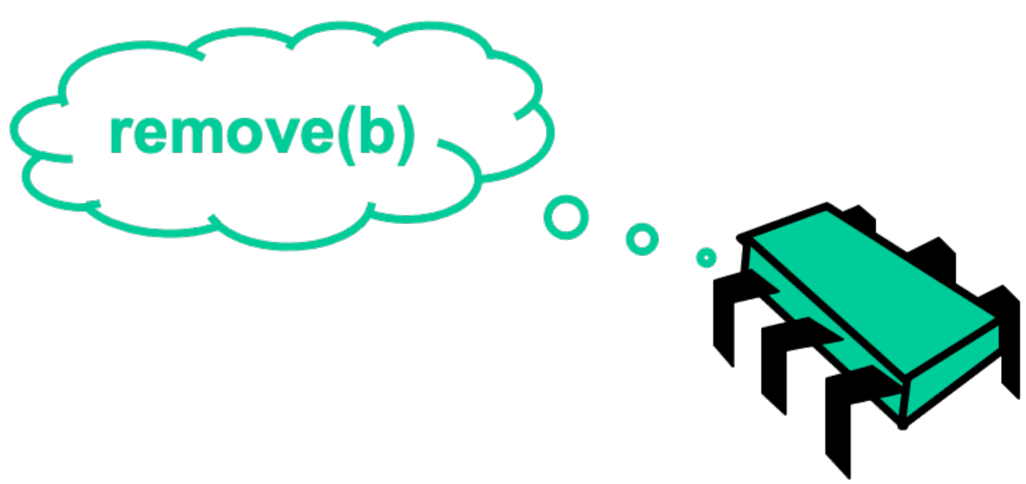
- Hand-over-Hand locking



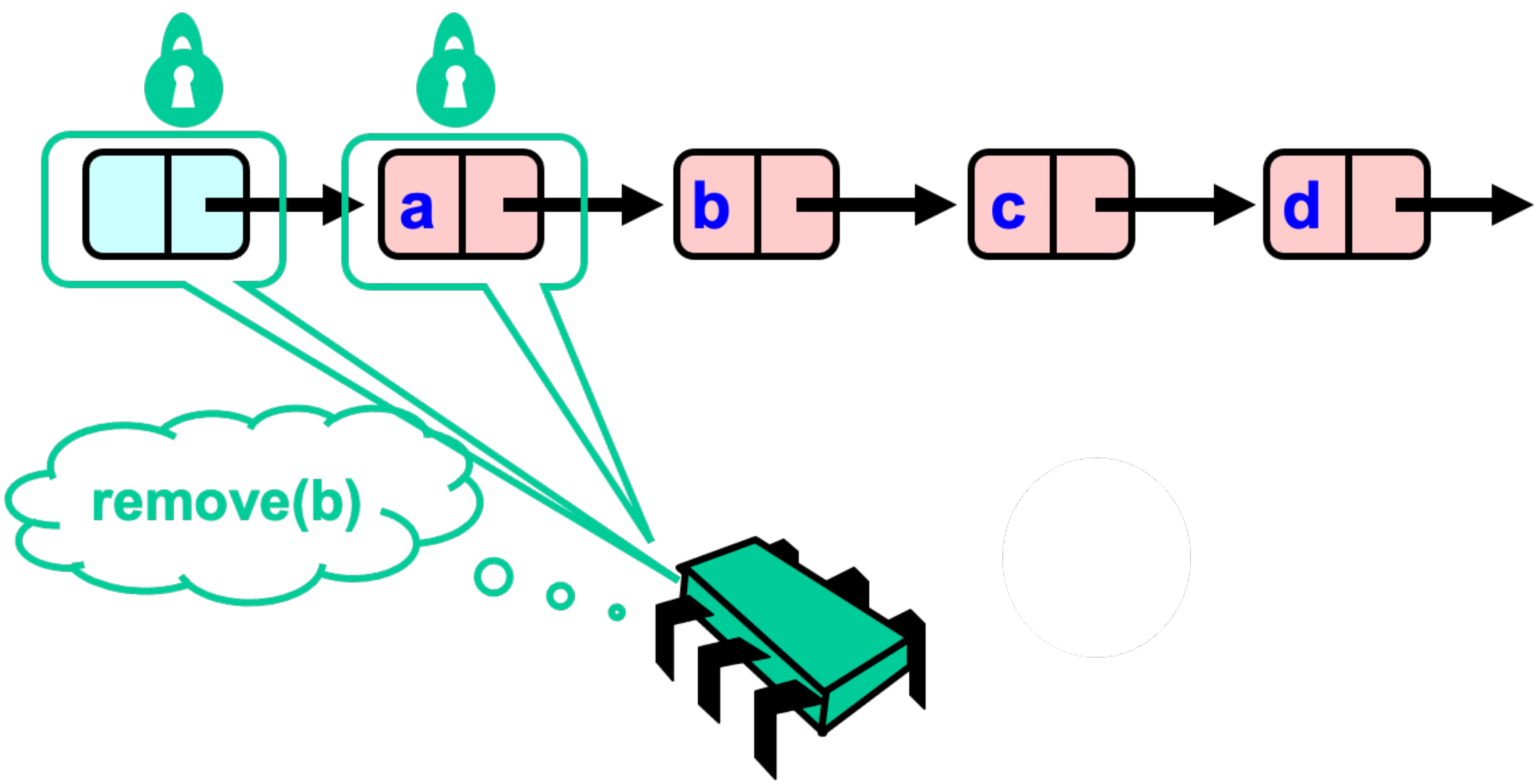
Fine Grained Locking

- Implementing remove
  - Problem arise when other threads try to access an adjacent node

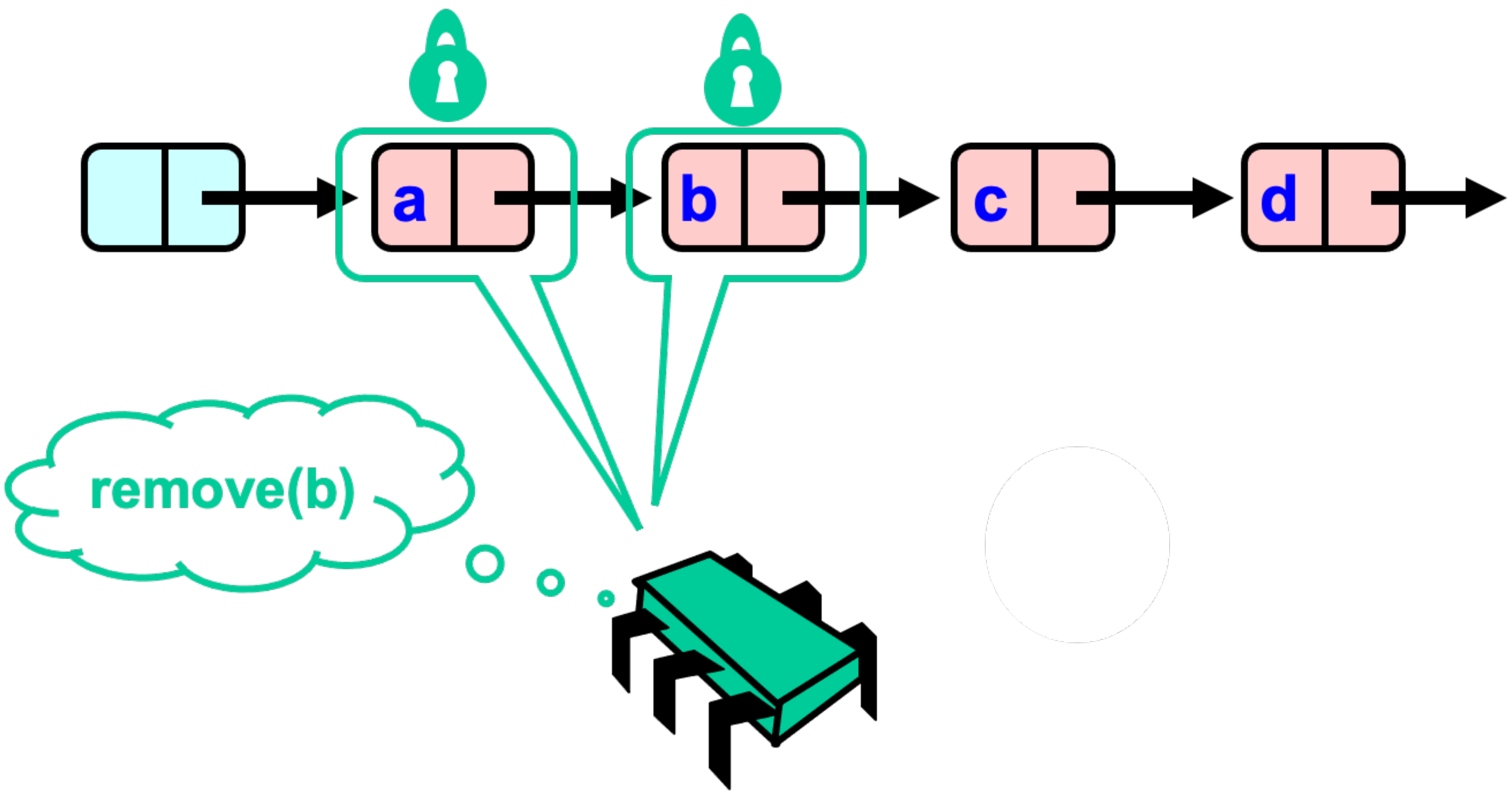




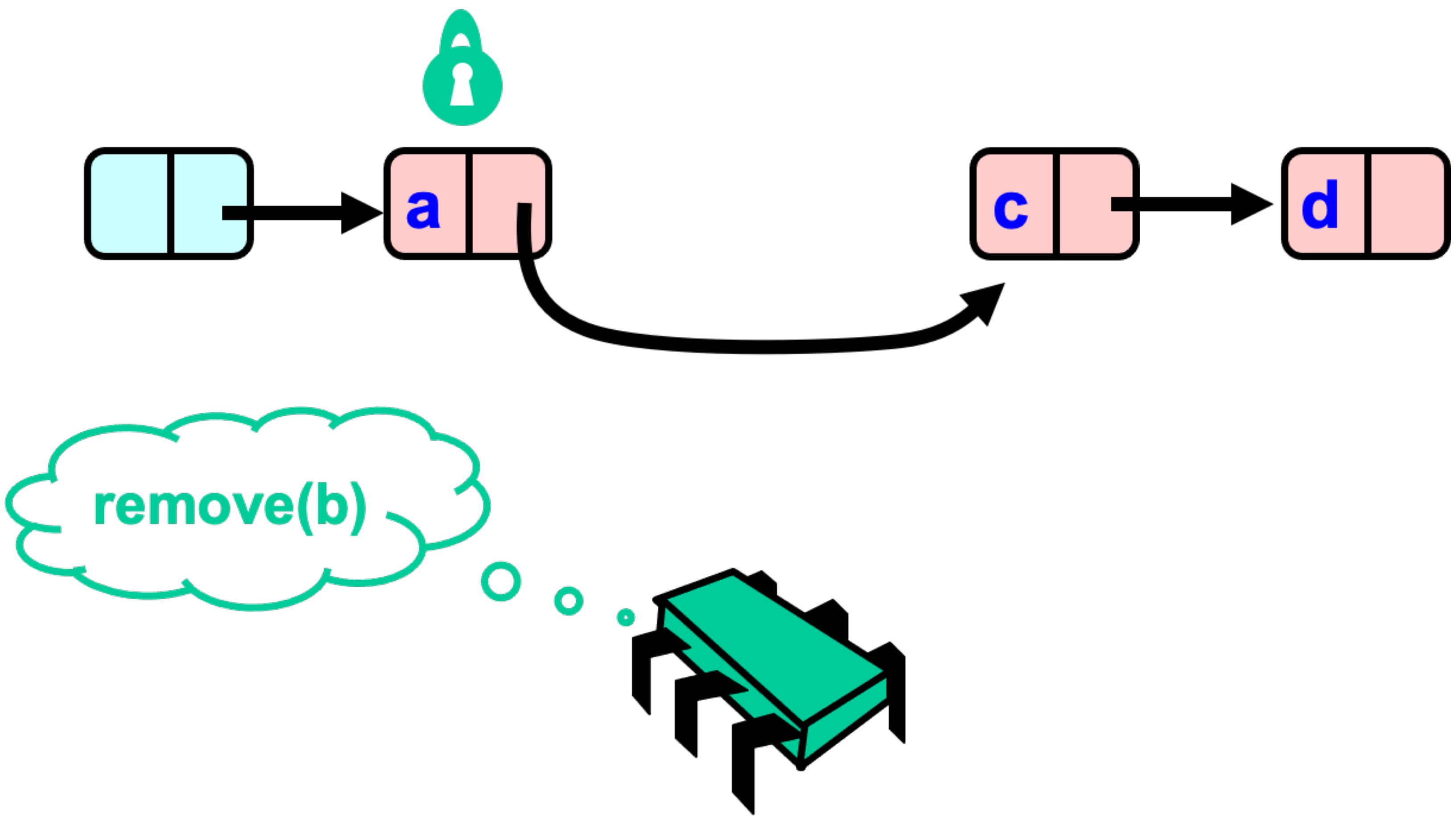
Fine Grained Locking

- Hand-to-hand locking assures that a thread that tries a competitive operation has a lock conflict
- 

Fine Grained Locking

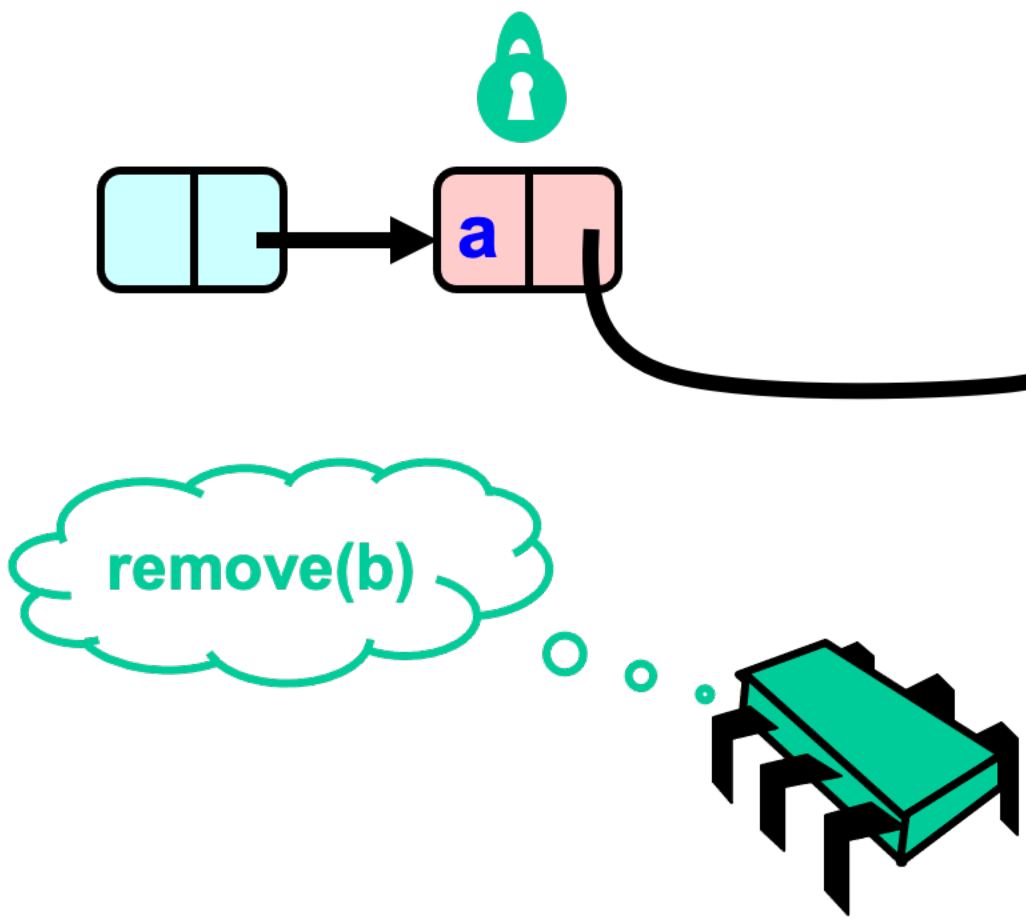


Fine Grained Locking



Fine Grained Locking

- Why lock the victim node?

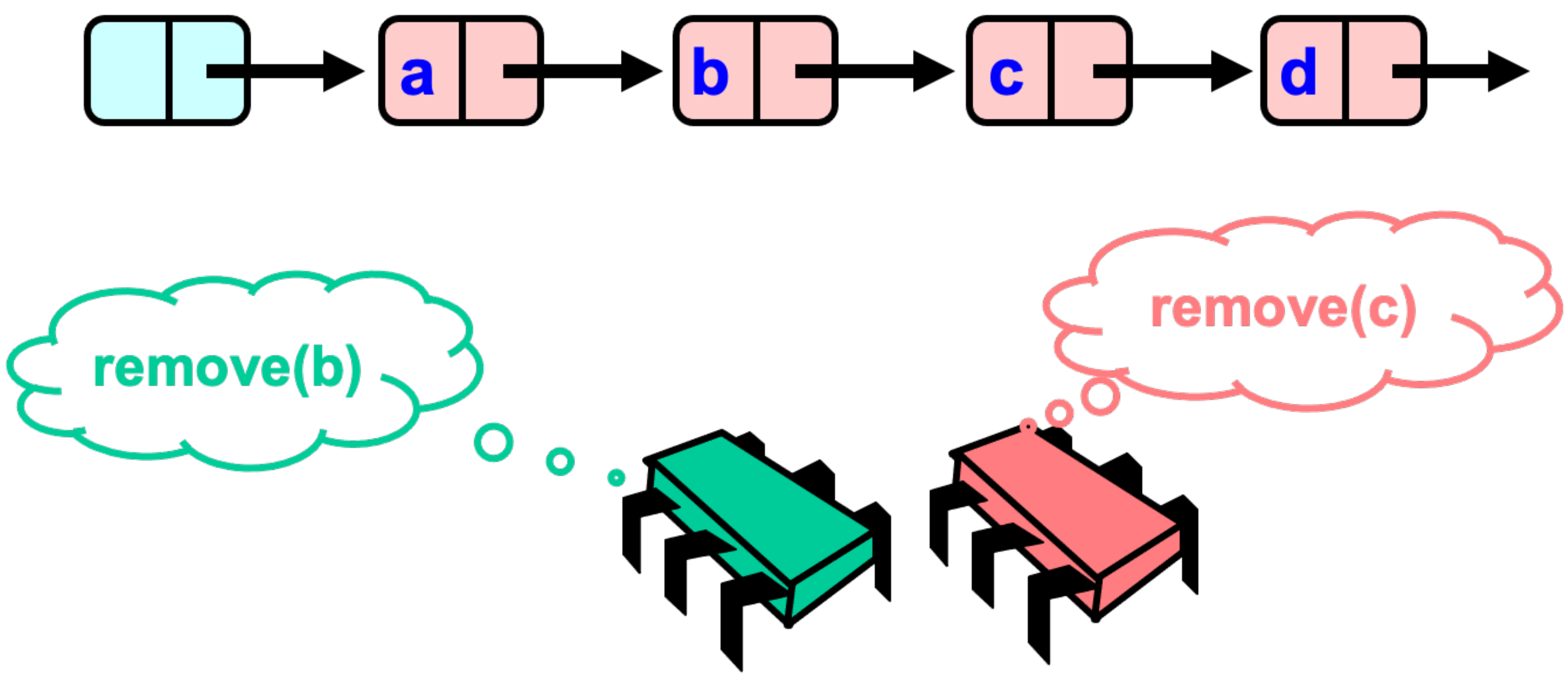


Fine Grained Locking

- Another thread might want to add after b
    - Homework 3

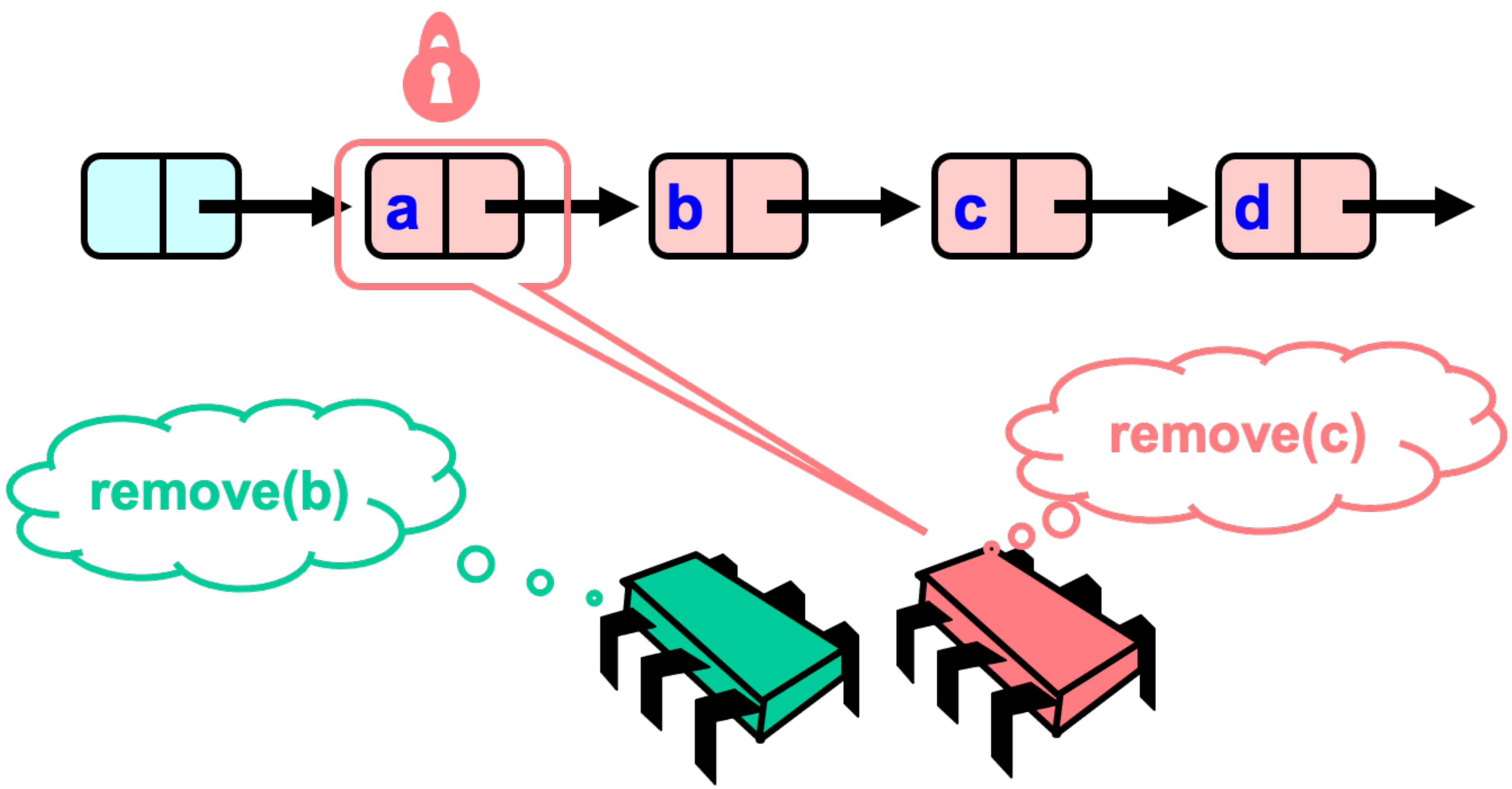
Fine Grained Locking

- Concurrent Removal



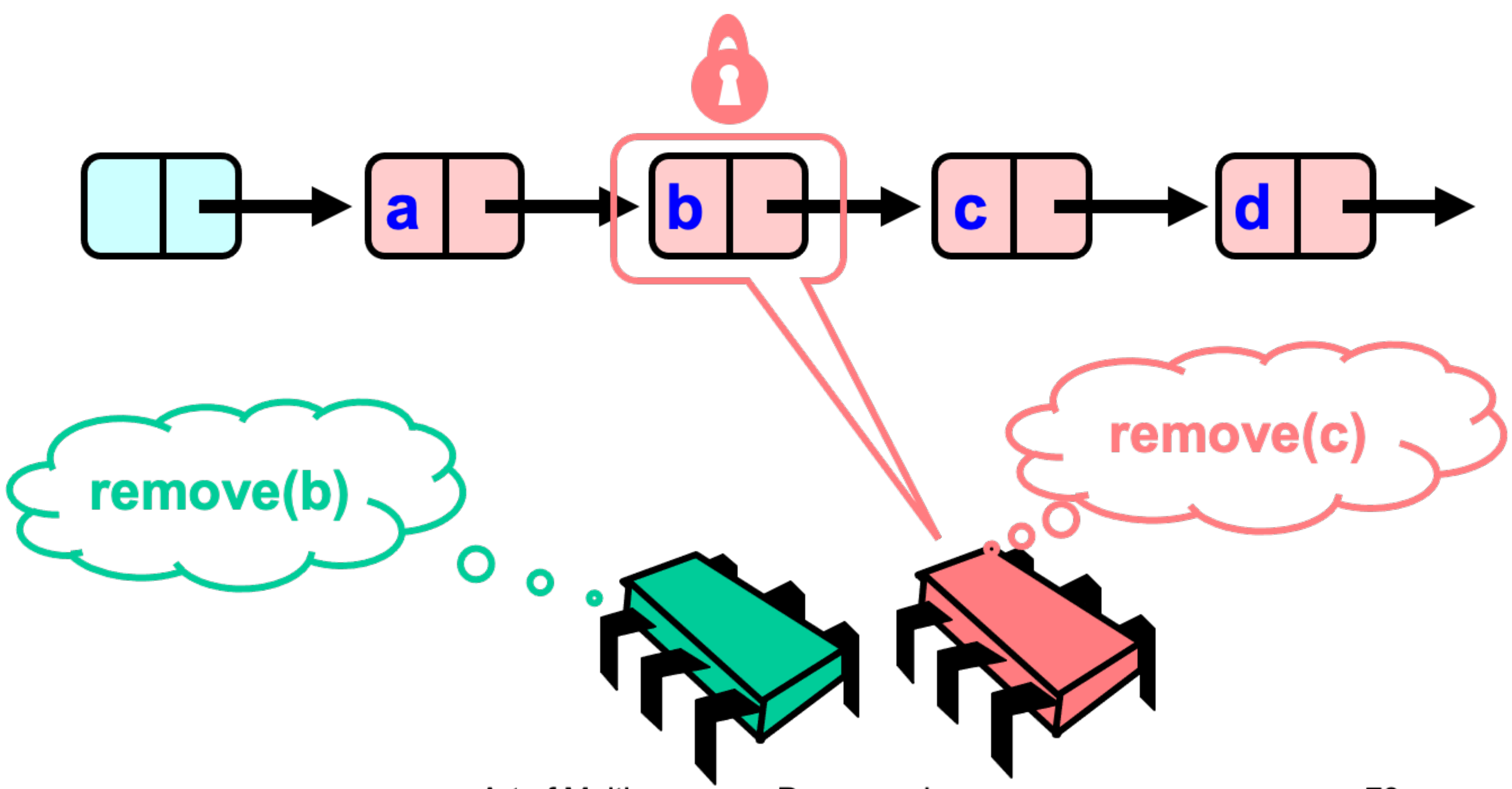
Fine Grained Locking

- Concurrent Removal



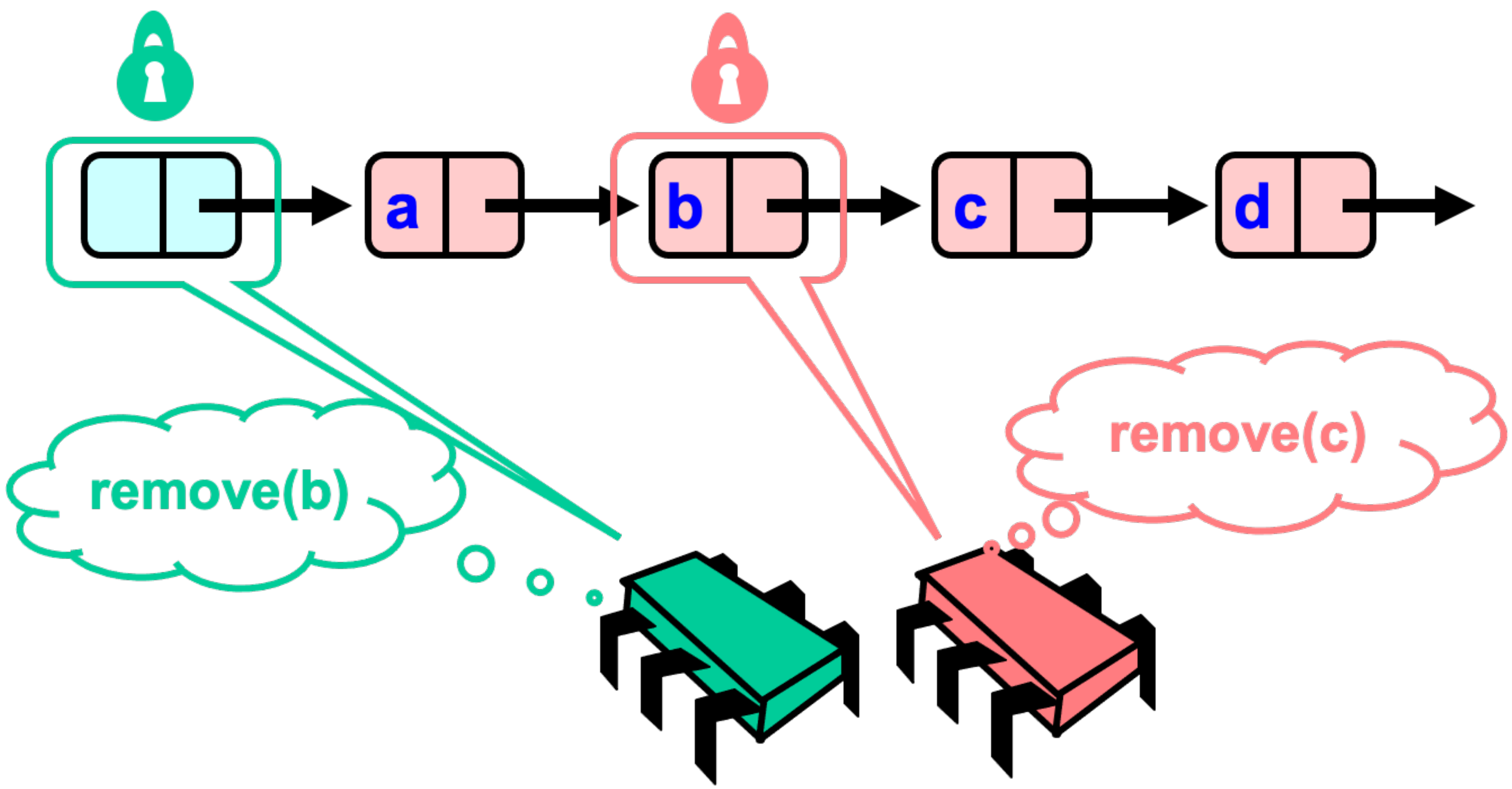
Fine Grained Locking

- Concurrent Removal



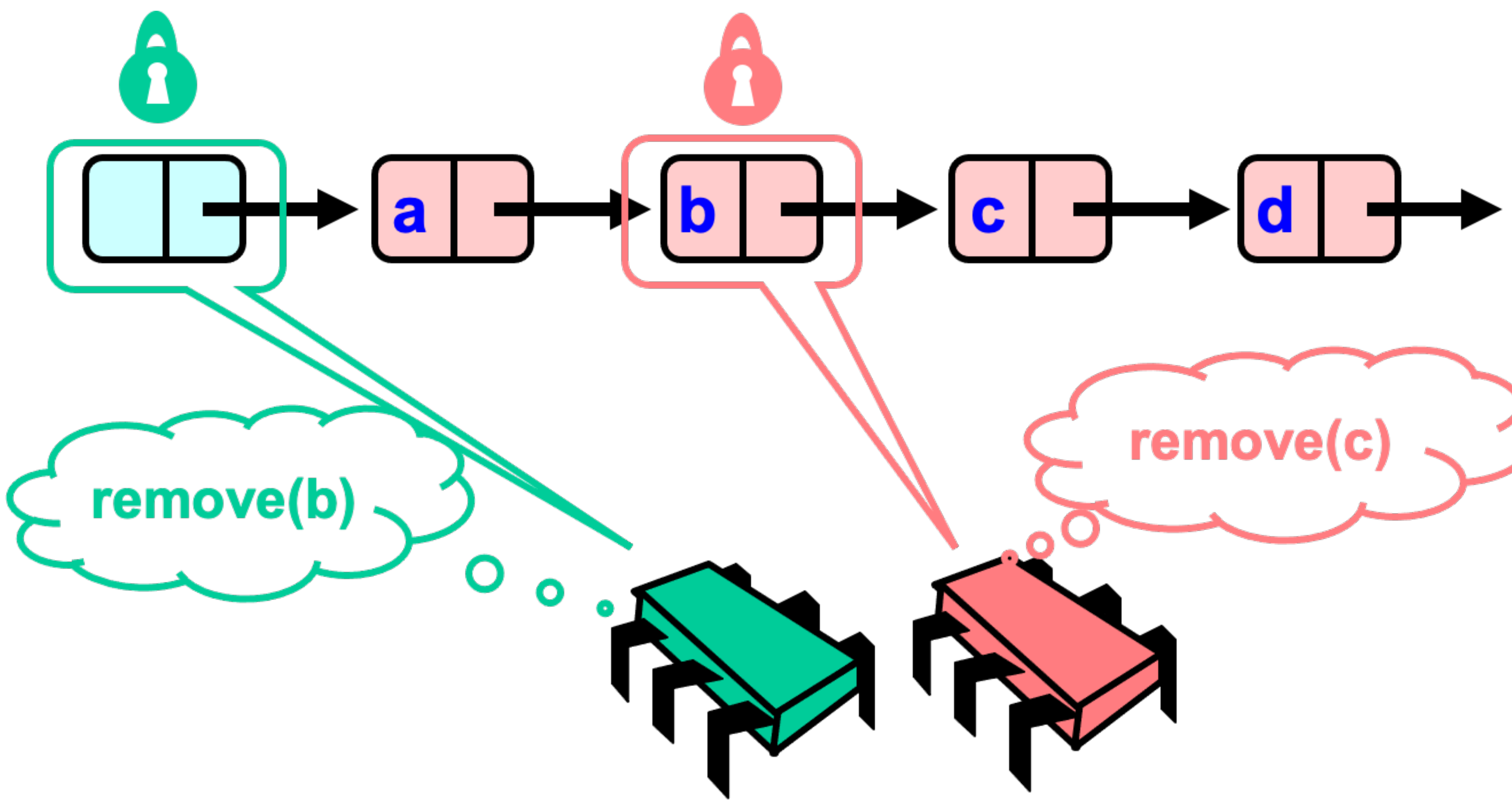
Fine Grained Locking

- Concurrent Removal



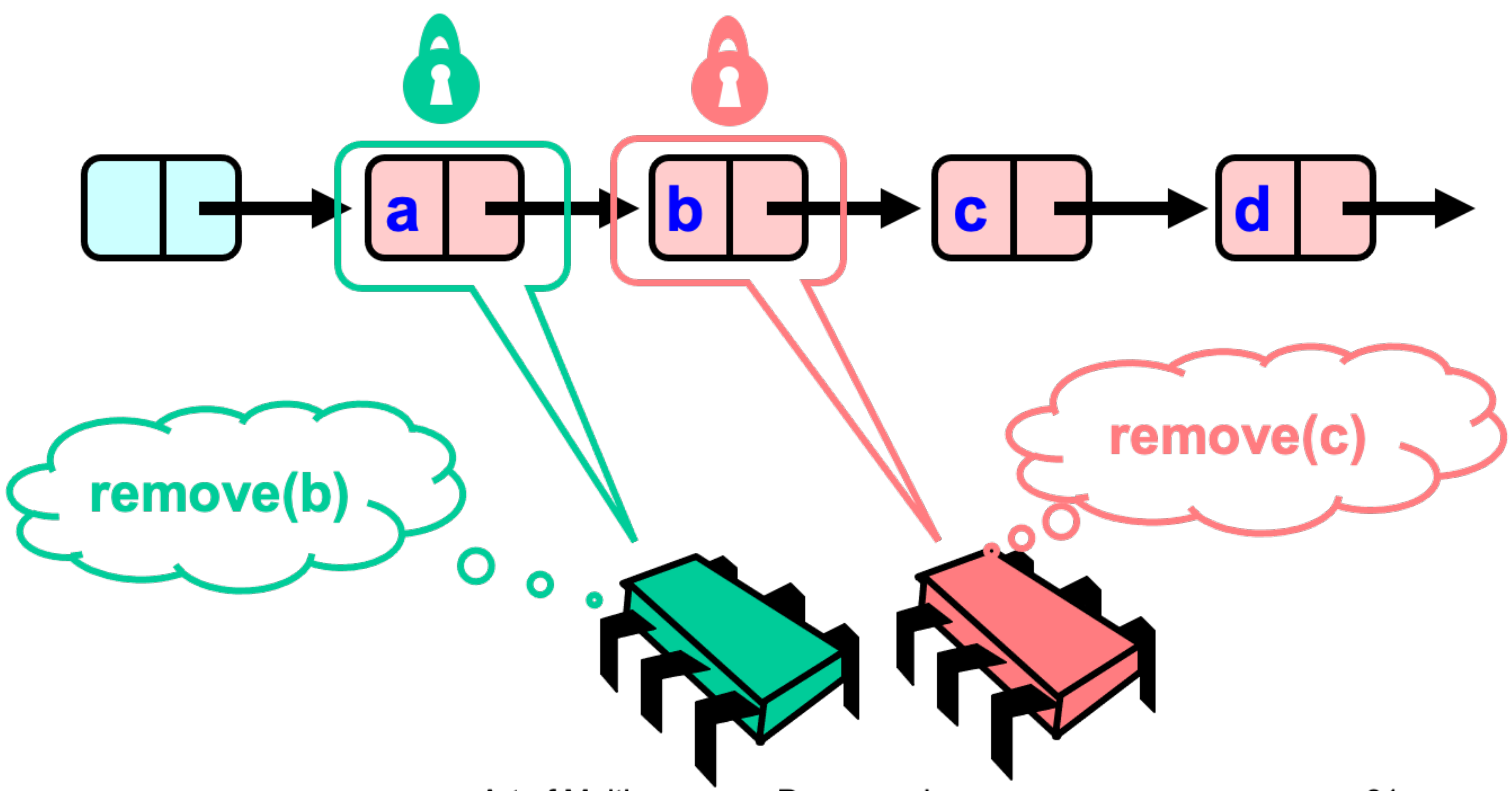
Fine Grained Locking

- Concurrent Removal



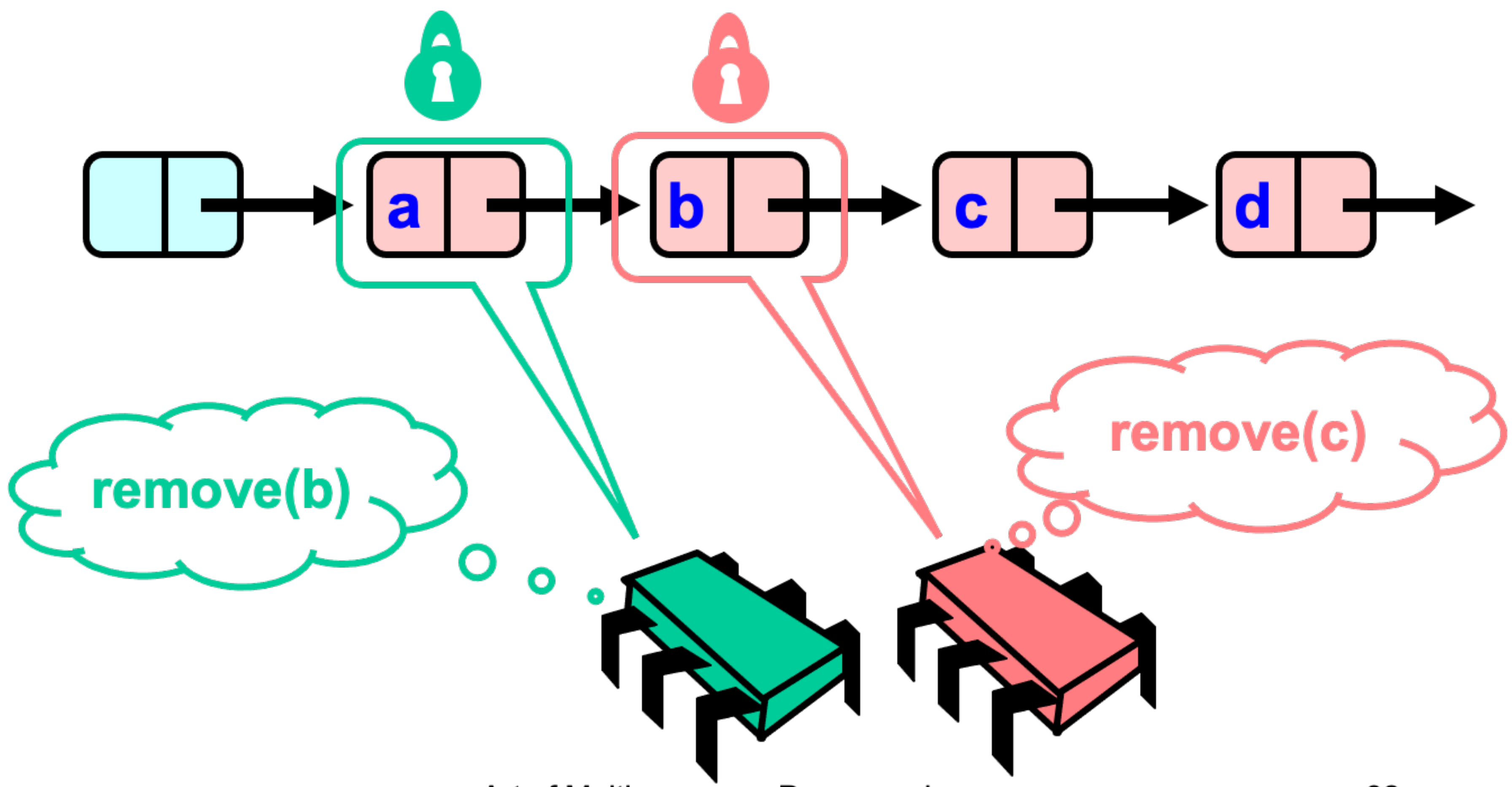
Fine Grained Locking

- Concurrent Removal



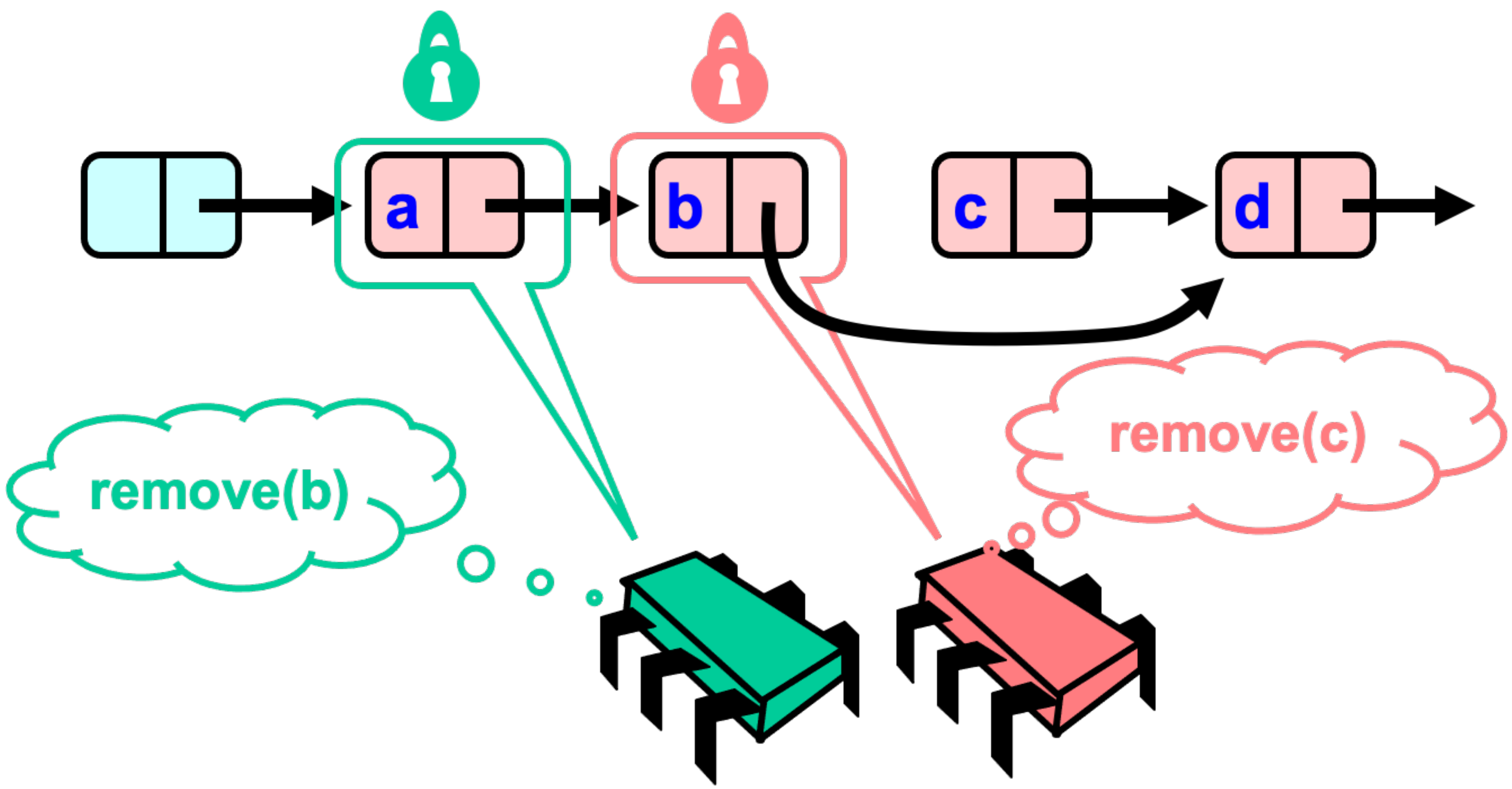
Fine Grained Locking

- Concurrent Removal



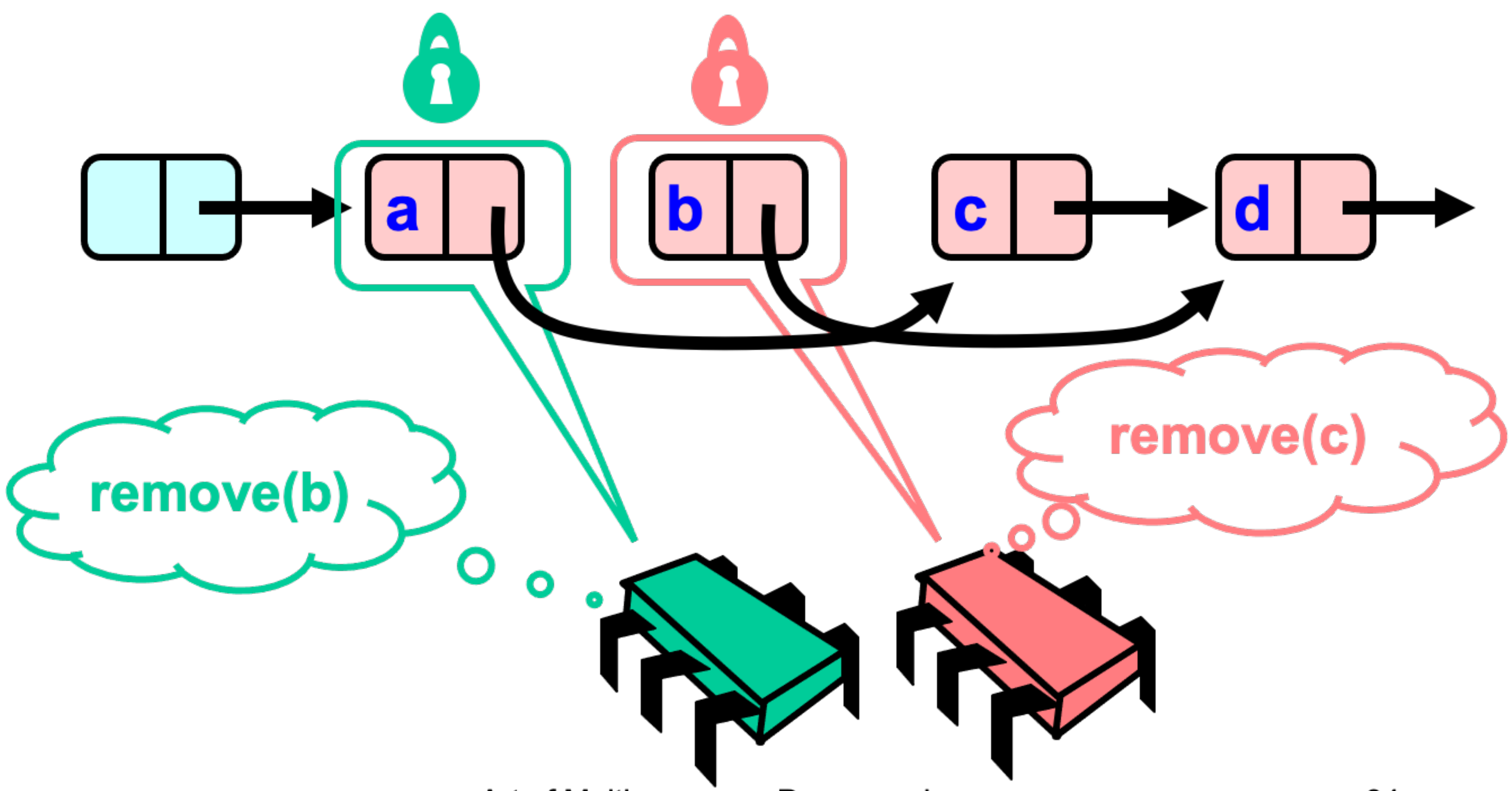
Fine Grained Locking

- Concurrent Removal



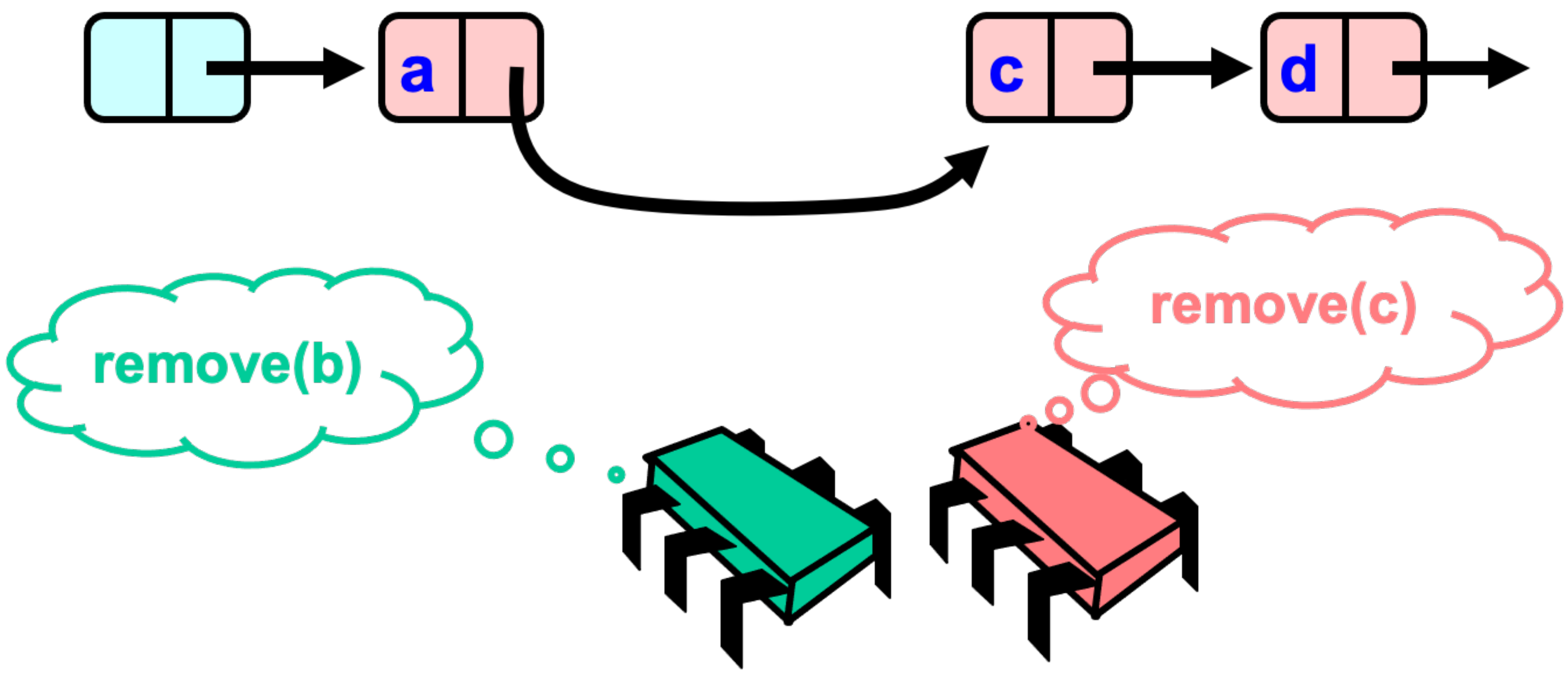
Fine Grained Locking

- Concurrent Removal



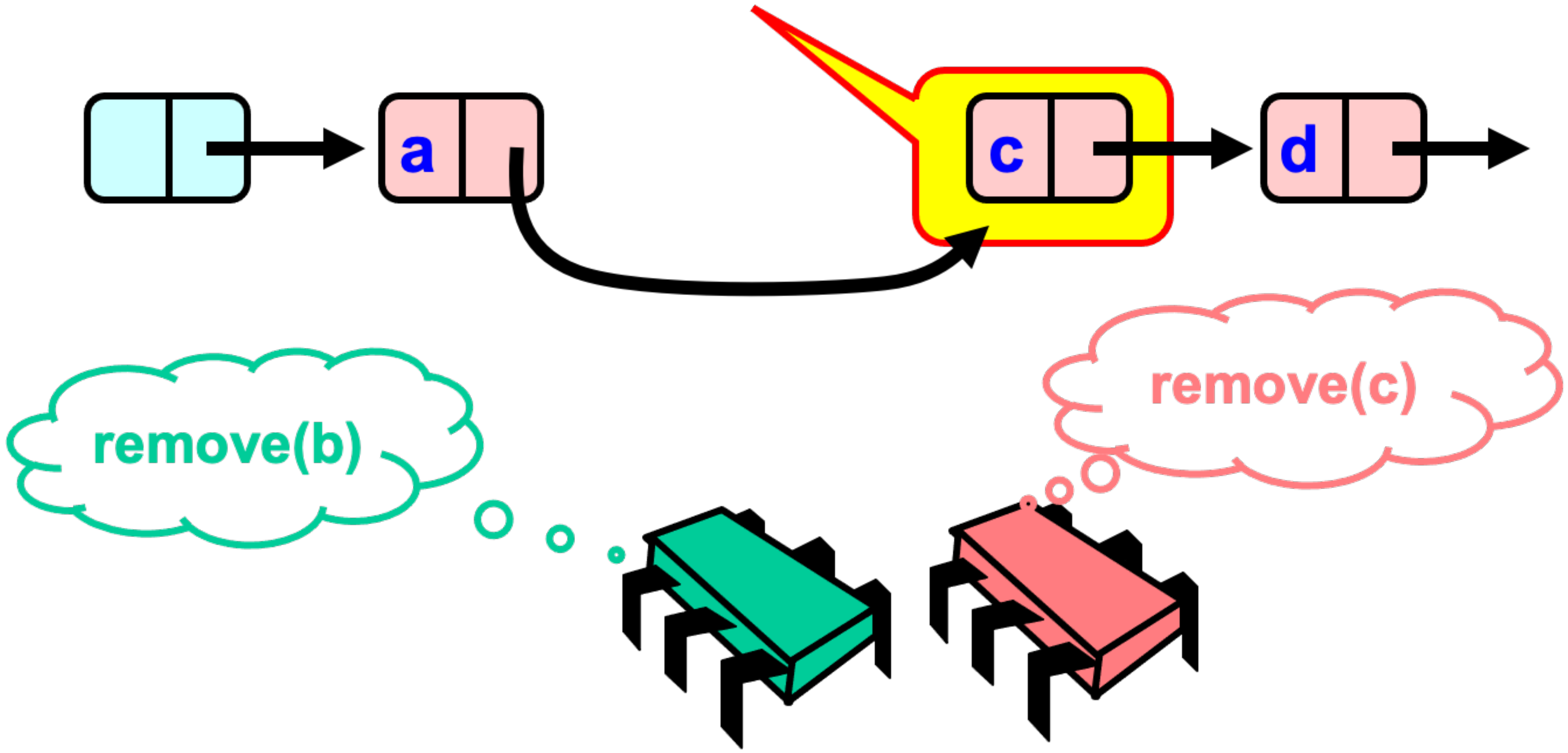
Fine Grained Locking

- Concurrent removal undoes one threads work



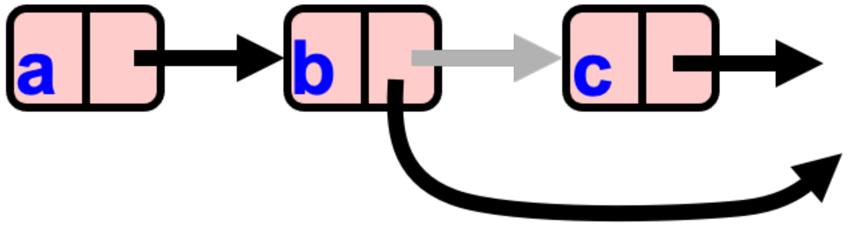
Fine Grained Locking

- Node c has not been removed

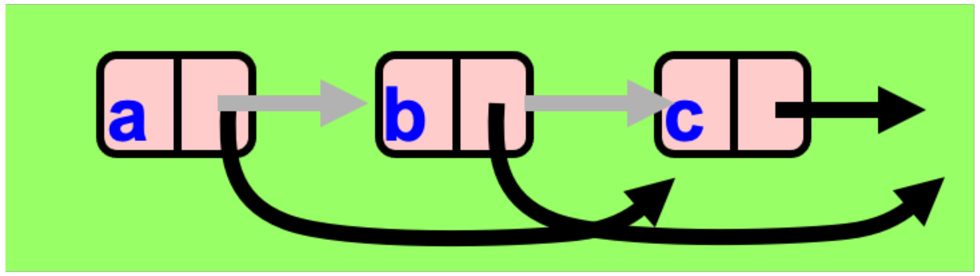


Fine Grained Locking

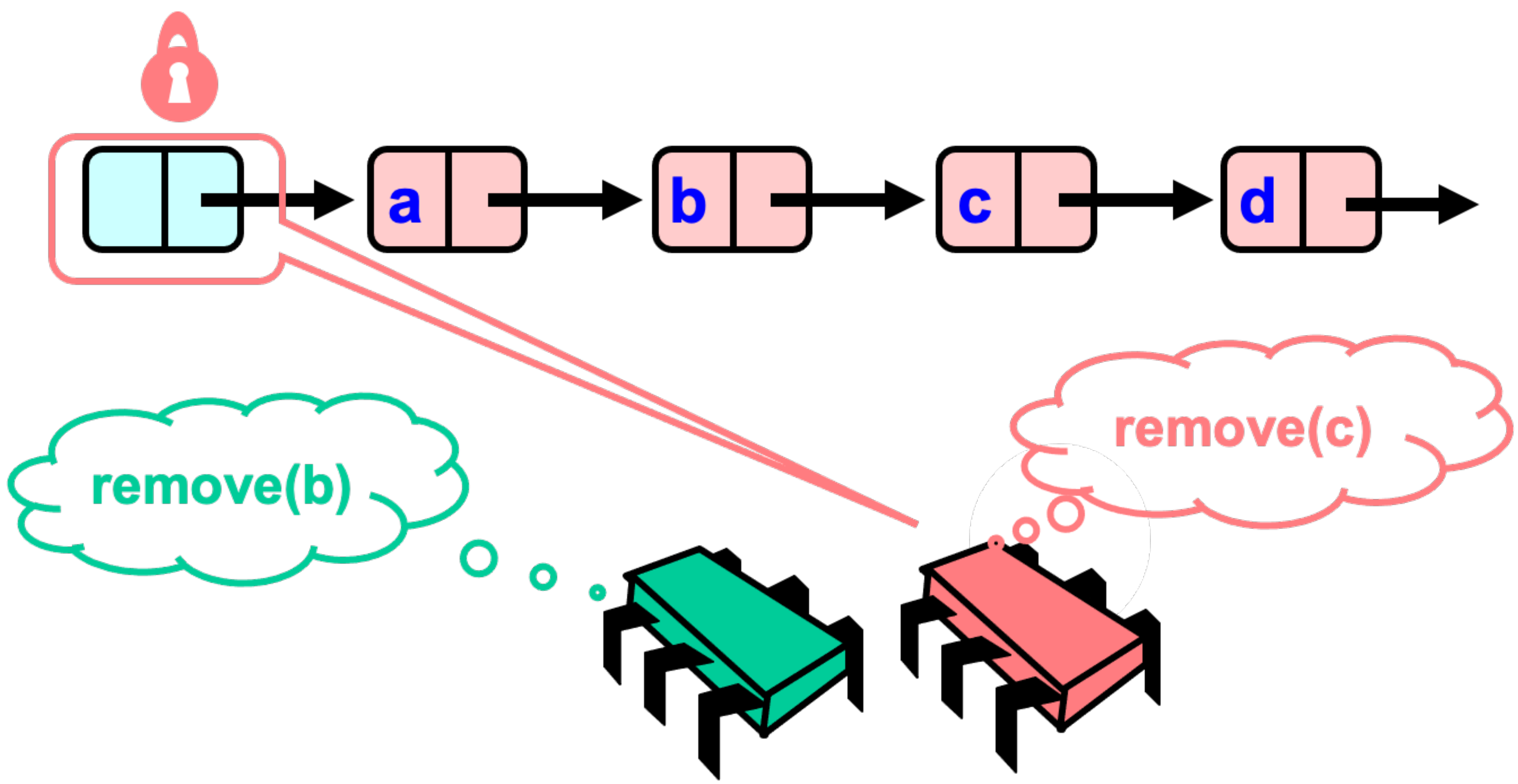
- Problem
  - To delete node C, we swing its predecessor's next-field to its successor



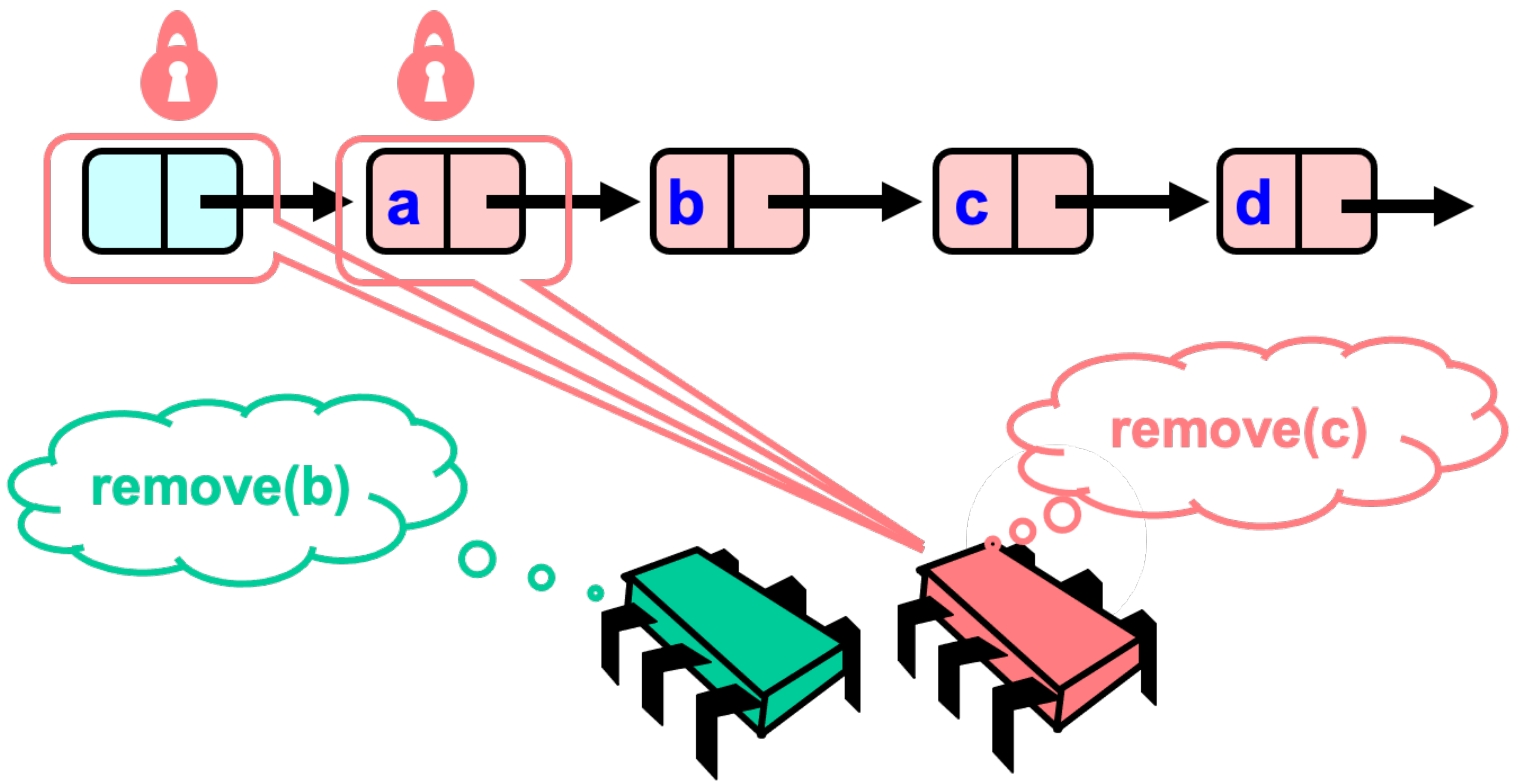
  - But someone could create another pointer to C



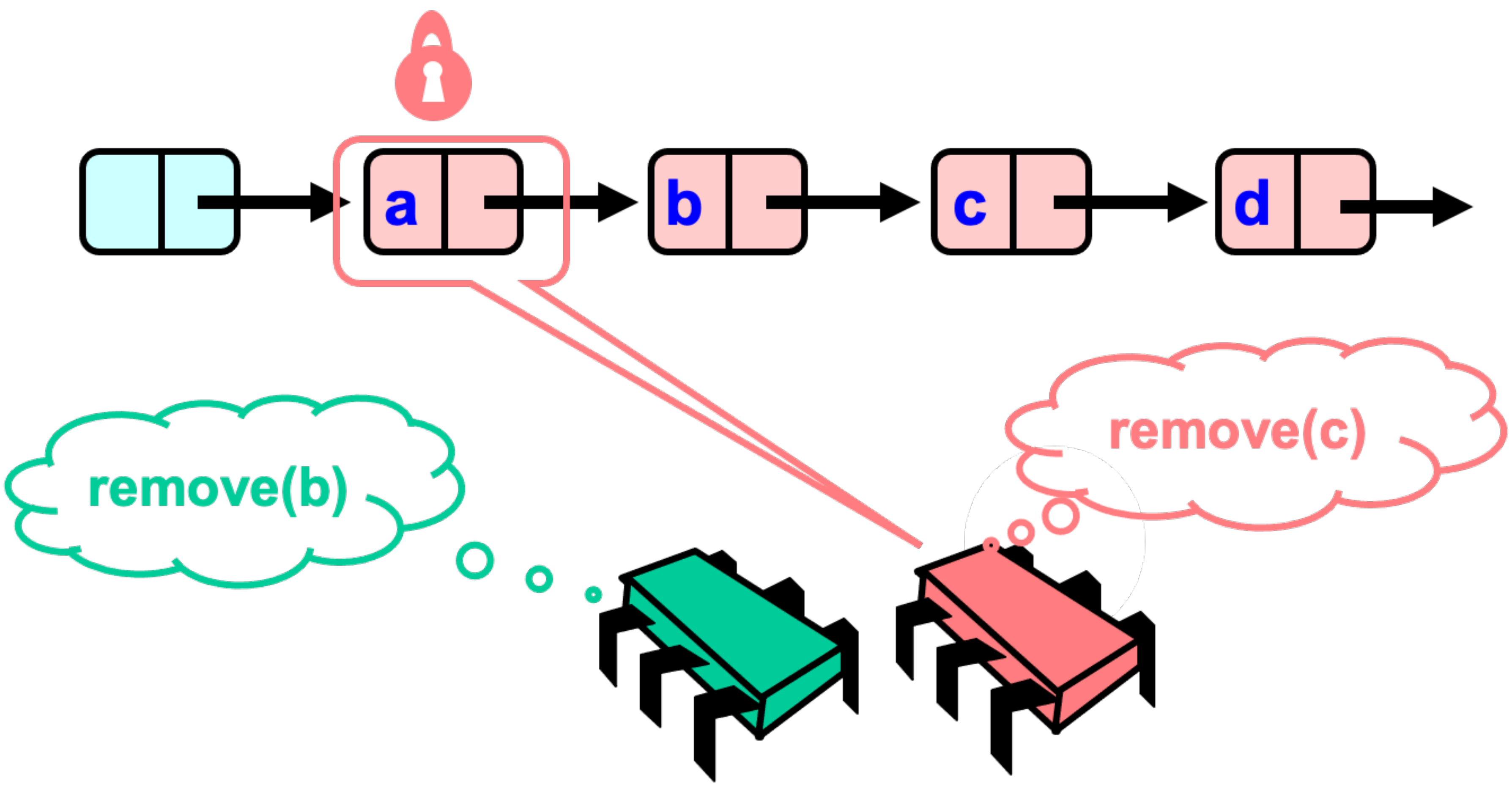
Fine Grained Locking



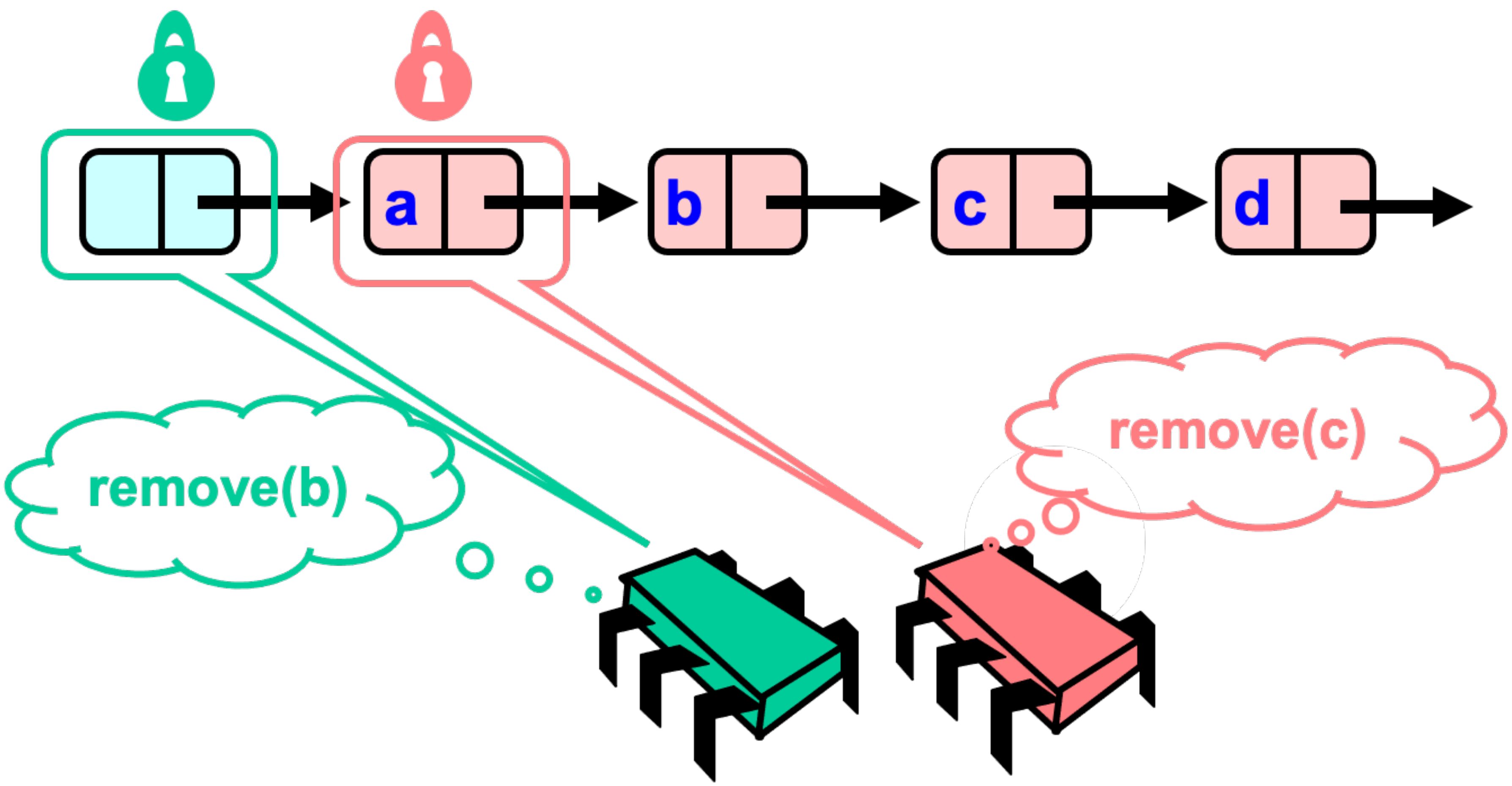
Fine Grained Locking



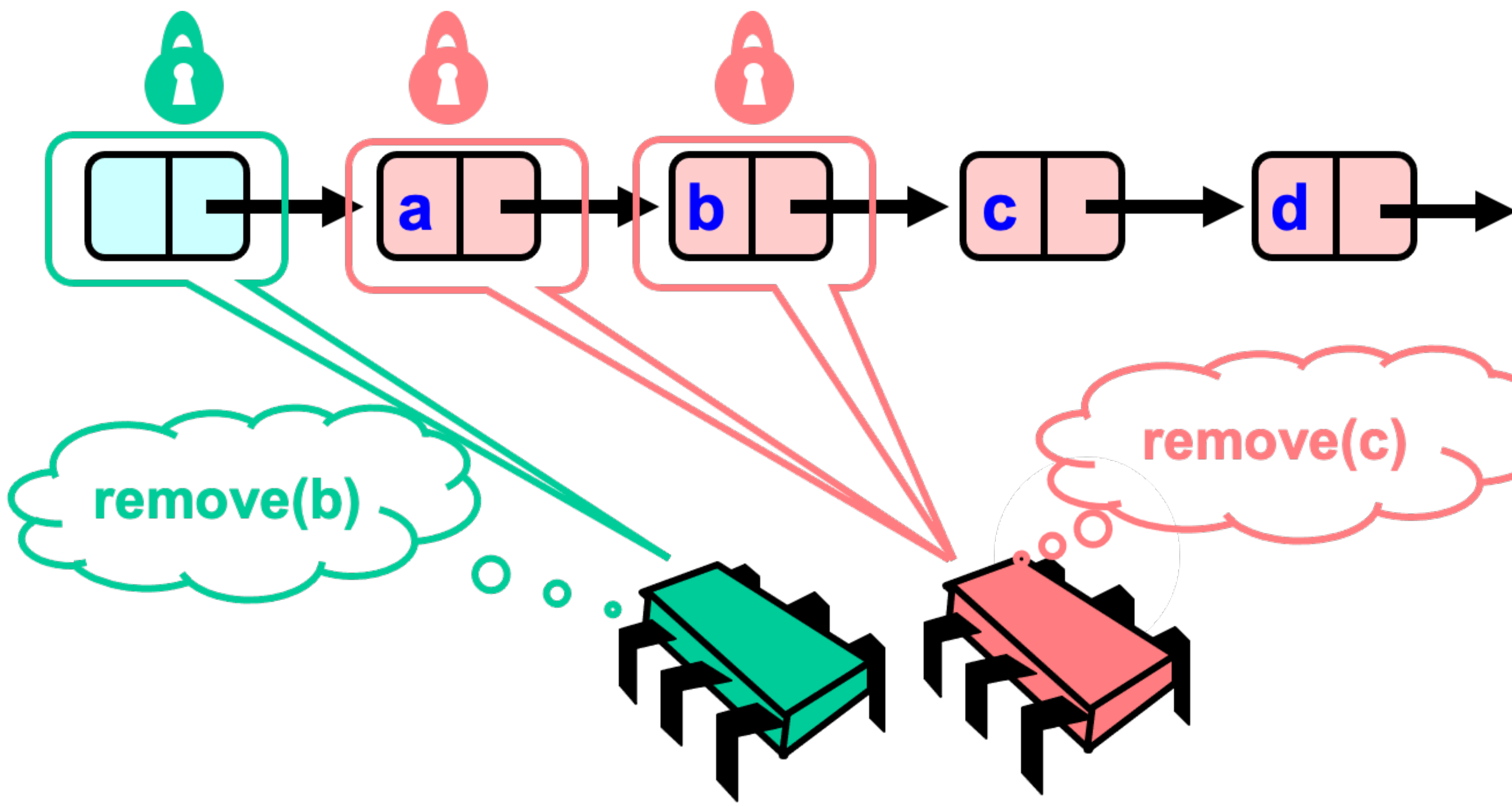
Fine Grained Locking



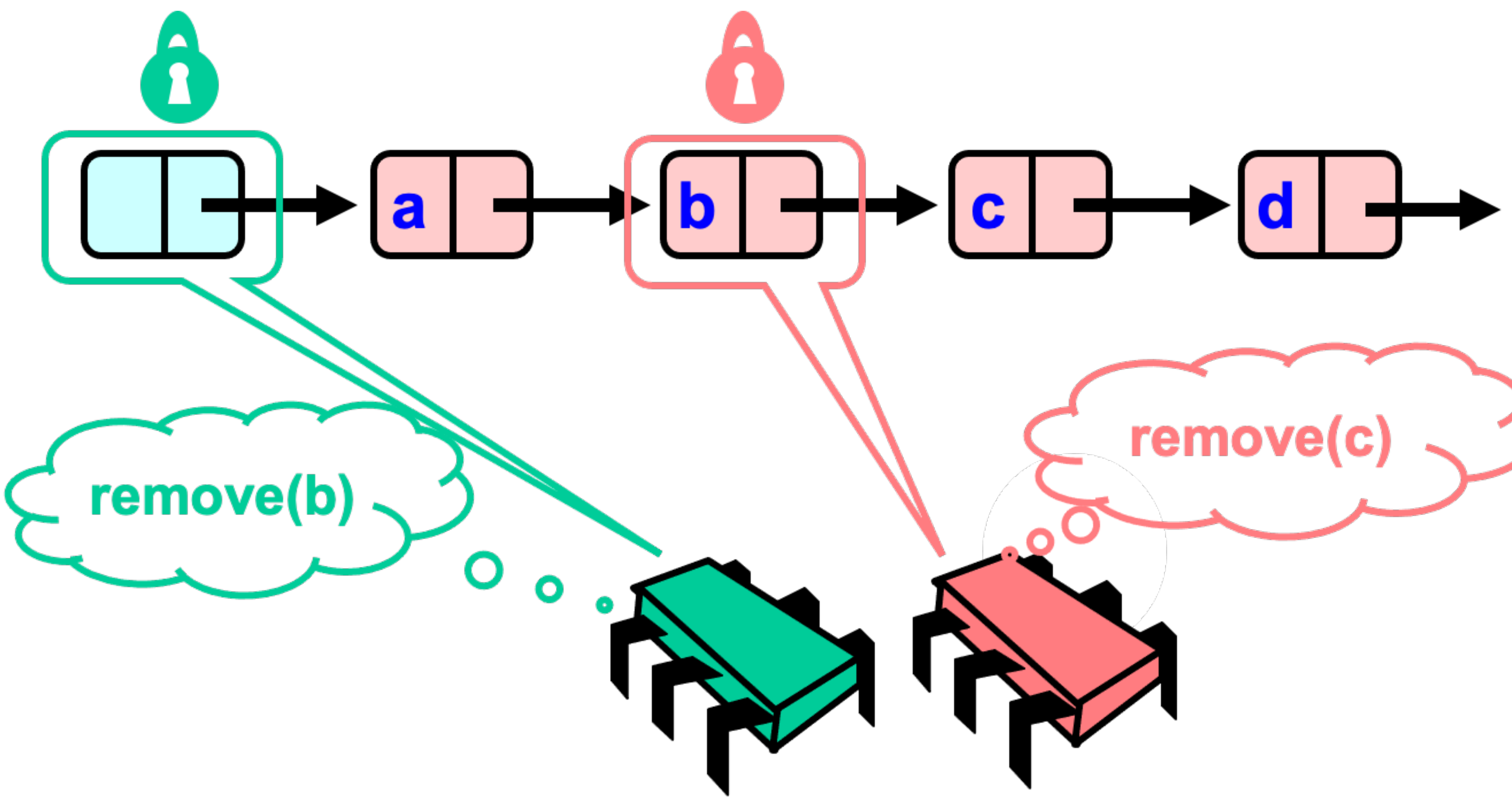
Fine Grained Locking



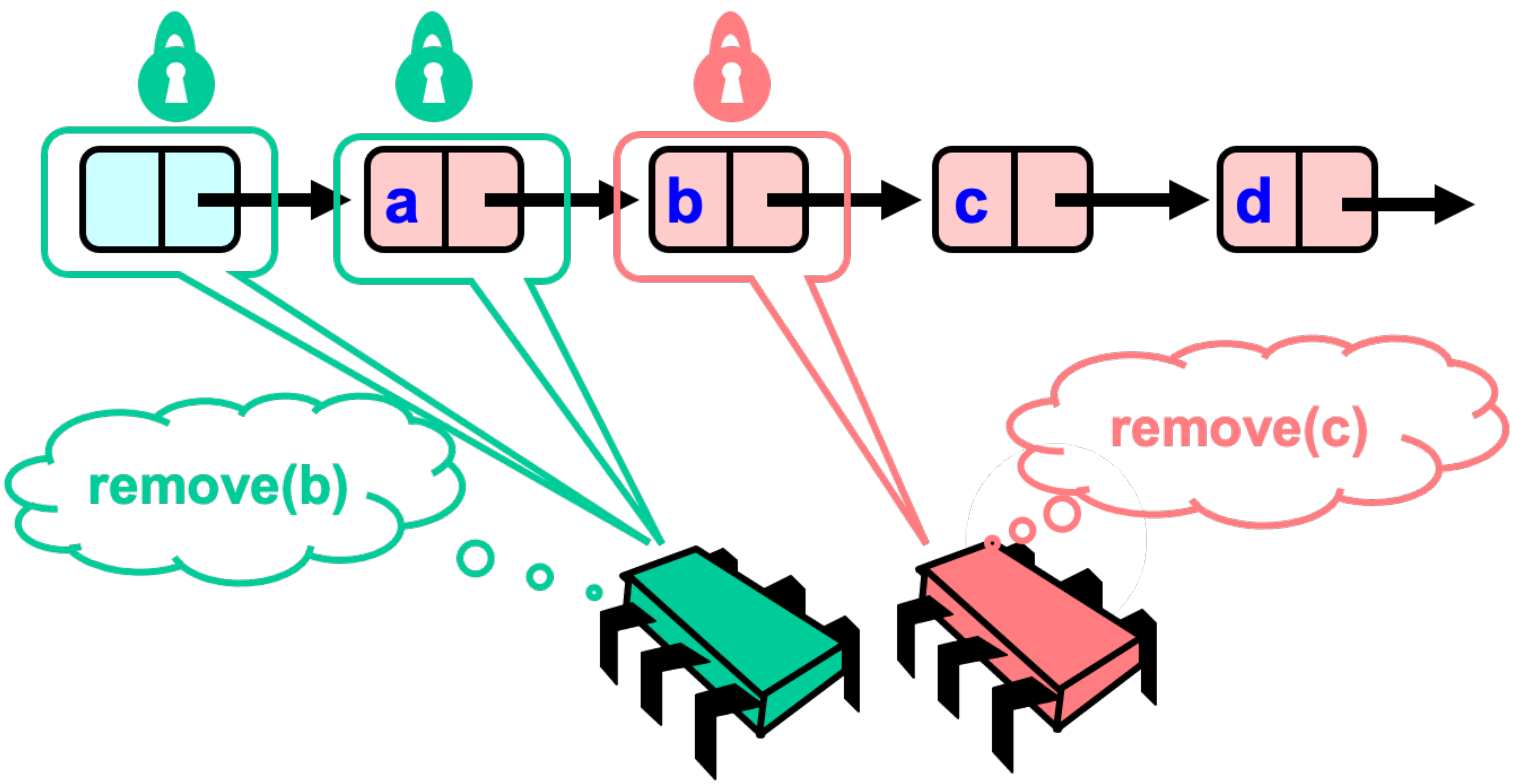
Fine Grained Locking



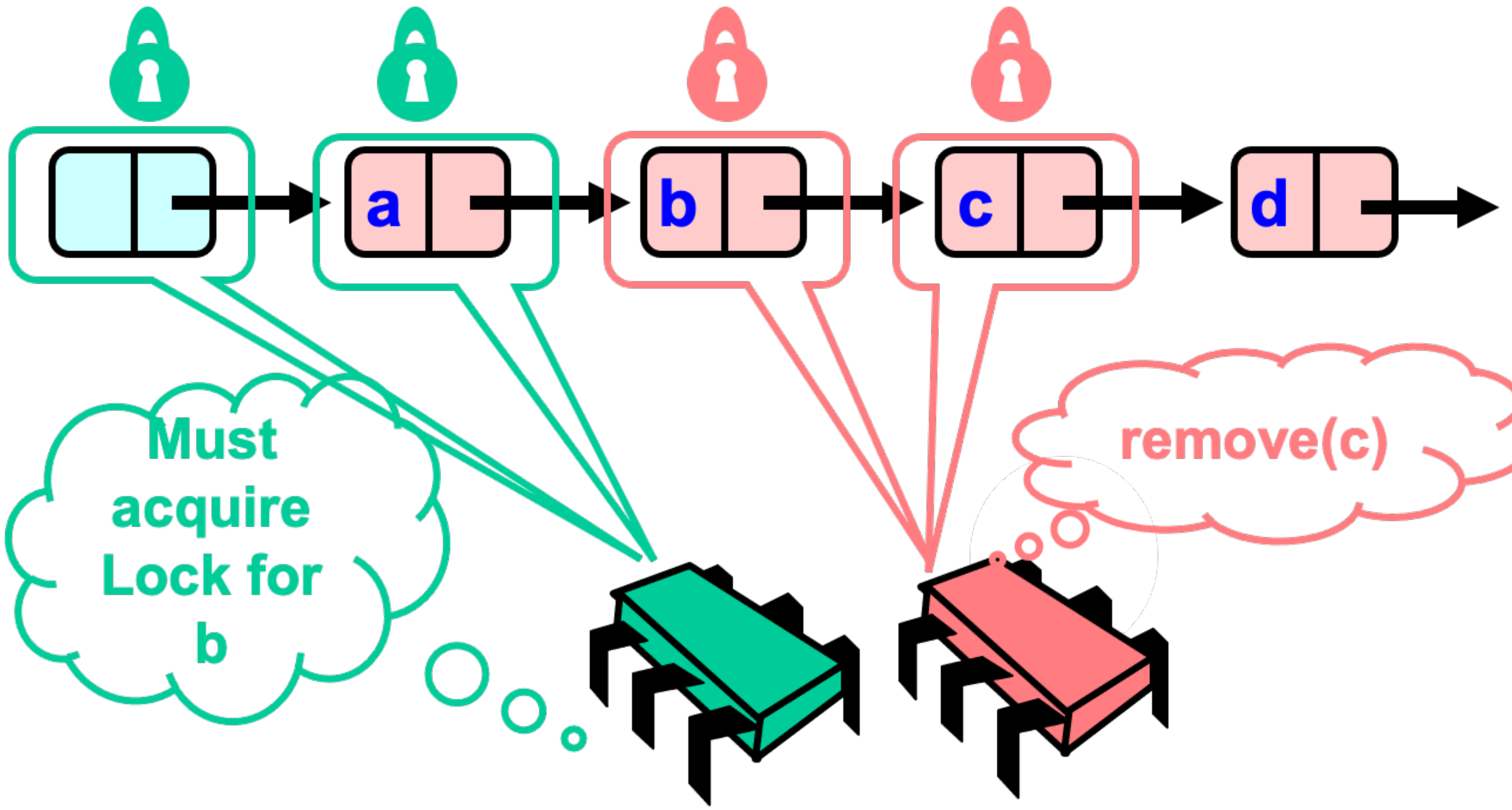
Fine Grained Locking



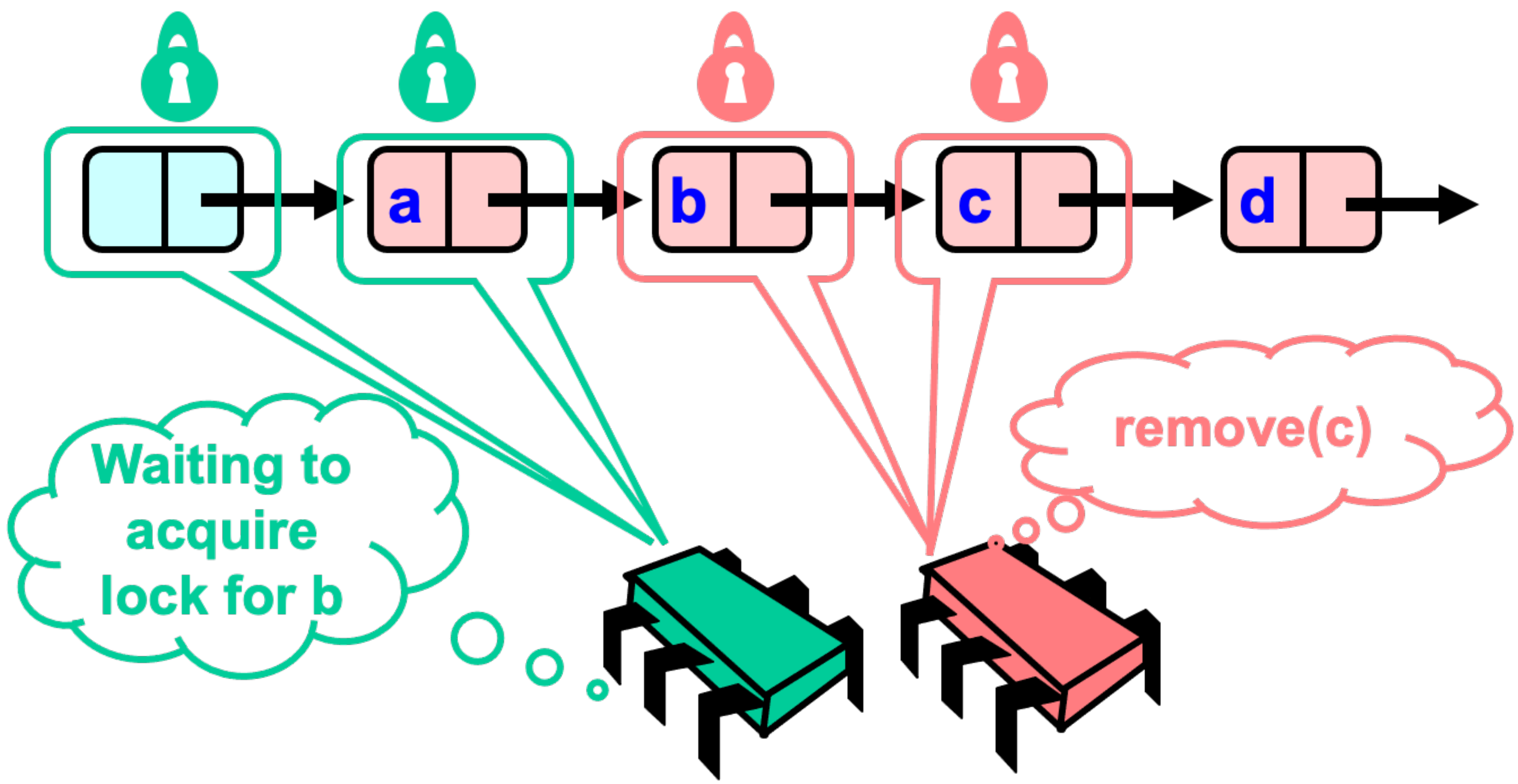
Fine Grained Locking



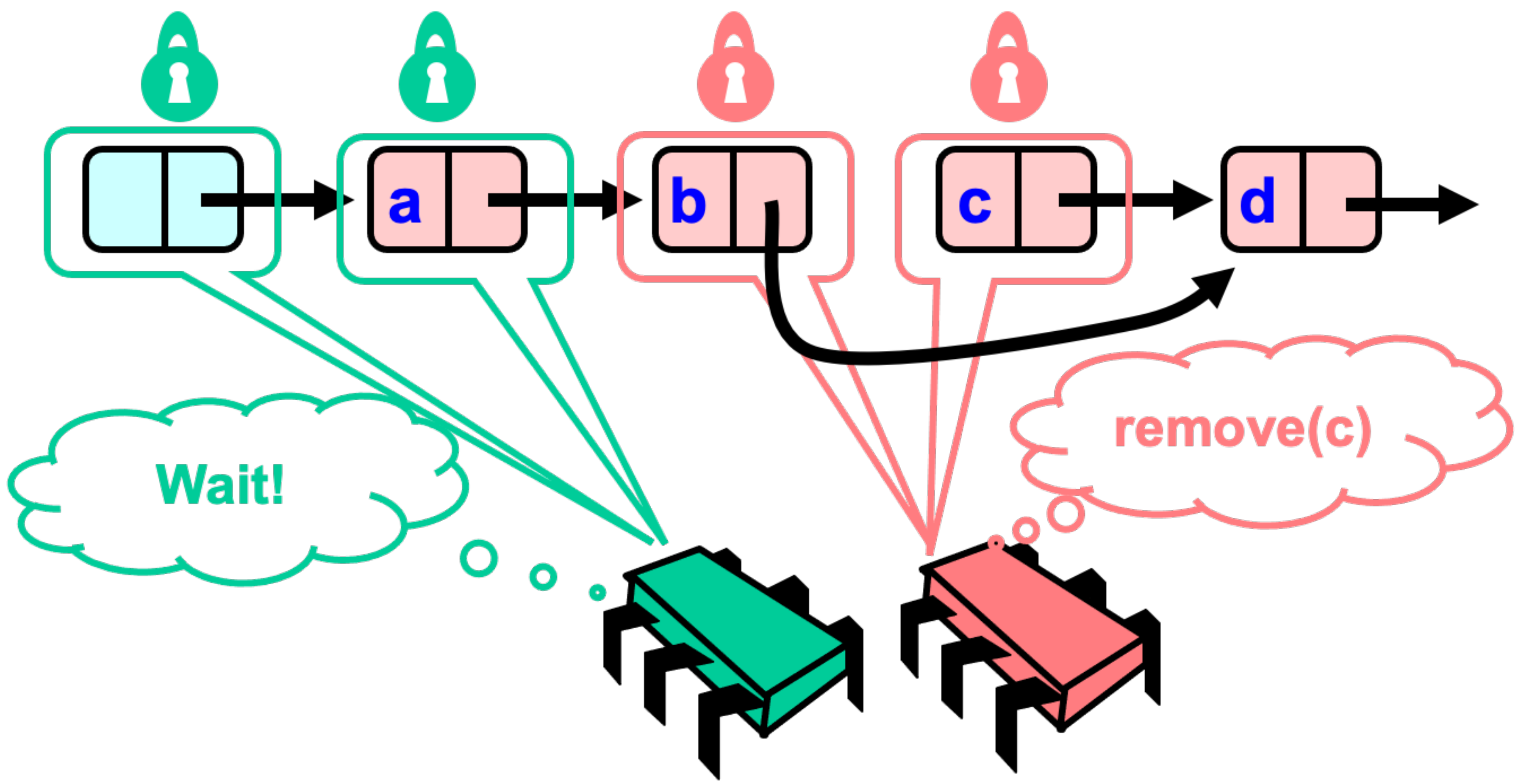
Fine Grained Locking



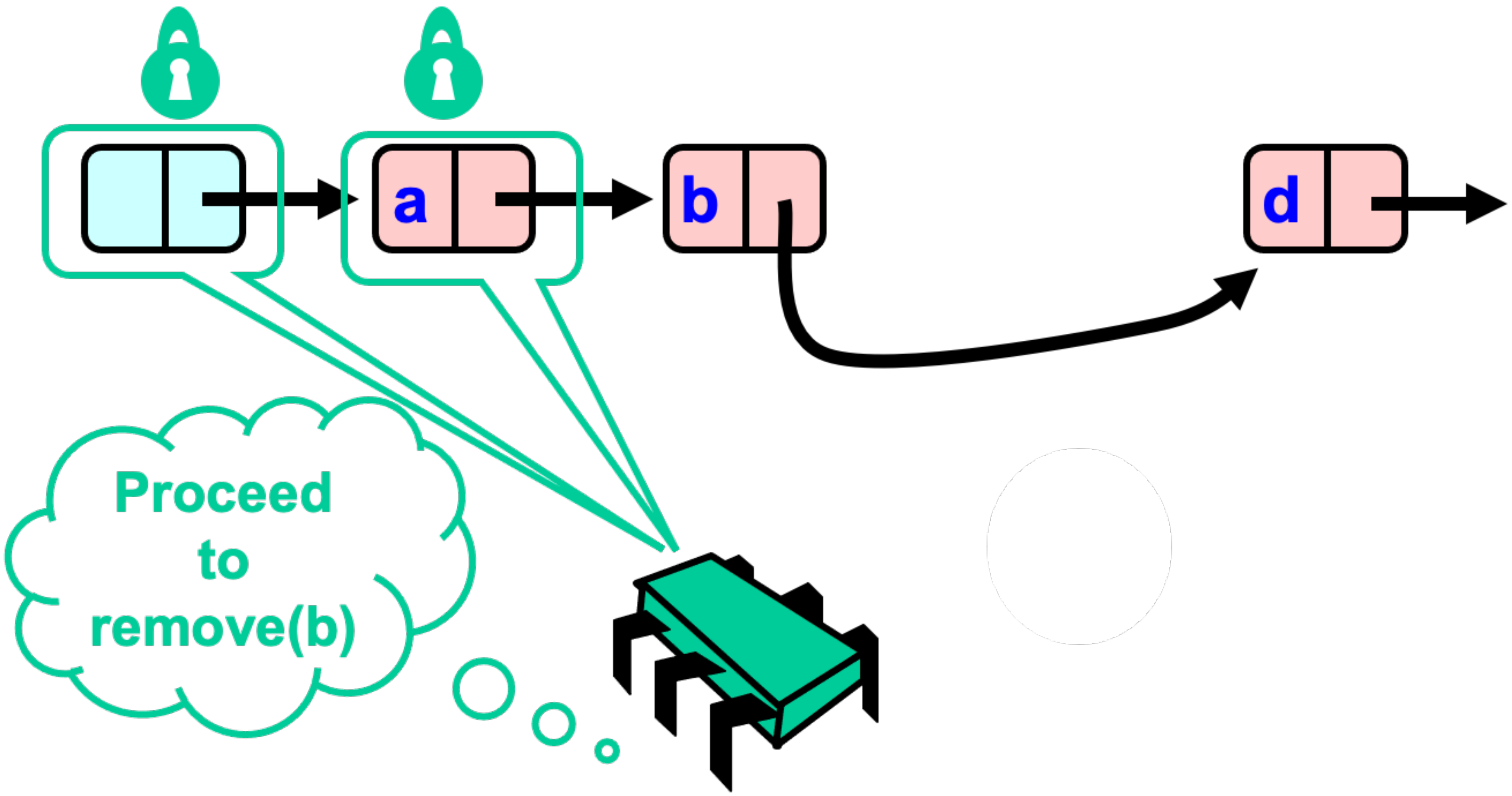
Fine Grained Locking



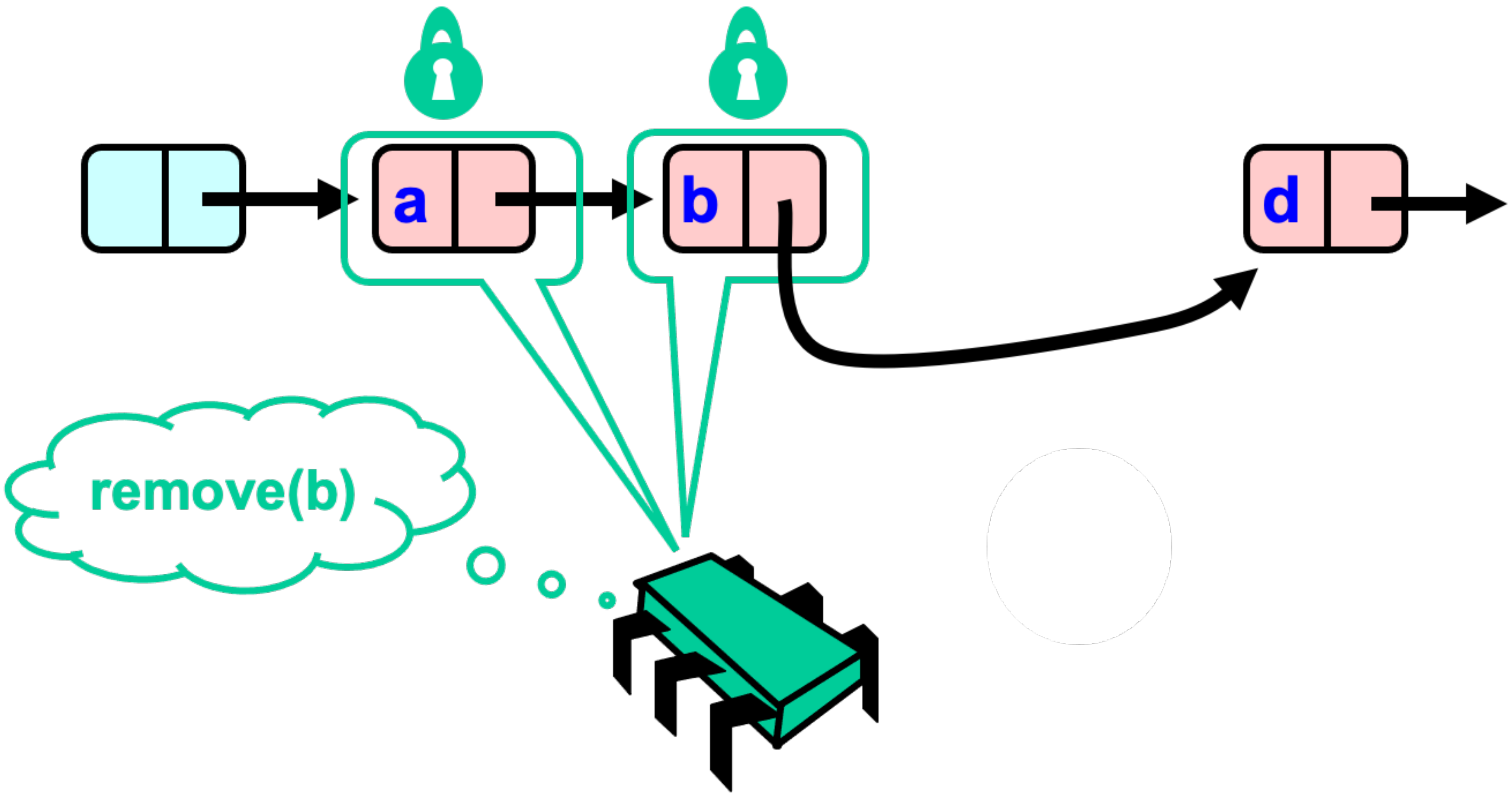
Fine Grained Locking



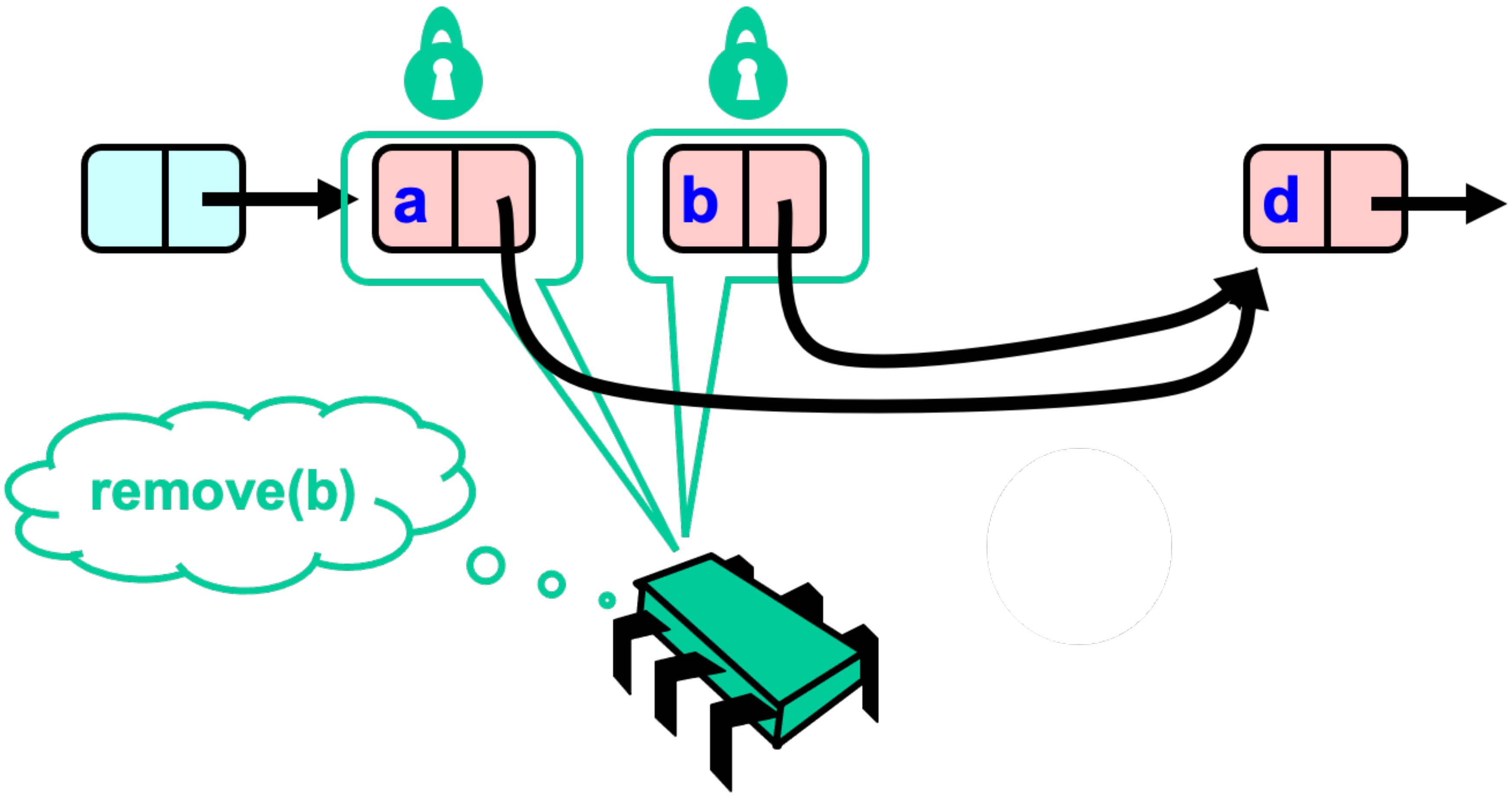
Fine Grained Locking



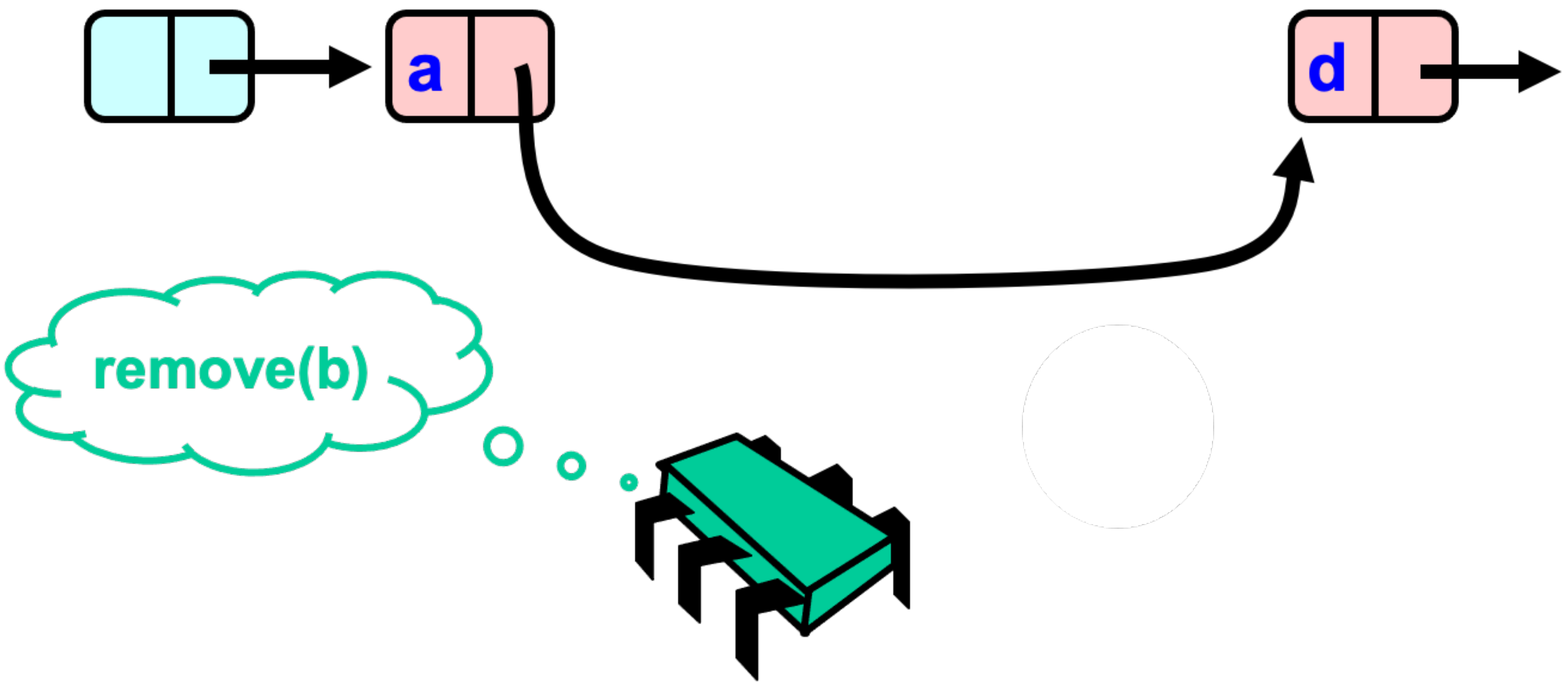
Fine Grained Locking



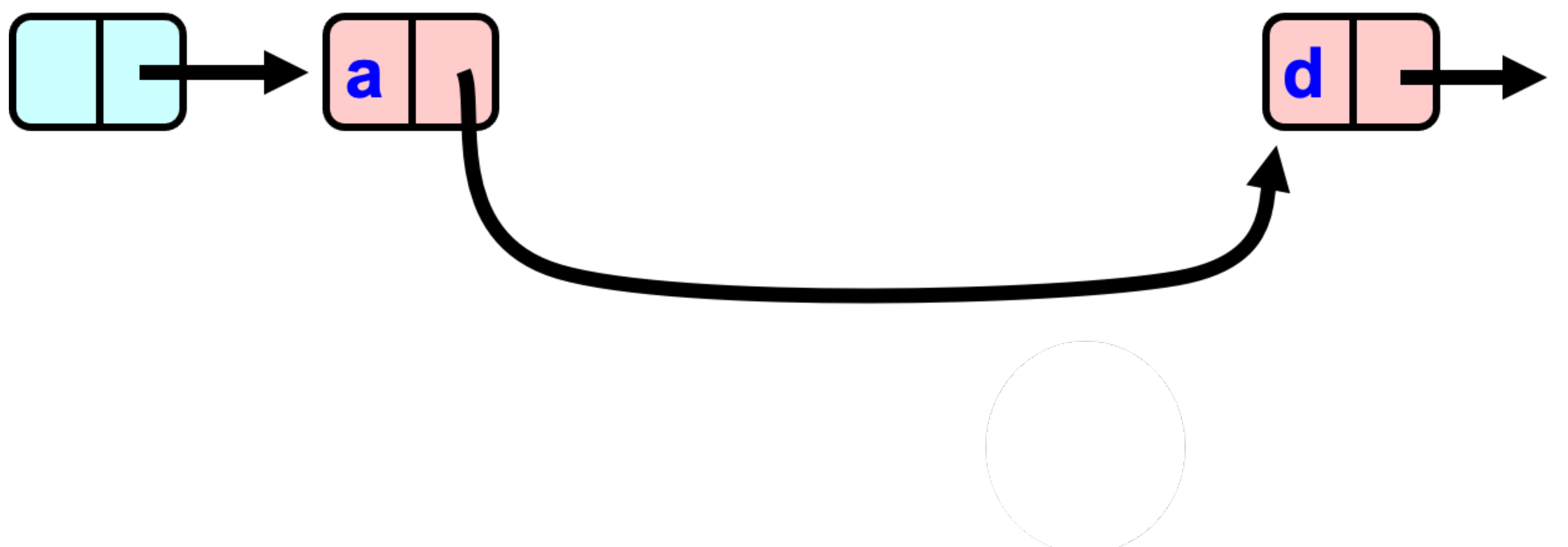
Fine Grained Locking



Fine Grained Locking



Fine Grained Locking



Fine Grained Locking

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Key used to order nodes

Fine Grained Locking

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Precursor and current node

Fine Grained Locking

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Make sure locks are freed

Fine Grained Locking

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Everything else

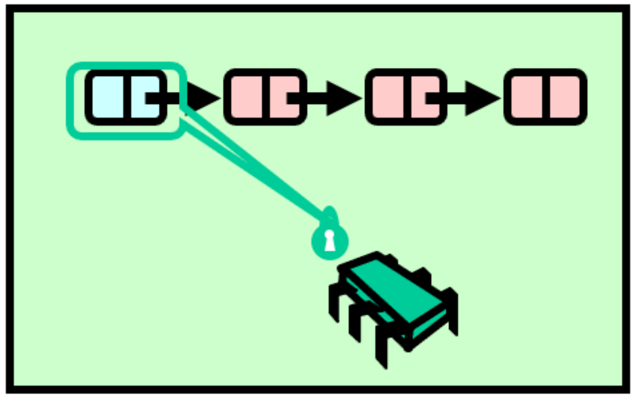
Fine Grained Locking

- Remove

```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

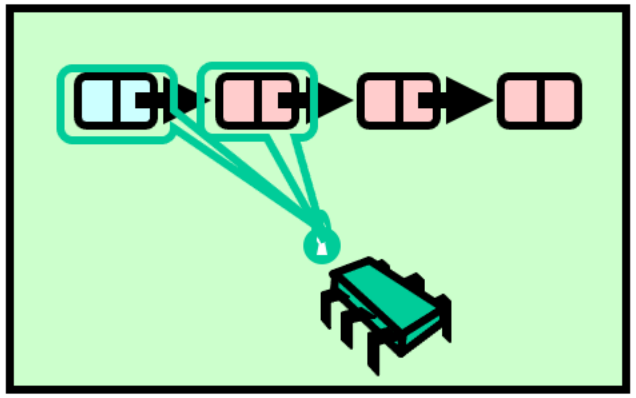
Fine Grained Locking

```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Fine Grained Locking

```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Searching

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Loop Invariant: At start of while, pred and curr are locked

Fine Grained Locking

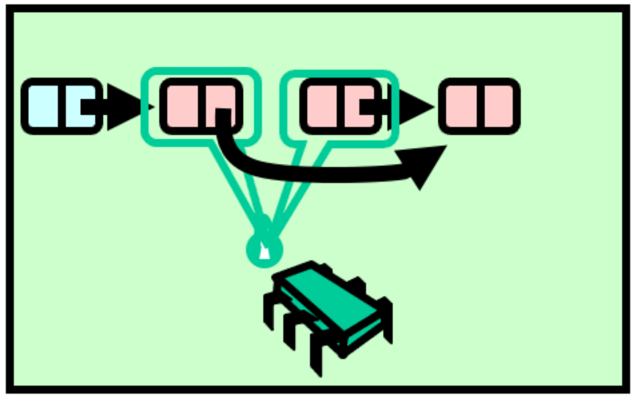
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Loop Invariant:
At start of while,
pred and curr are
locked

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

If item found, delete node



Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Unlock predecessor

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Move right

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Acquire next node

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Acquire next node

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Lock next node

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Loop invariant restored

Fine Grained Locking

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, return false

Fine Grained Locking

- Execution history is **Linearizable**:
  - Equivalent to a sequential history
- To argue something is linearizable:
  - Can find "linearization points"

Fine Grained Locking

- Invariants:
  - All items in the set are in nodes reachable from head
  - All nodes are arranged in order
- We show that invariants are maintained by methods

Fine Grained Locking

- Why remove is linearizable
  - Case 1: Item is in the list

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Fine Grained Locking

- Why remove is linearizable
  - Case 1: Item is in the list
    - Then pred.next = curr.next is a linearization point
- Invariants:
  - pred is reachable from head
  - curr is pred.next
  - curr is in the set
- No other thread can access either pred or curr during assignment

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
return false;
```

Fine Grained Locking

- Why remove is linearizable
  - After removal:
    - curr is no longer reachable: item is removed
    - pred is reachable from head
    - old curr.next is reachable
    - for all other nodes, reachability has not changed

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
return false;
```

Fine Grained Locking

- Why remove is linearizable
  - Case 2: Item is not in the list
    -

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Fine Grained Locking

- Why remove is linearizable
  - Case 2: Item is not in the list
    - return false is linearization point

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

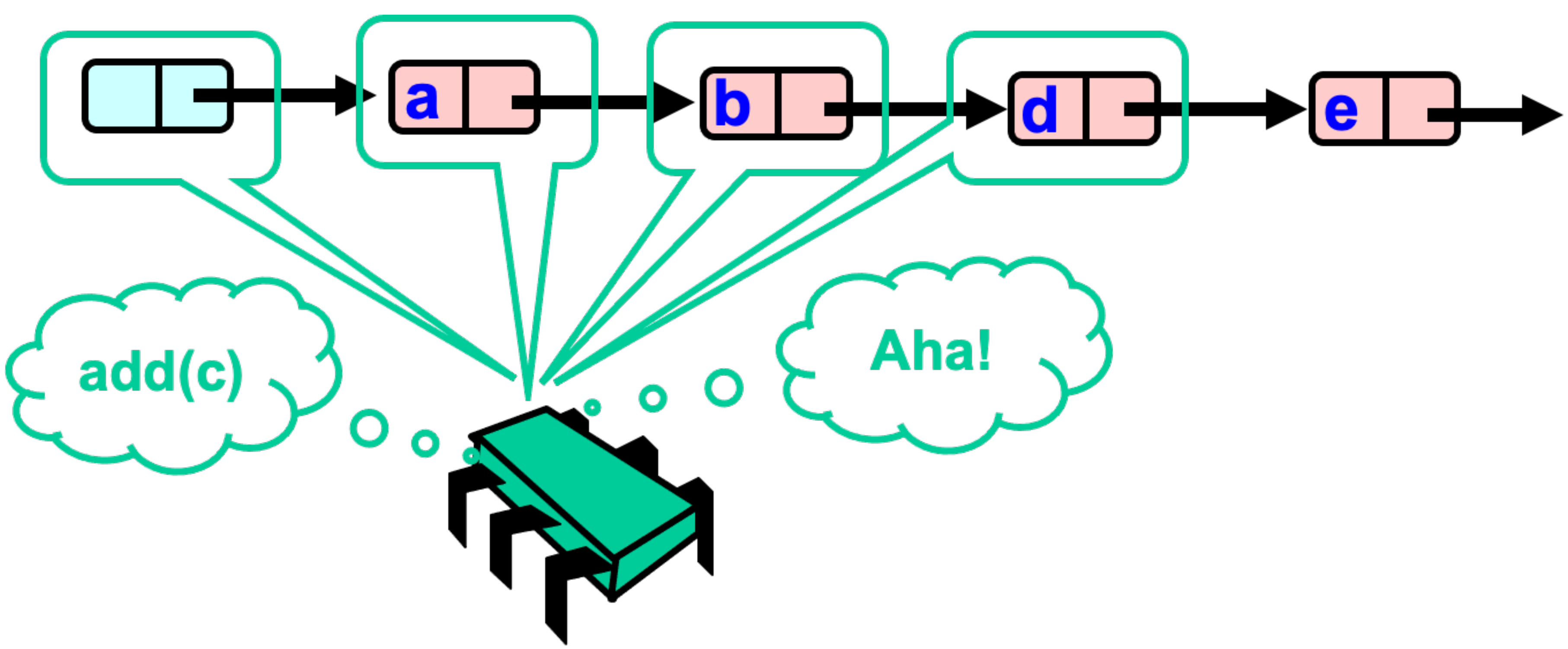
Fine Grained Locking

- Why remove is linearizable
  - Invariants are not changed
  - Need to show correctness:
    - Use induction to argue that item is not in the set

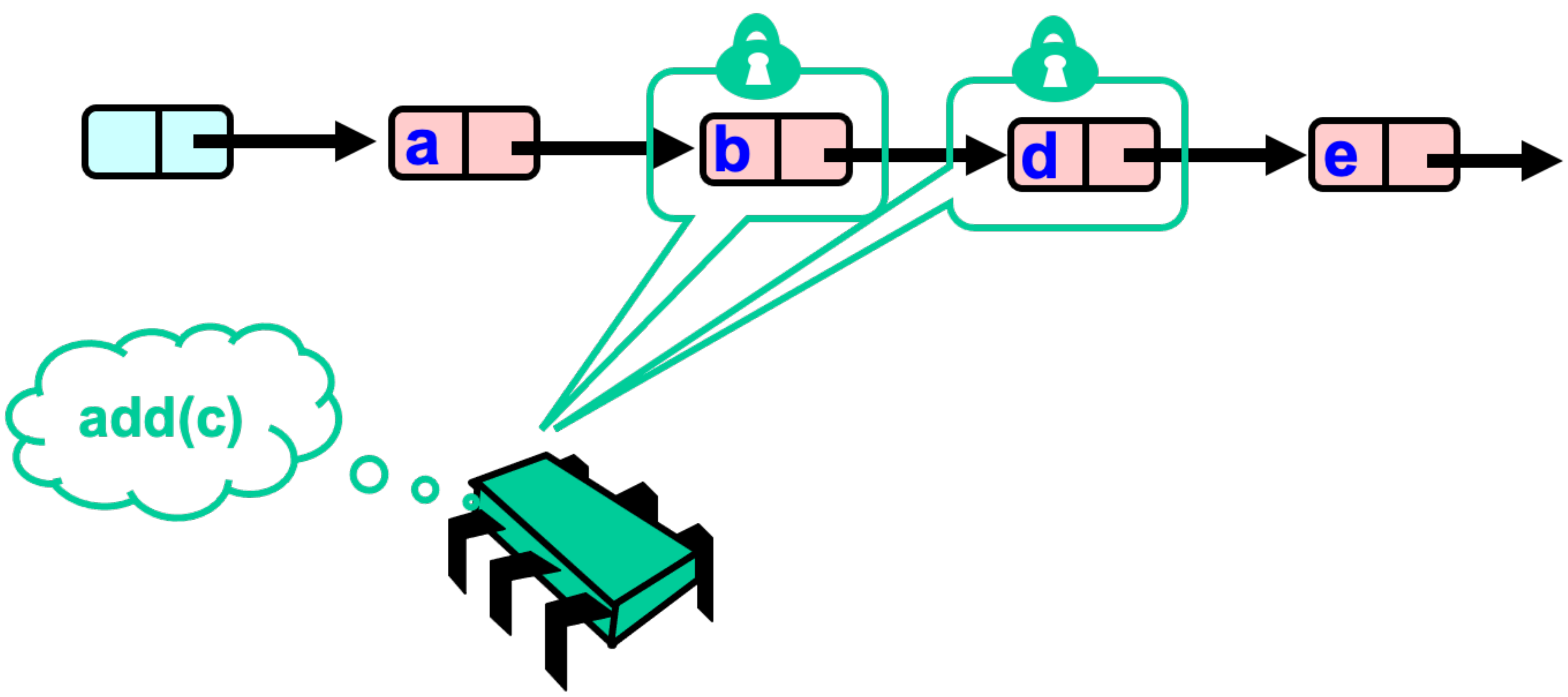
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Optimistic Locking

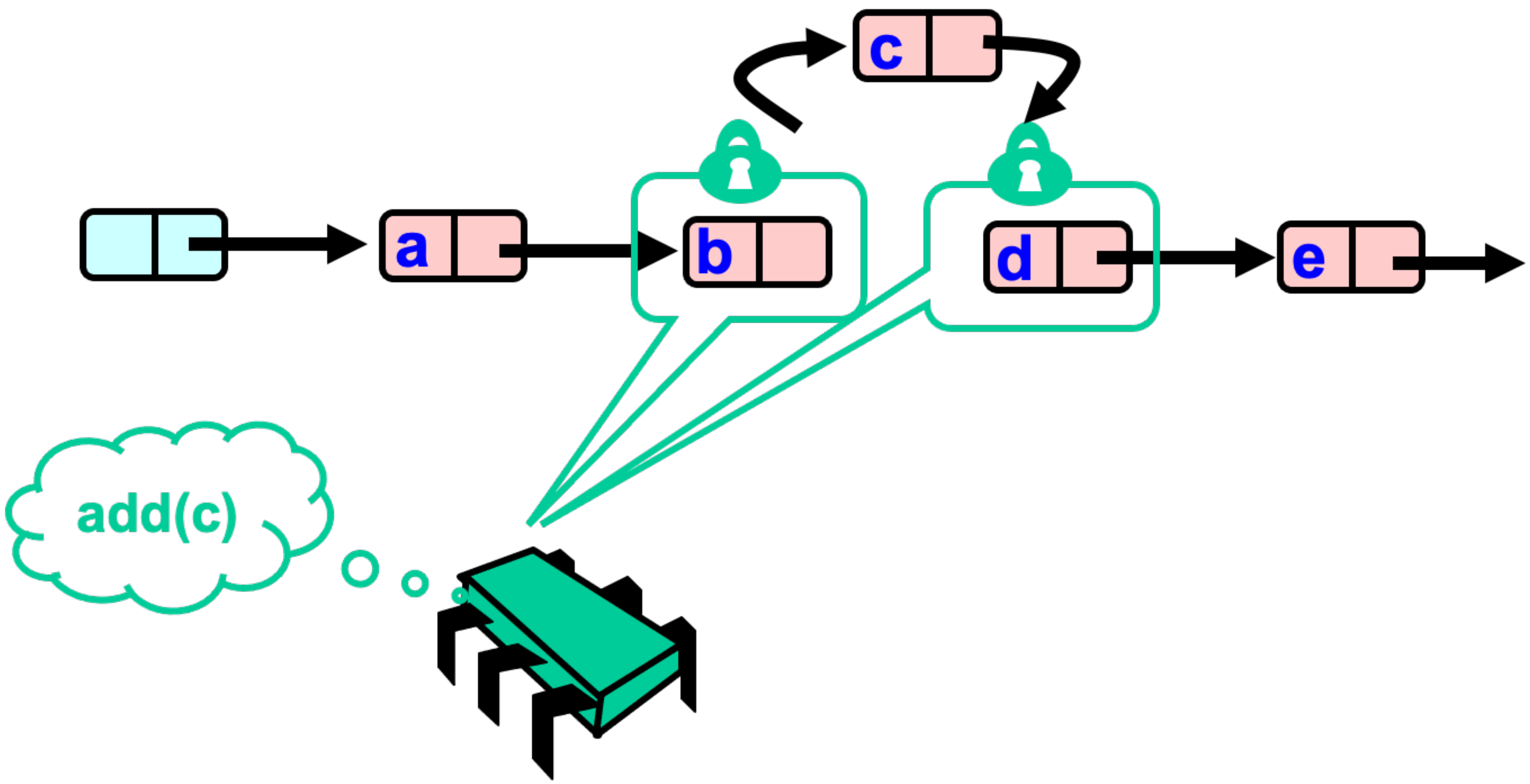
- Only lock when you are ready
  - Traverse list to find insertion / removal point
  - Then lock needed nodes after validation!



Optimistic Locking

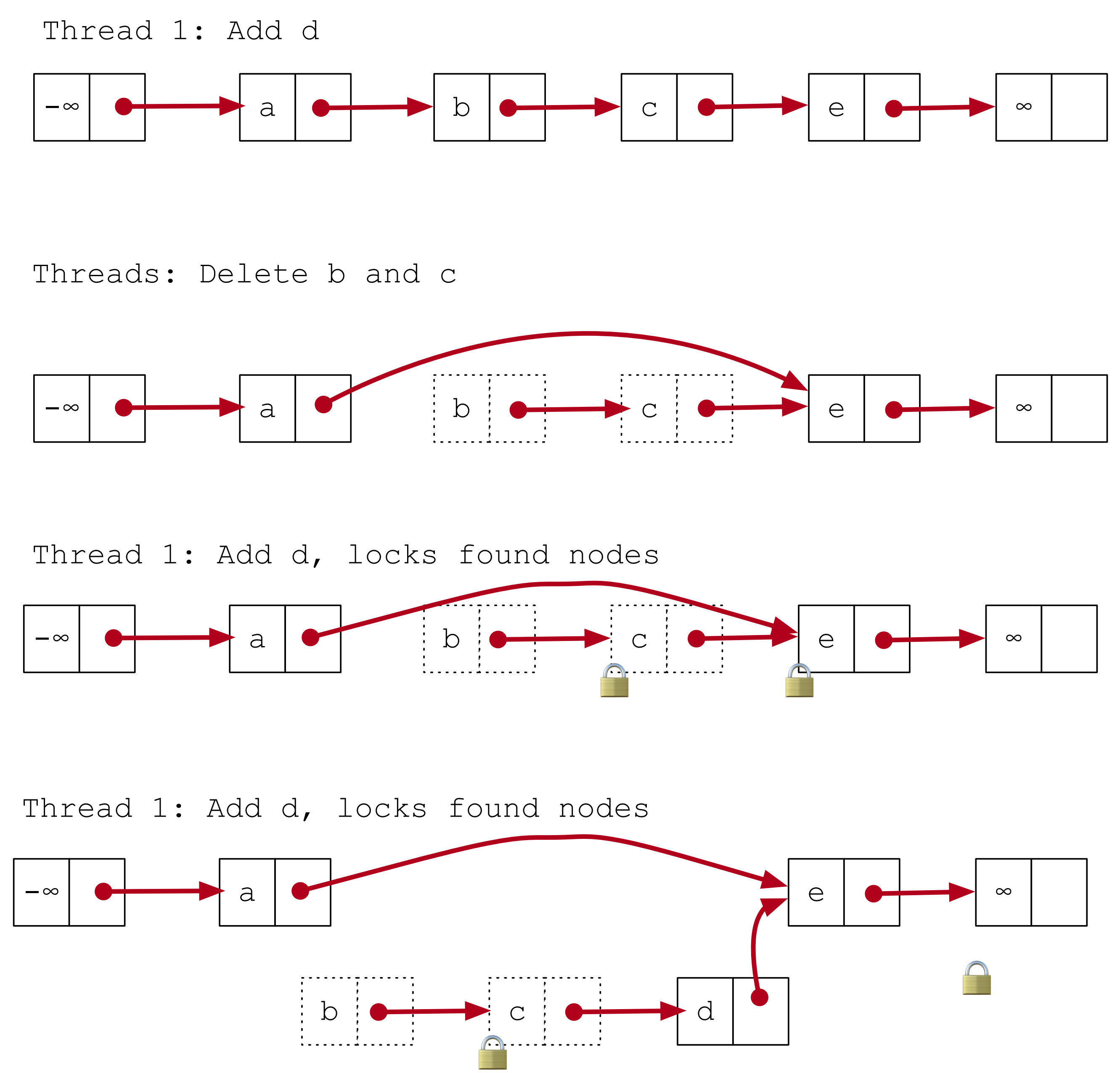


Optimistic Locking



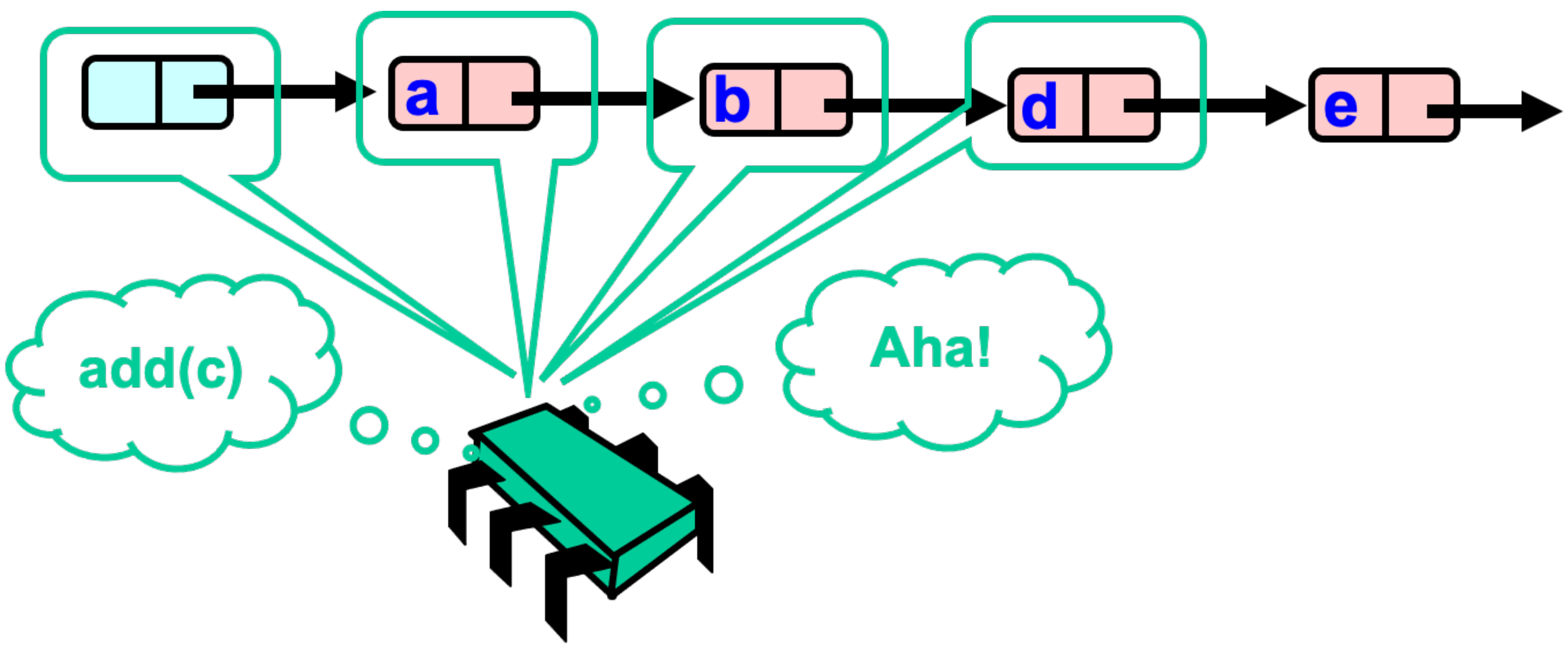
Optimistic Locking

- Why we need validation

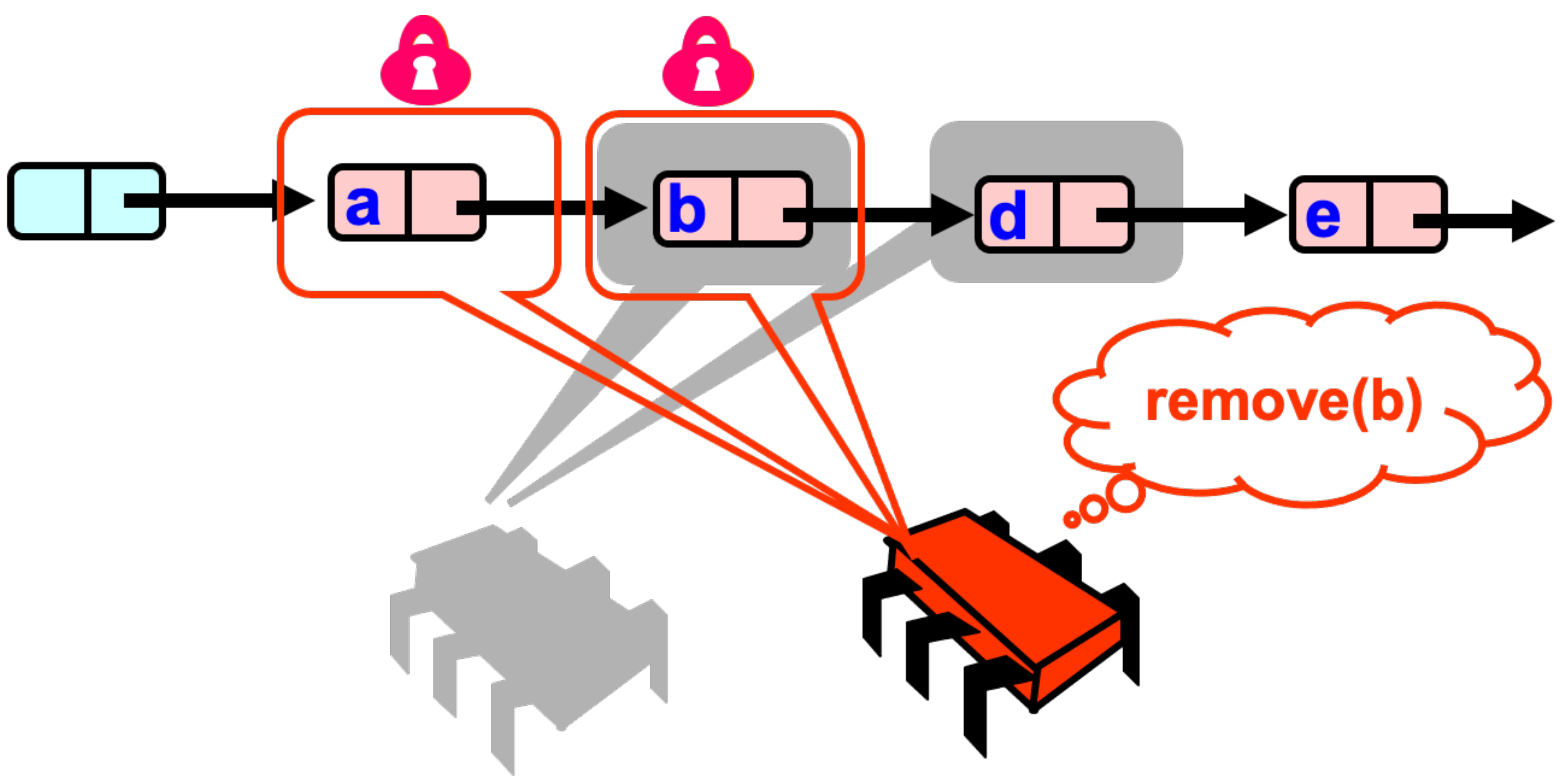


Optimistic Locking

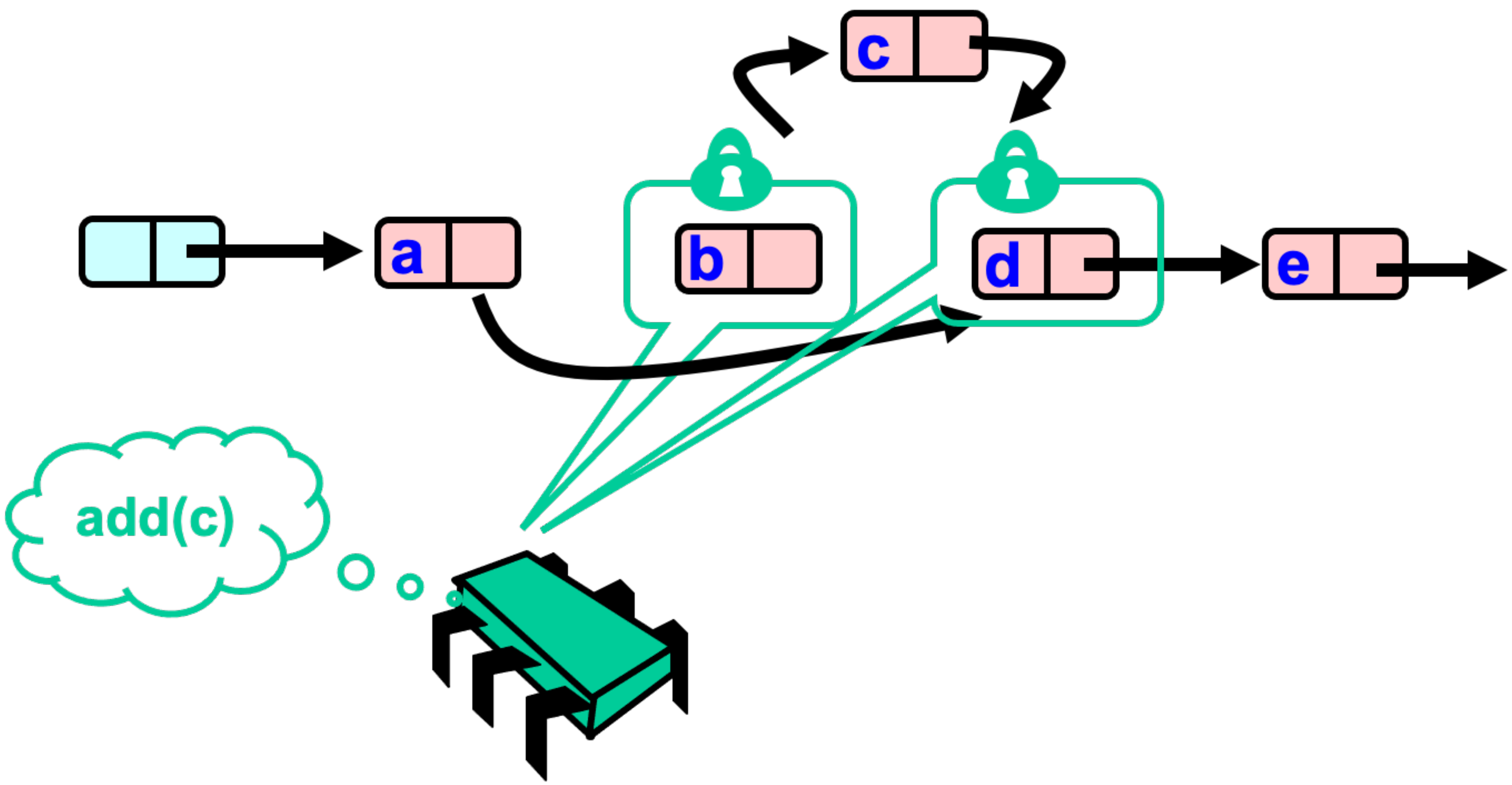
- What can go wrong?
  - Nodes might no longer be there



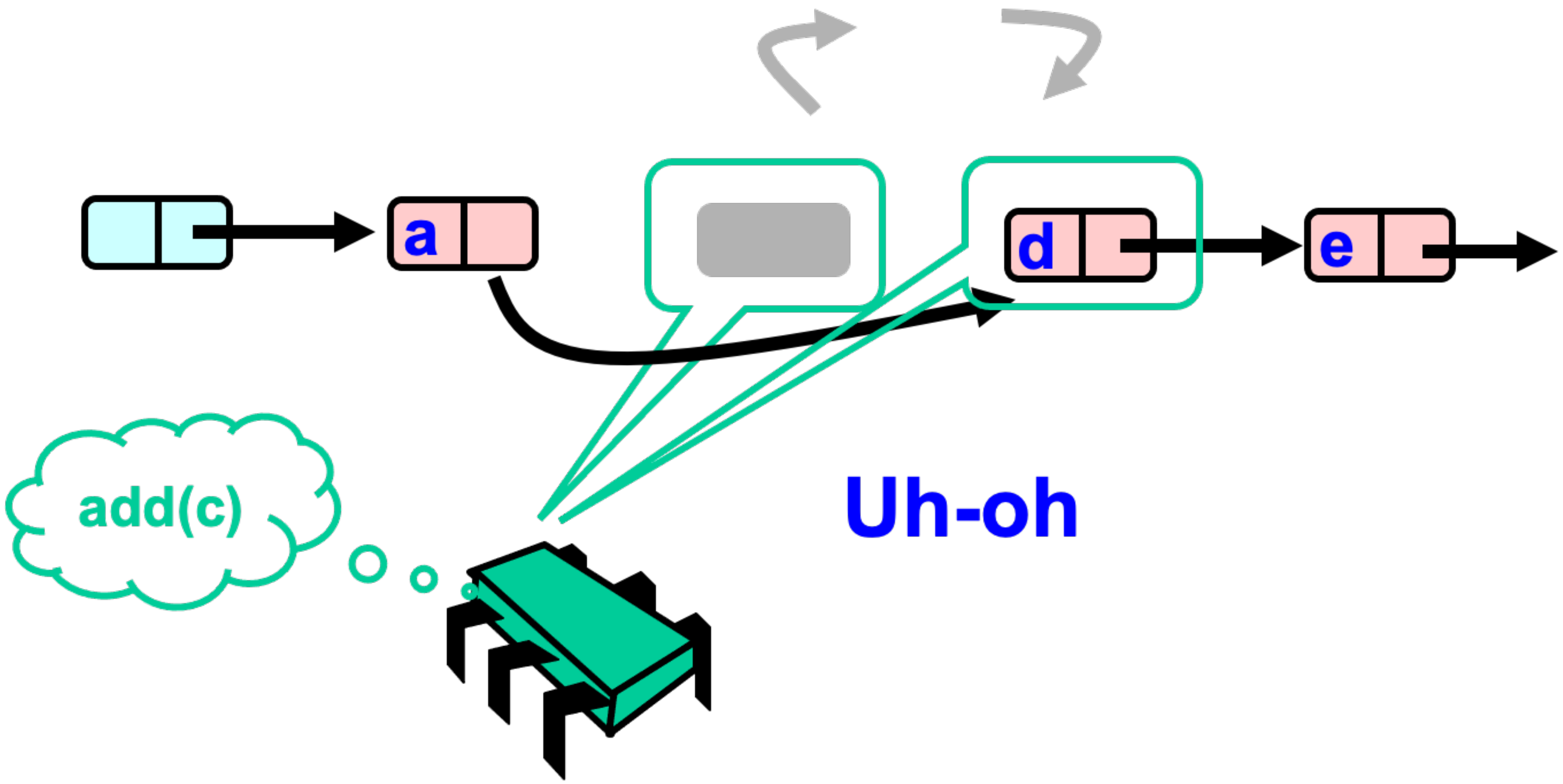
Optimistic Locking



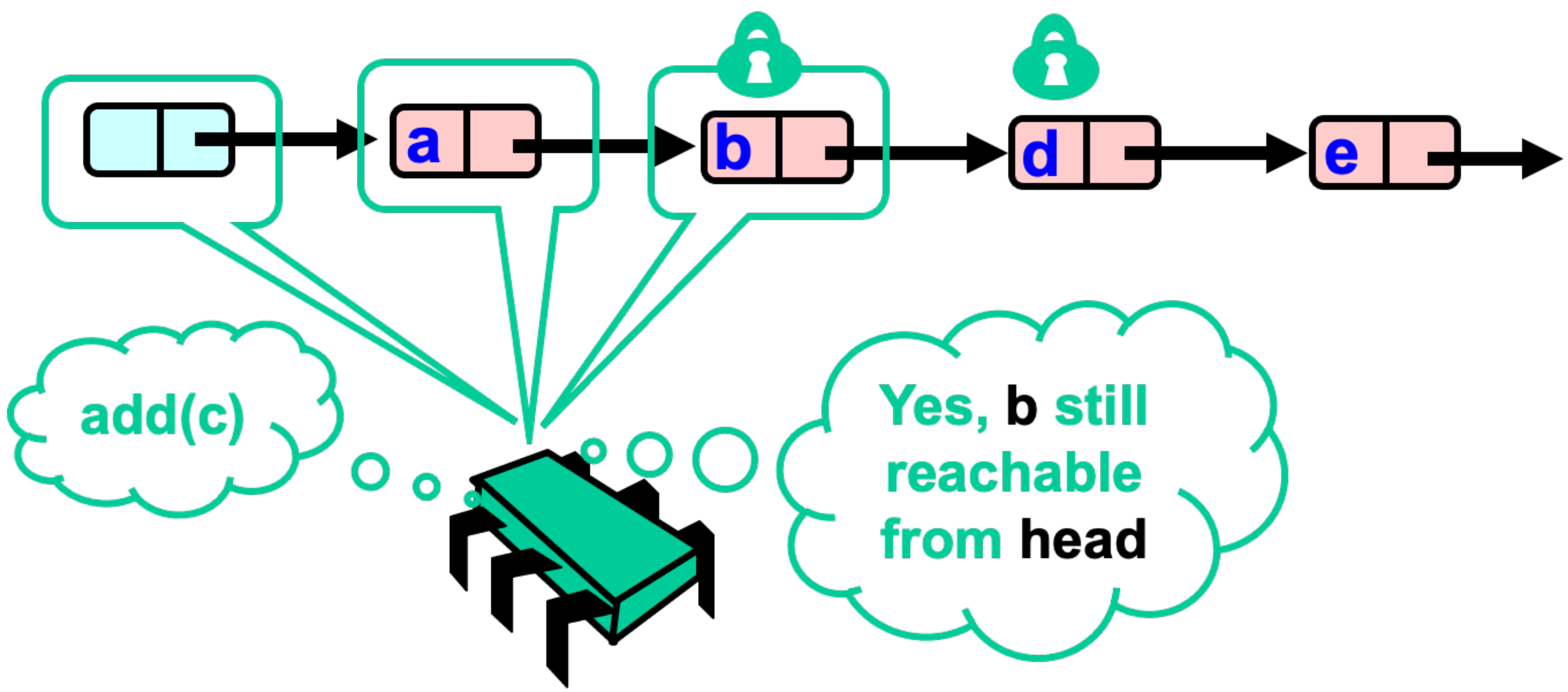
Optimistic Locking



Optimistic Locking



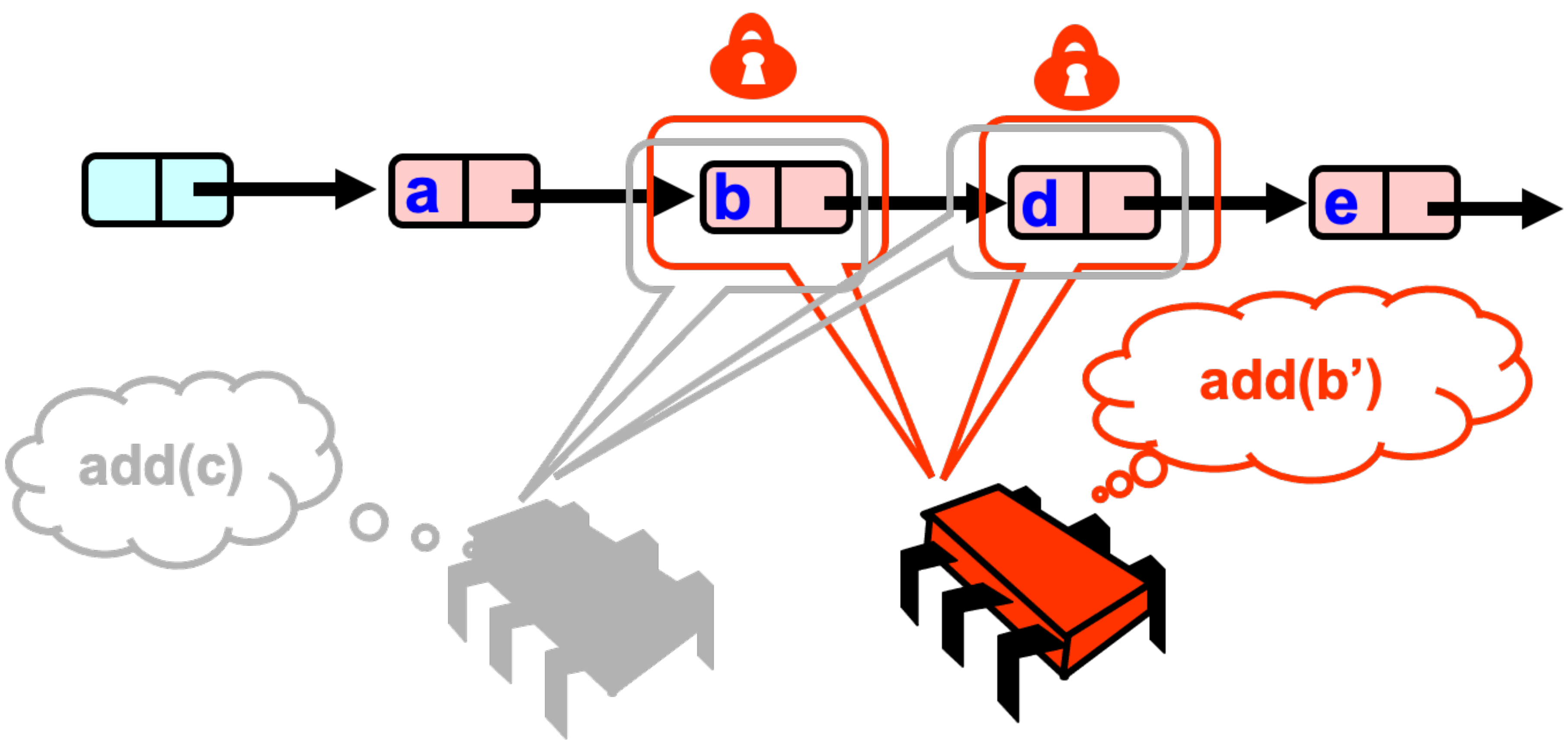
Optimistic Locking



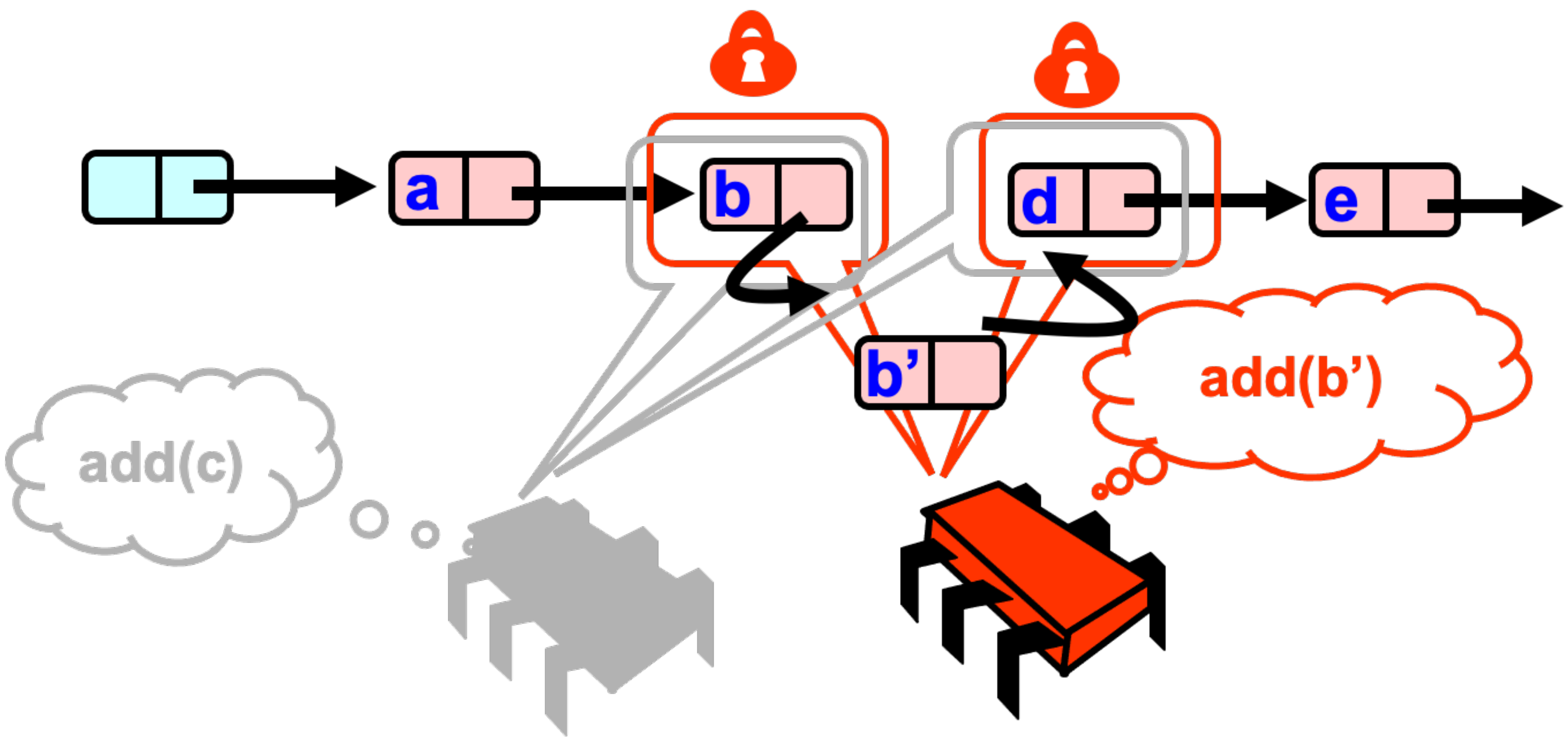
Need to validate

Optimistic Locking

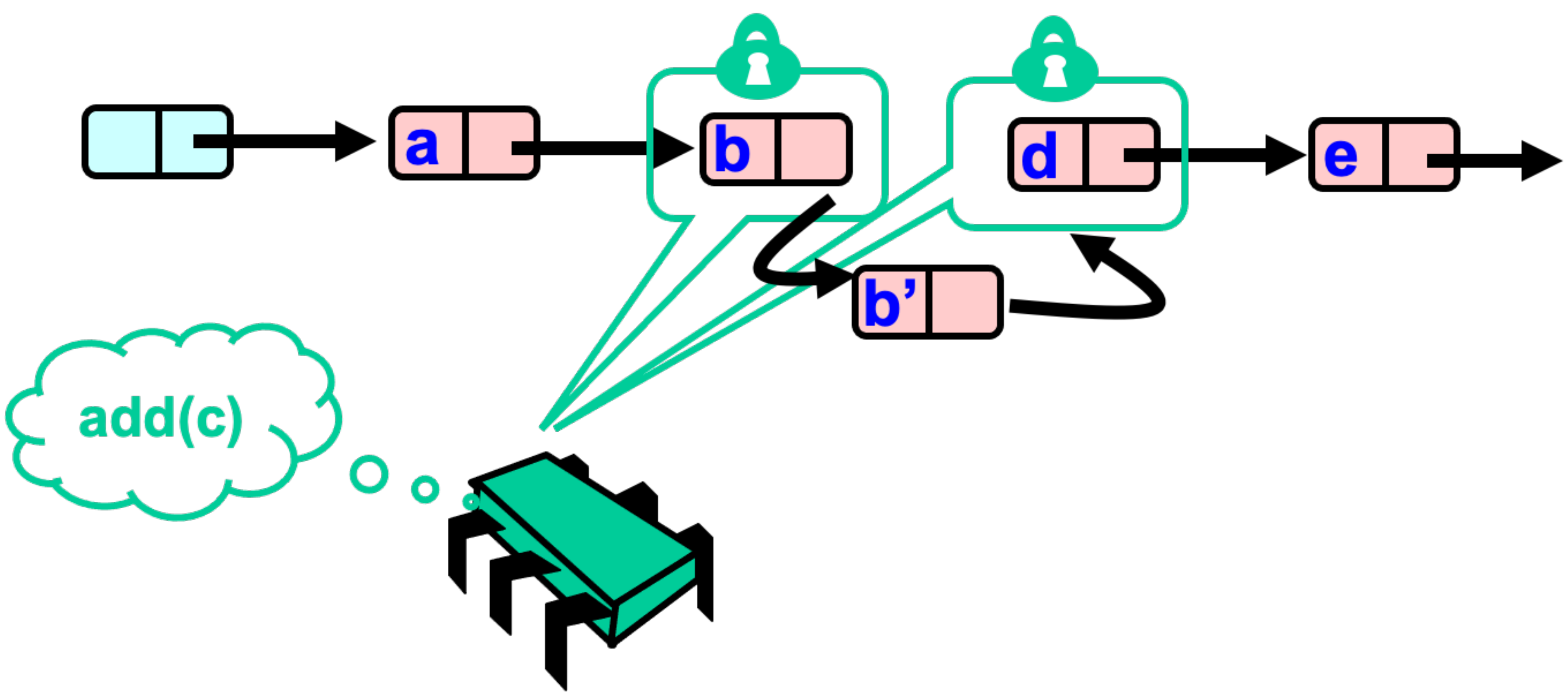
- What else can go wrong?



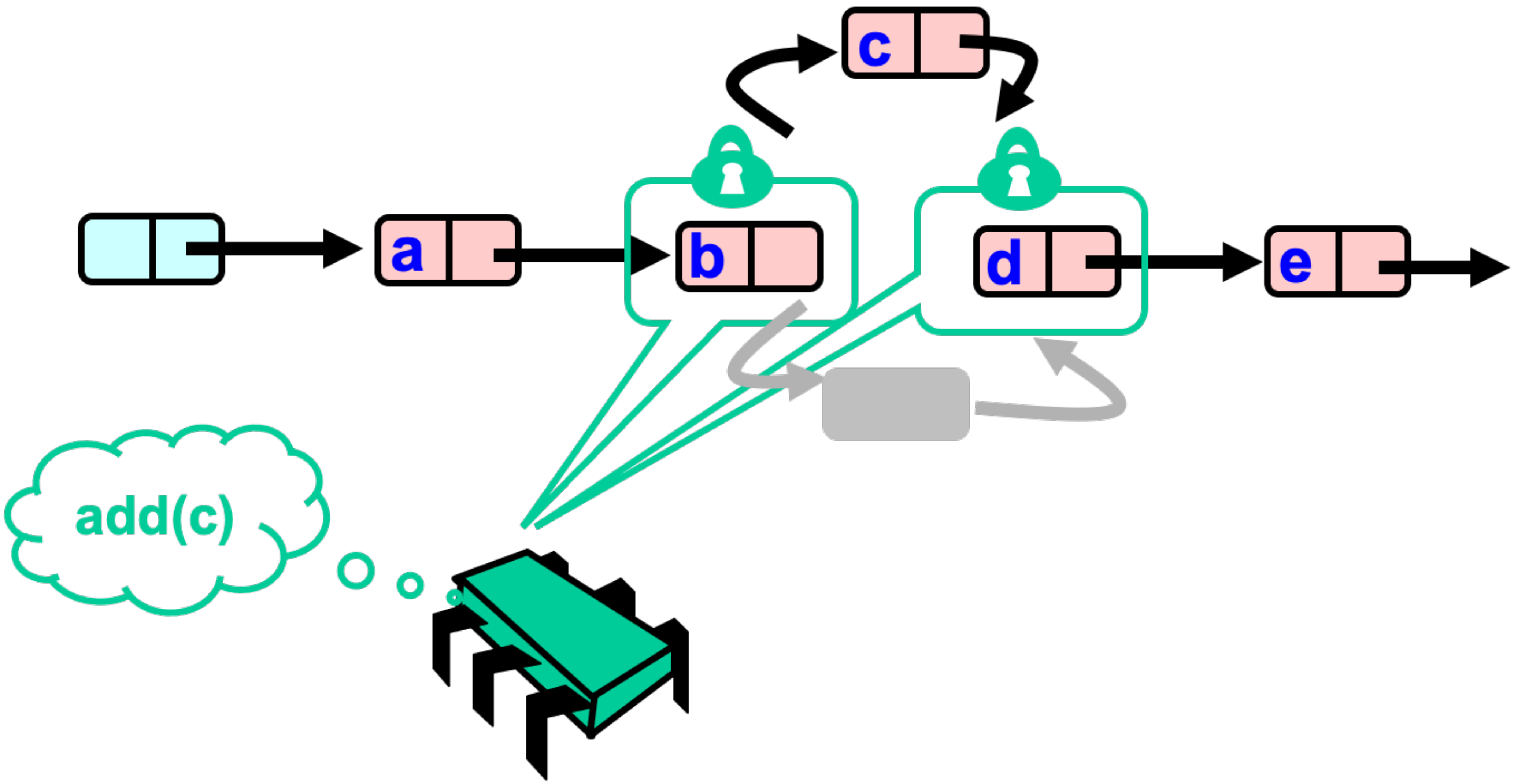
Optimistic Locking



Optimistic Locking

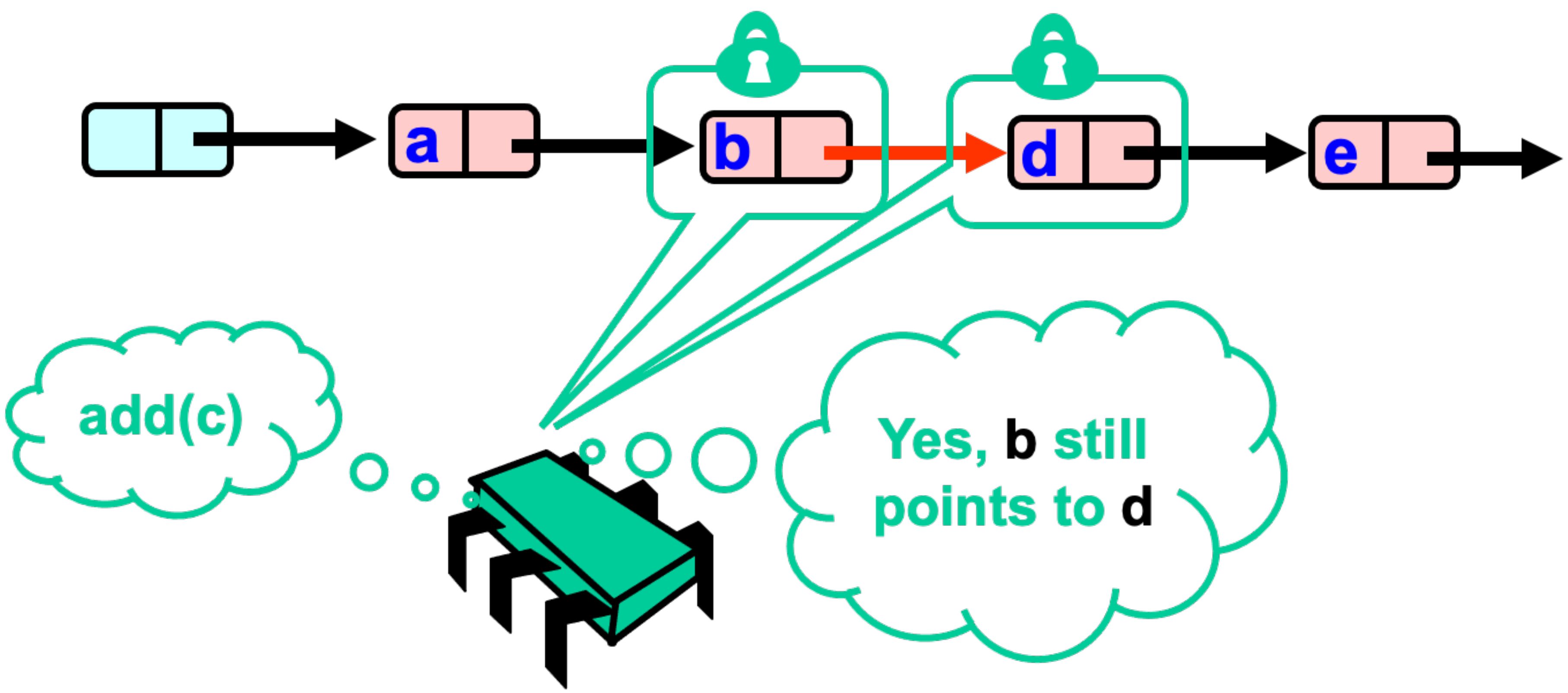


Optimistic Locking



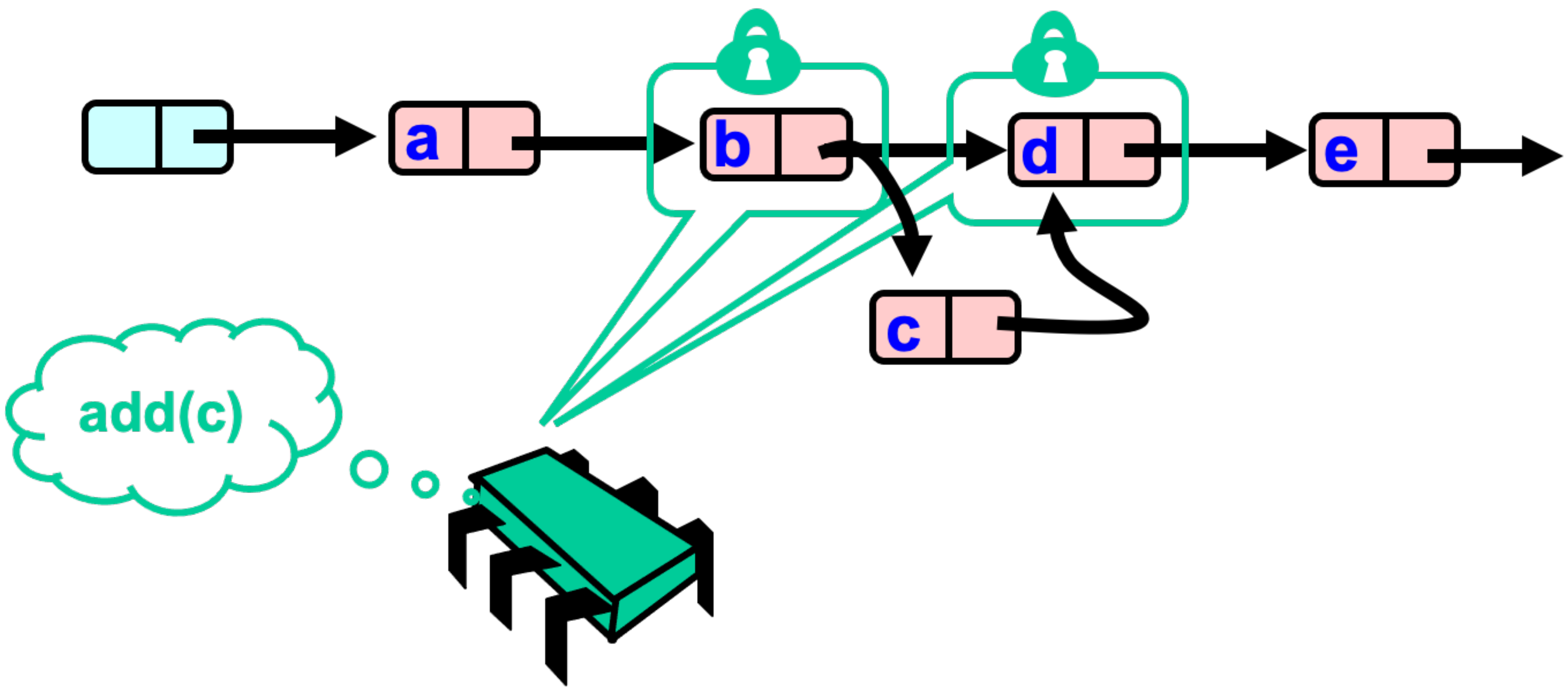
Optimistic Locking

- Need to validate **while holding locks**



Optimistic Locking

- Need to validate **while holding locks**
- Linearization point



Optimistic Locking

- Optimistic locking:
  - Search without acquiring locks
  - Lock the nodes found
  - Confirm that locked nodes are correct
    - For inserting a node between Node A and Node B:
      - Node A is reachable from head
      - Node B is still the successor of Node A

Optimistic Locking

- Validation:
  - Reachability of Node A
    - No operation changes reachability with exception of the Node being removed
      - Verify that!
    - Therefore: we do not need locks to verify reachability

Optimistic Locking

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

Optimistic Locking

- Addition: Phase 1: searching

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr; curr = curr.next;
```

Optimistic Locking

- Addition: Phase 2: Locking

```
pred.lock();
curr.lock();
```

Optimistic Locking

- Addition: Phase 3: Validation and Update

```
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 {
     pred.unlock();
     curr.unlock();
}
```

Optimistic Locking

- Remove

```
public boolean remove(T item) {
   int key = item.hashCode();
    while( true ){
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key)  {
            pred = curr;
            curr = curr.next;
        }
```

Optimistic Locking

- Remove: Lock phase

```
pred.lock();
curr.lock();
```

Optimistic Locking

- Remove: Validation and deletion phase

```
try {
   if (validate(pred, curr)) {
      if (curr.key == key) {
         pred.next = curr.next;
         return true;
      } else {
         return false;
   }
}
} finally {
           pred.unlock(); curr.unlock();
         }
}}
```

Optimistic Locking

- On exit from loop **and in the absence of synchronization problems:**
  - If item is present:
    - curr holds item
    - pred just before curr
  - If item is absent:
    - curr has higher key
    - pred just before curr

```
public boolean remove(T item) {
    int key = item.hashCode();
     while( true ){
         Node pred = head;
         Node curr = pred.next;
         while (curr.key < key)  {
             pred = curr;
             curr = curr.next;
         }
```

Optimistic Locking

- Remove: Validation and deletion phase

```
try {
    if (validate(pred, curr)) {
        if (curr.key == key) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
    }
}
} finally {
            pred.unlock(); curr.unlock();
        }
}}
```

Check for synchronization problems

Optimistic Locking

- Limited hot-spots:
  - Targets of add, remove, contains
- No contention on traversals
- Traversals are wait-free

Lazy Locking

- Optimistic locking:
  - Traverses list twice
  - Contains locks
- Lazy locking:
  - Make validation simpler
    - By marking deleted nodes

Lazy Locking

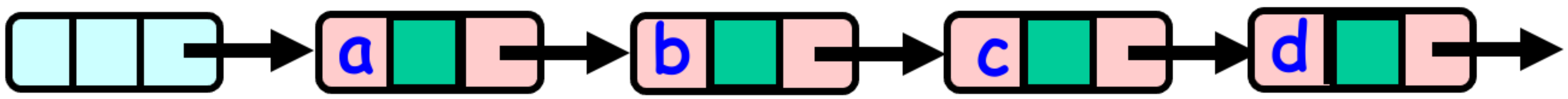
- Add to each node a Boolean `marked` field
- Traversals no longer need to validate that a node is reachable:
  - New invariant:
    - Every unmarked node is reachable

Lazy Locking

- Contains:
  - Just traverse the list, including nodes marked deleted
  - If the item is in the list and the node is not marked deleted, then it is in the set

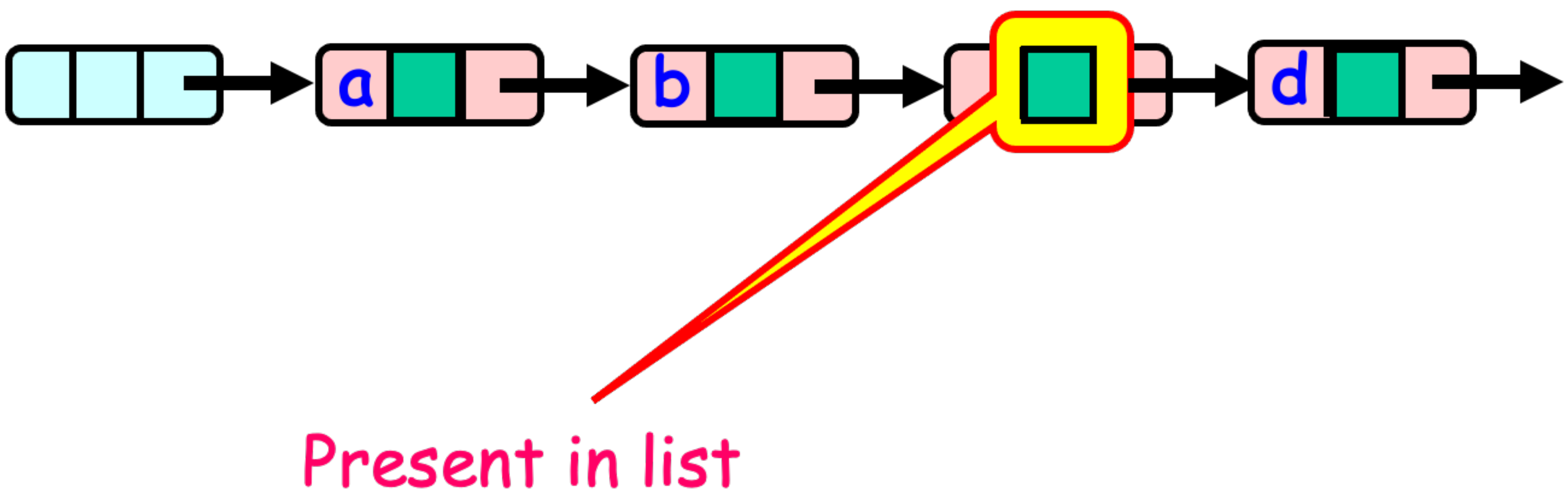
Lazy Locking

- Lazy removal



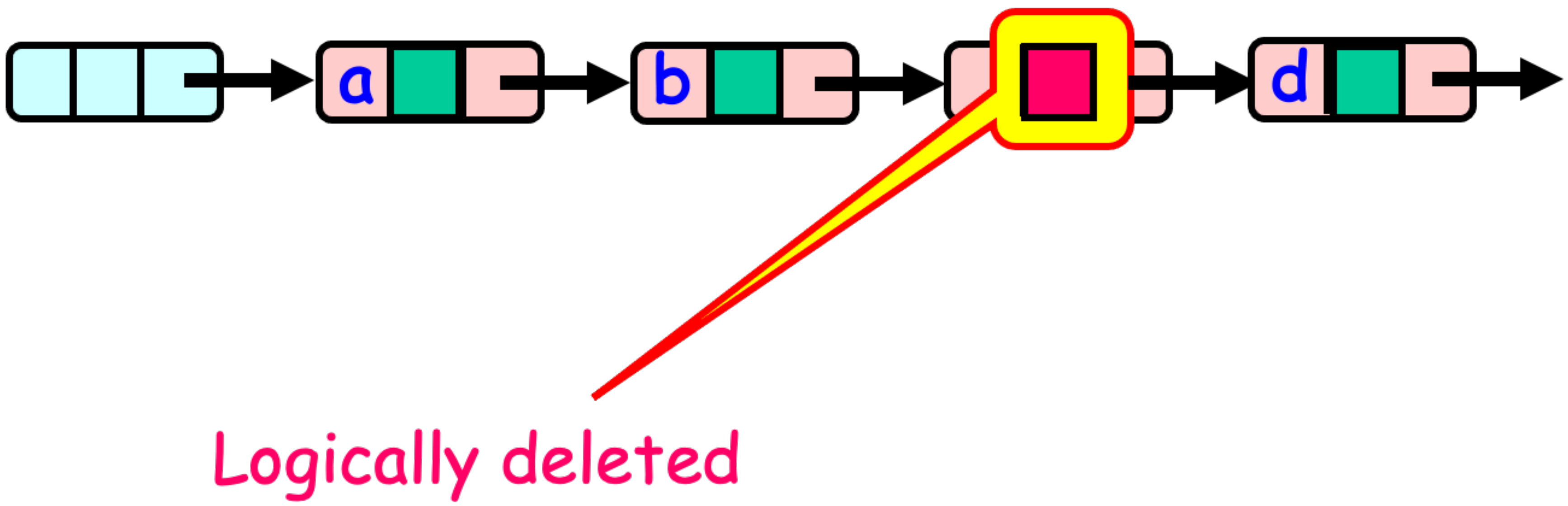
Lazy Locking

- Lazy removal

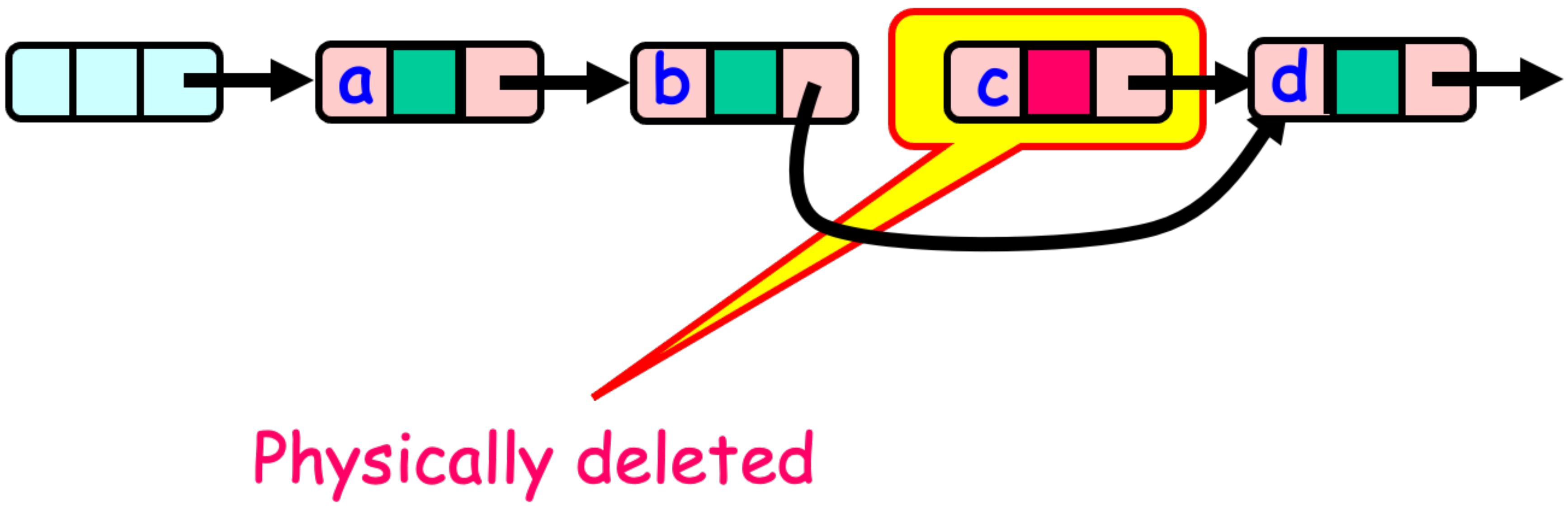


Lazy Locking

- Lazy removal

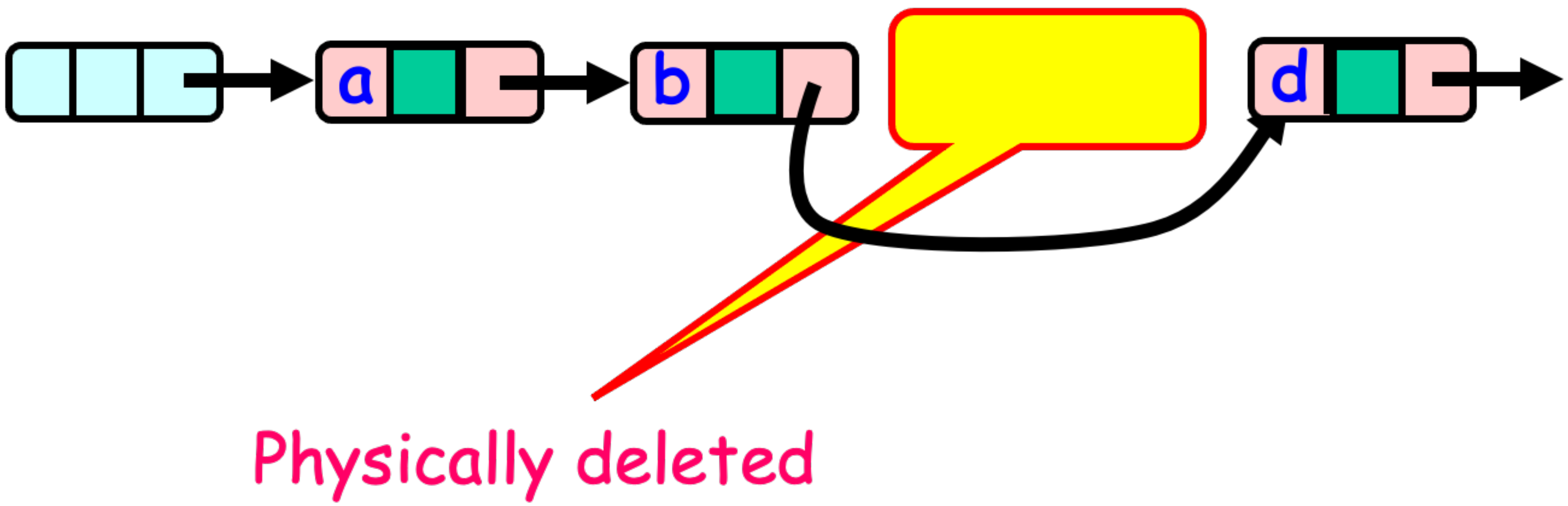


Lazy Locking



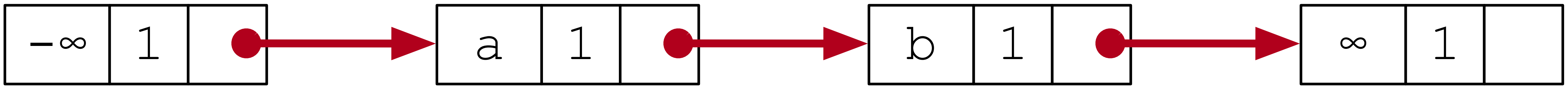
Lazy Locking

- Lazy removal



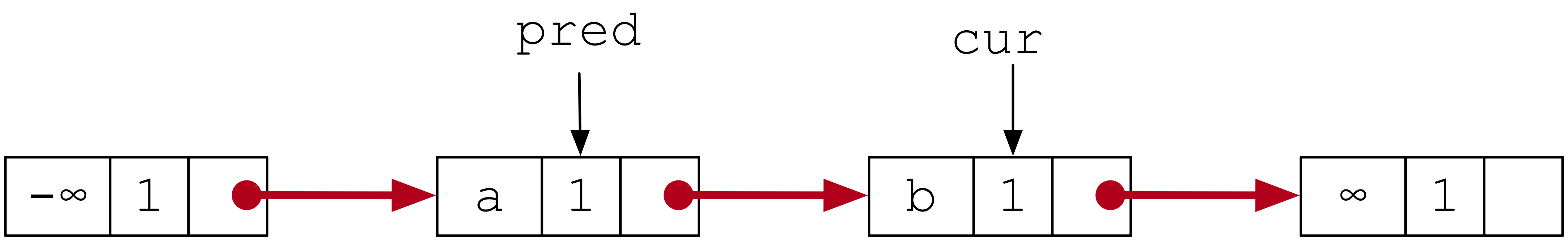
Lazy Locking

- Why do we need to validate?
  - Thread I removes b



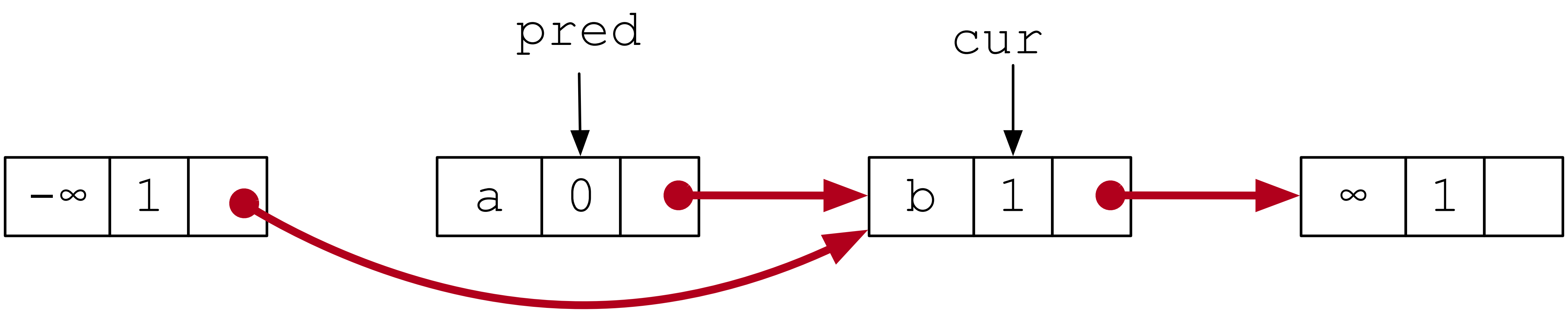
Lazy Locking

- Thread 1 finds b



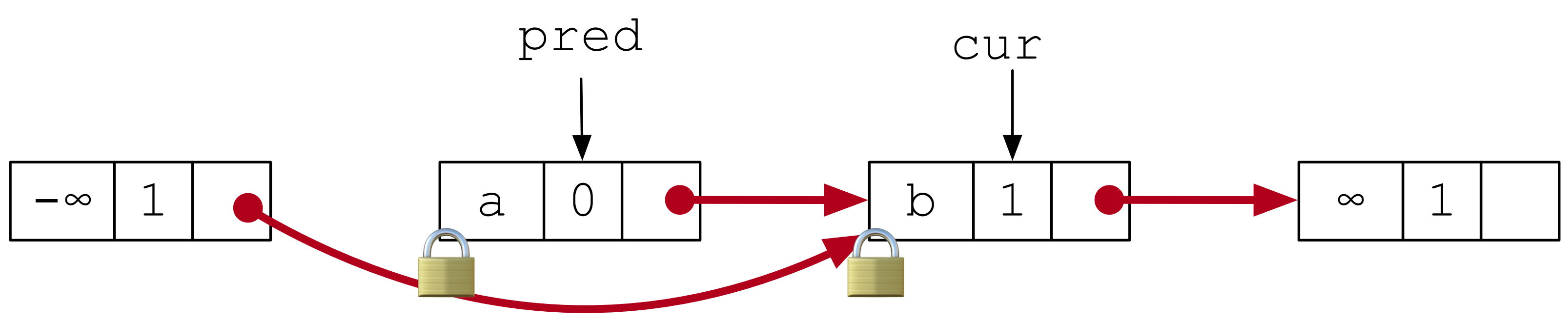
Lazy Locking

- Before Thread 1 acquires the lock, another thread logically and physically removes the predecessor



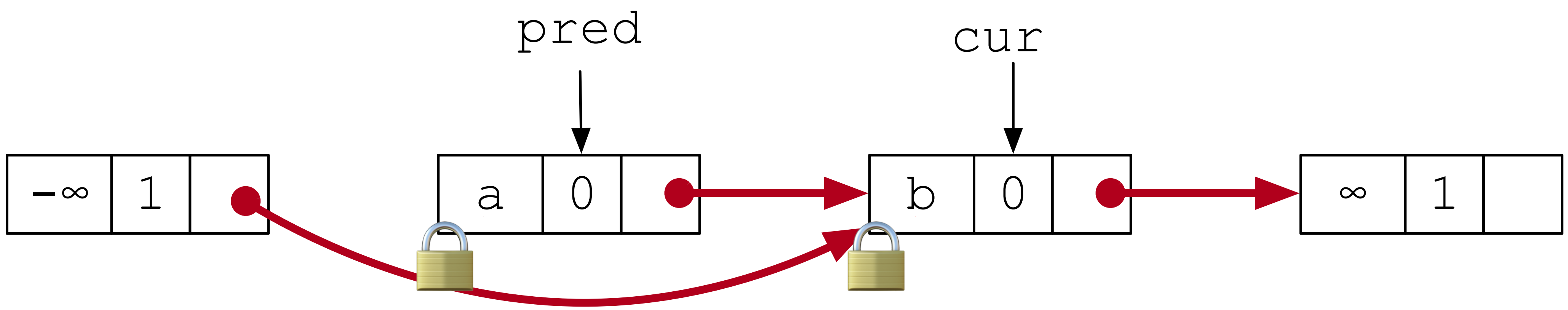
Lazy Locking

- Thread 1 now acquires the lock



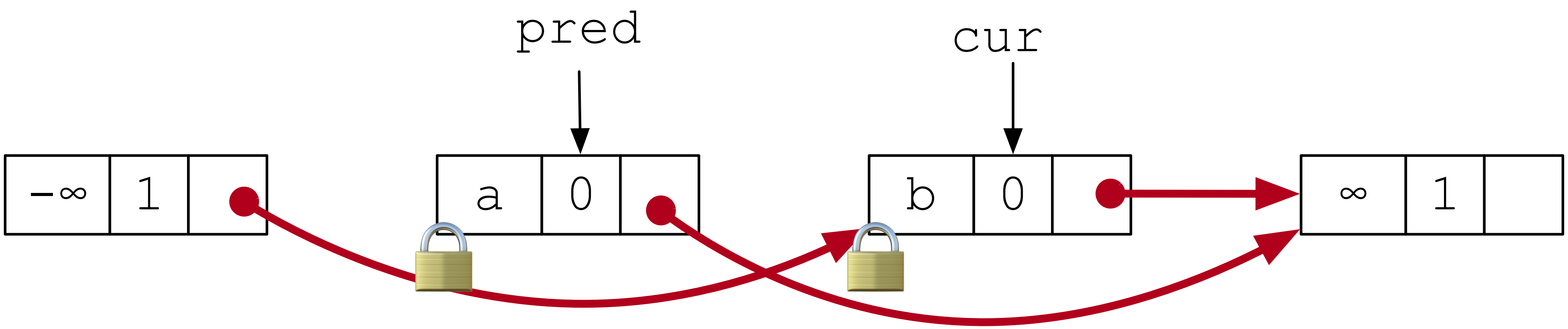
Lazy Locking

- Thread I marks b as deleted



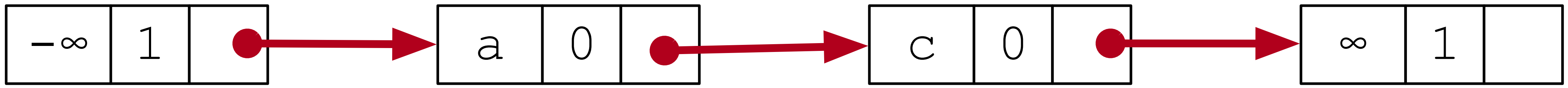
Lazy Locking

- And then removes it physically



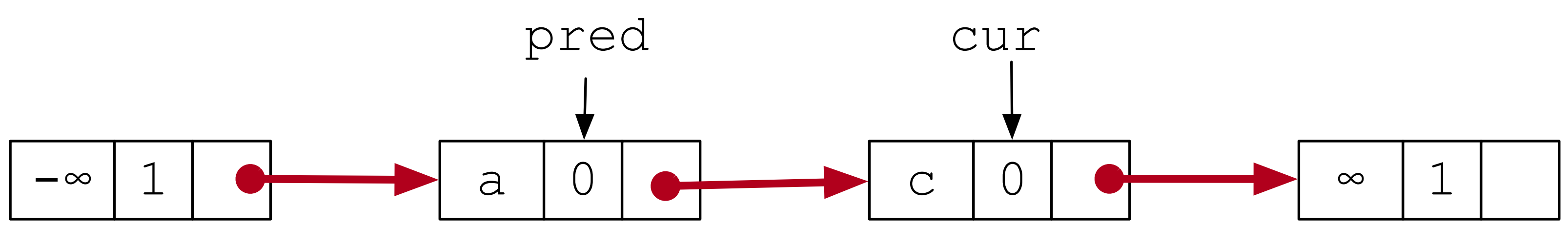
Lazy Locking

- Another scenario:
    - Thread I tries to remove c



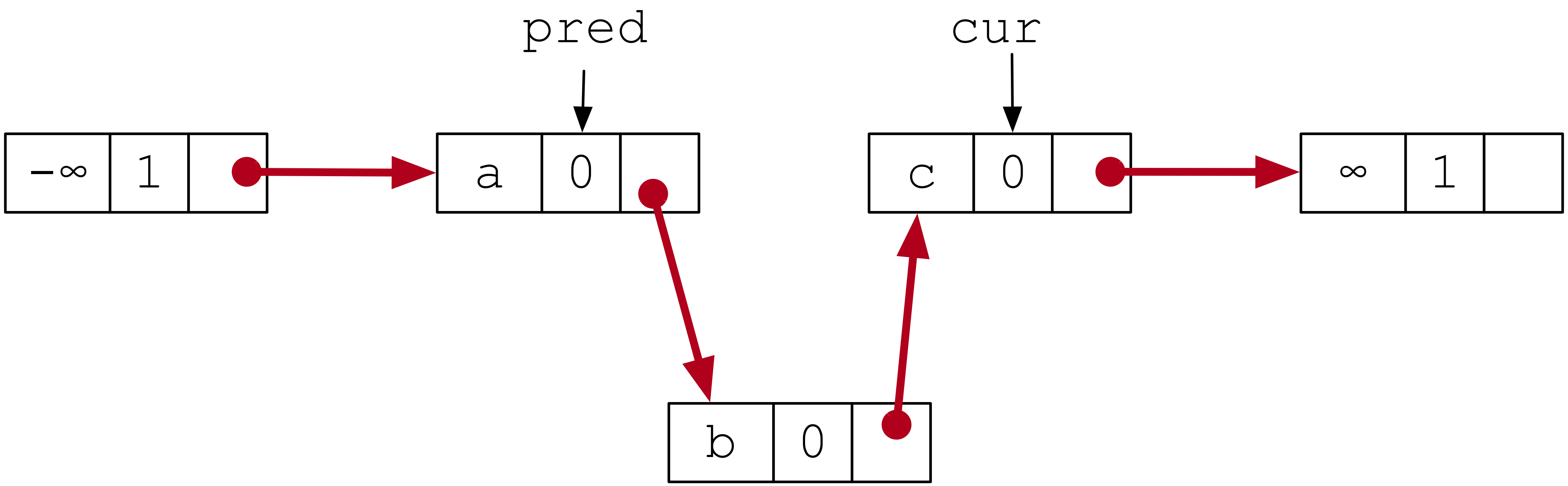
Lazy Locking

- Thread I finds them



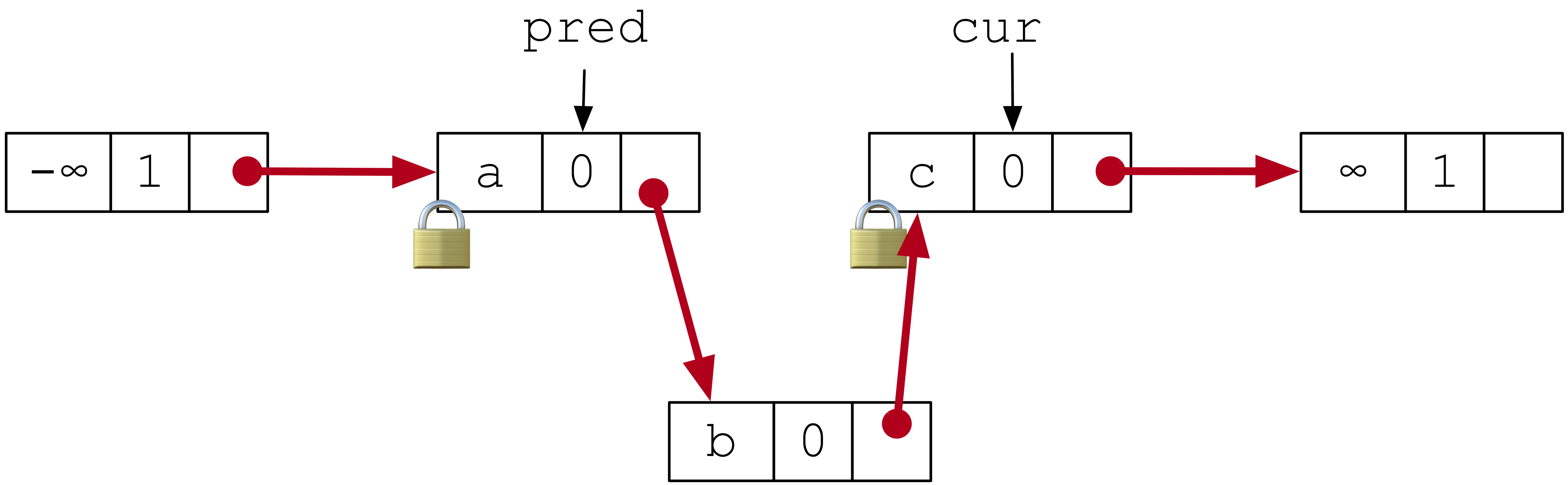
Lazy Locking

- But before locking, another thread adds a node b



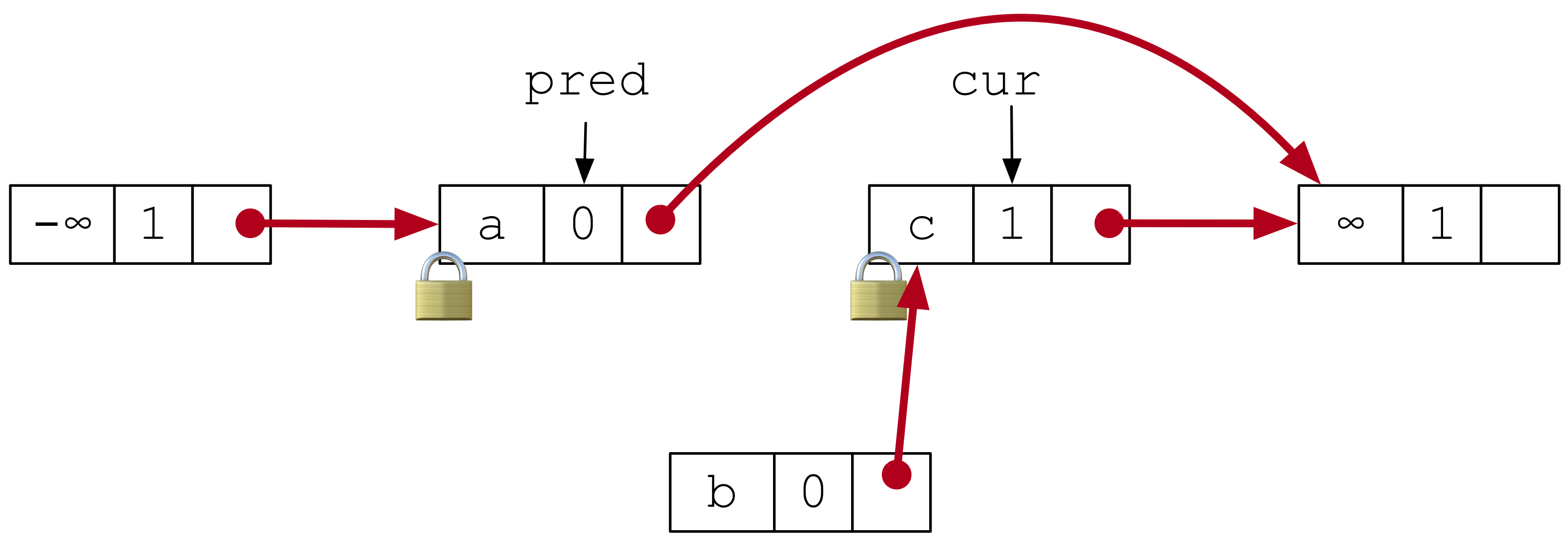
Lazy Locking

- Thread I now locks



Lazy Locking

- And virtually and physically removes node c



Lazy Locking

- Validation:
    - Check that pred is not marked
    - Check that curr is not marked
    - Check that pred.next == curr

Lazy Locking

- Validation

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

Lazy Locking

- Validation

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

predecessor not logically deleted

Lazy Locking

- Validation

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

current node not logically deleted

Lazy Locking

- Validation

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

predecessor still predecessor

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

lock both nodes

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

validate

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

key found

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

logic delete

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

logic delete

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

physical delete

Lazy Locking

- Removal

```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
  pred.unlock();
  curr.unlock();
   }}}
```

done

Lazy Locking

- Containment

```
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;
}
```

Lazy Locking

- Containment

```
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 head

Lazy Locking

- Containment

```
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 list without locking

Nodes might be deleted

Lazy Locking

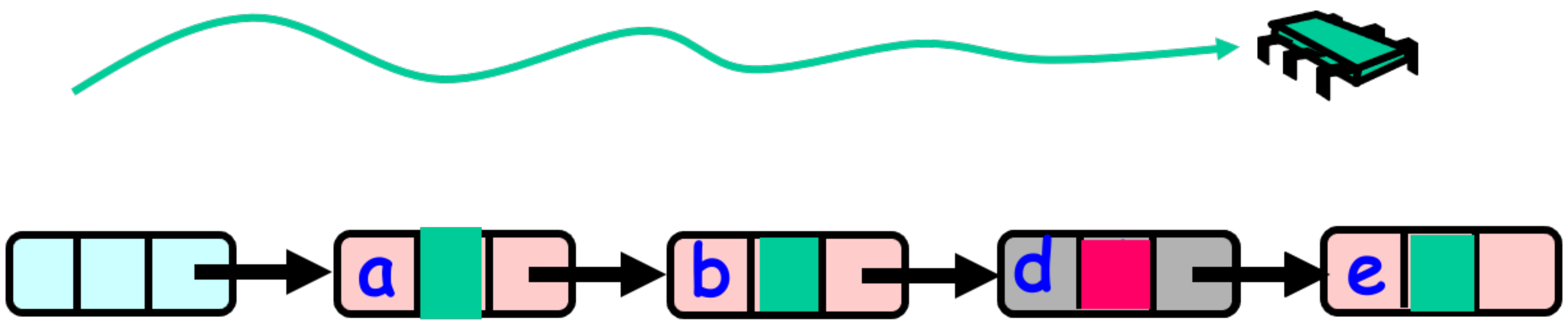
- Containment

```
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 Locking

- Summary



- Combine mark bit and list ordering

Lazy Locking

- Lazy adds and removes
- Wait-free contains

Lazy Locking

- Good:
  - Contains is wait-free
  - Uncontended calls do not re-traverse
- Bad:
  - Contended add / removes require re-traversion

CAS

- CAS instruction: Compare And Set
  - `Boolean register.CAS(expected, update)`
    - Atomic operation
    - If `register` value is equal to `expected` then its value becomes `update` and returns `true`
    - If `register` value is not equal to `expected`, returns `false`, but does not change the value

CAS

- Example: Consensus protocol for *n* threads 0, …, *n*-1
- `AtomicInteger` class has a CAS method

```
class CASConsensus extends ConsensusProtocol {
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST);
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get();
        if (r.compareAndSet(FIRST, i)) // I won
             return proposed[i];
        else // I lost
             return proposed[r.get()];
    }
}
```

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

Load r with First

CAS

```
class CASConsensus extends ConsensusProtocol {
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST);
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get();
        if (r.compareAndSet(FIRST, i)) // I won
            return proposed[i];
        else // I lost
            return proposed[r.get()];
    }
}
```

Each thread loads global array proposed with a value

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

Try whether there is still the original value in r

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
           return proposed[r.get()];
   }
}
```

If it is, exchange with thread-number

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

This happens for only one thread, who gets to update the value of r with its thread number

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

All other threads will find the value different

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

All other threads will find the value different:
The value is the number of the winning thread
Therefore, they return its proposed value

CAS

```
class CASConsensus extends ConsensusProtocol {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST);
   public Object decide(Object value) {
      propose(value);
      int i = ThreadID.get();
      if (r.compareAndSet(FIRST, i)) // I won
           return proposed[i];
      else // I lost
          return proposed[r.get()];
   }
}
```

The one and only thread to win will get its value as the consensus

CAS

- A register with CAS and get has an infinite consensus number

Bit-Stealing

- C++ has pointers
  - To atomically mark a pointer with a boolean value:
    - Observe that pointers to objects never have the least significant two bit set
    - In fact, alignment is usually in multiples of 16, so 4 least significant bits are zero
    - Use one of these bits as a marker
      - Can still recover the original pointer

Bit-Stealing

- In Java:
  - java.util.concurrent.atomic has an object
  - AtomicMarkableReference<T>:
    - Reference to an object of type T
    - Boolean mark field
    - Can be updated atomically together or individually

Bit Stealing

- Interface:

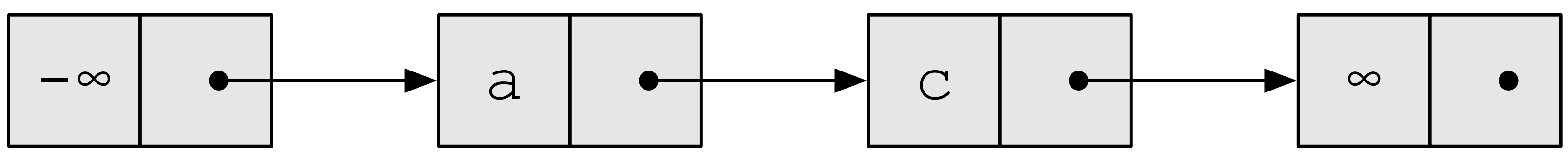
```
public boolean compareAndSet(T expectedReference,
                             T newReference,
                             boolean expectedMark,
                             boolean newMark);

public boolean attemptMark(T expectedReference,
                           boolean newMark);

public T get(boolean[] marked);
```

  - returns the encapsulated reference and stores mark at position 0 in the array

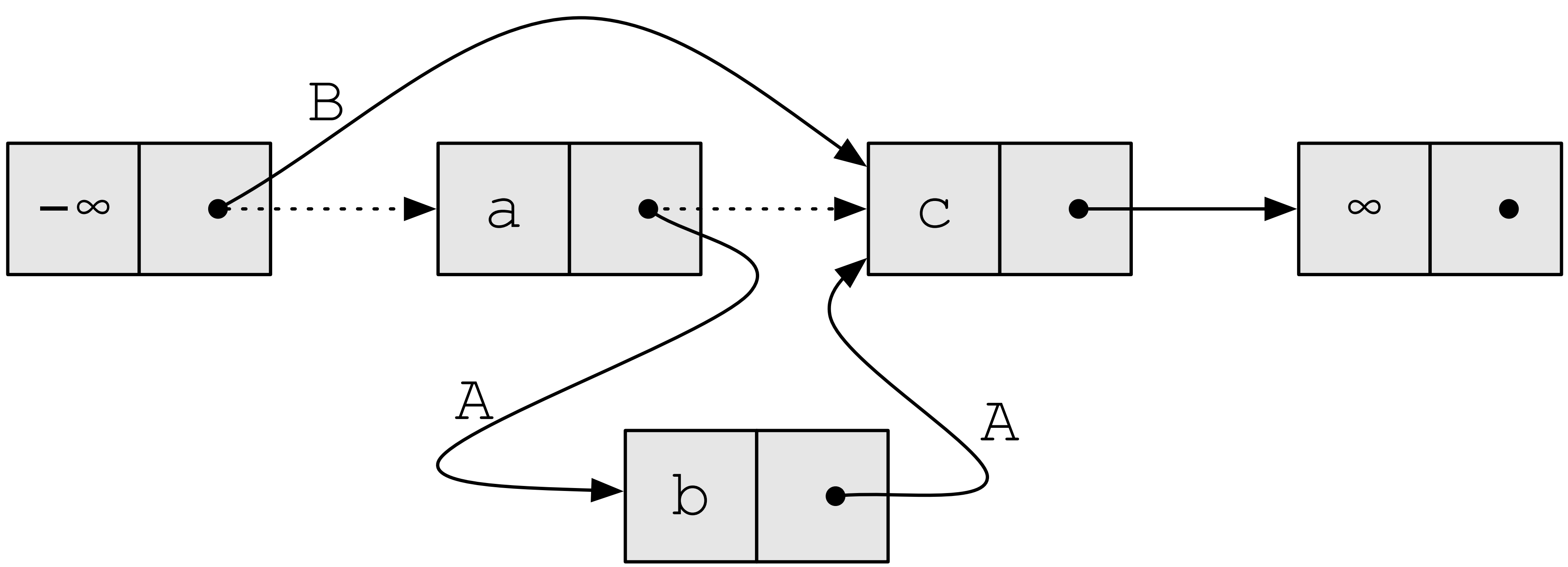
Lock-free Lists

- First attempt:
  - Use compareAndSet to change the next field
- Example:
  - 

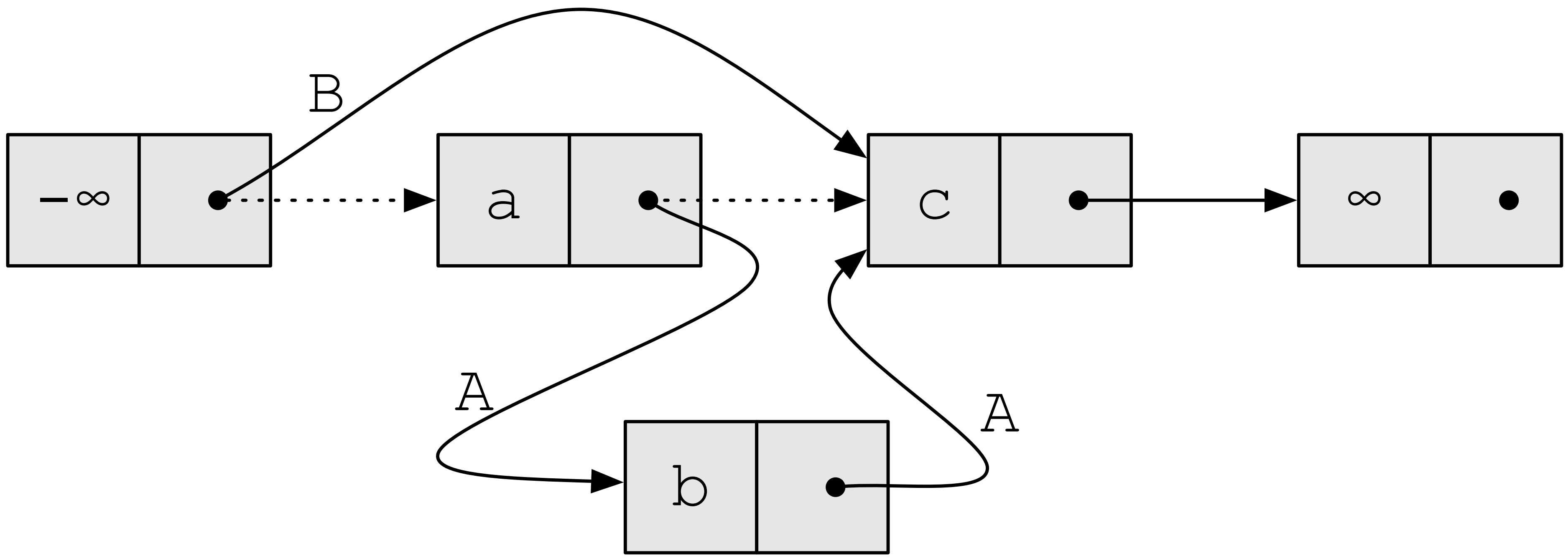
  - Thread I: add b
  - Thread II: remove a

Lock-free Lists

- Thread A applies CAS to a.next
- Thread B applies CAS to -∞.next
- Both succeed regardless of who comes first:



Lock-free Lists



- We must prevent manipulation of a removed node!

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists

Lock-free Lists