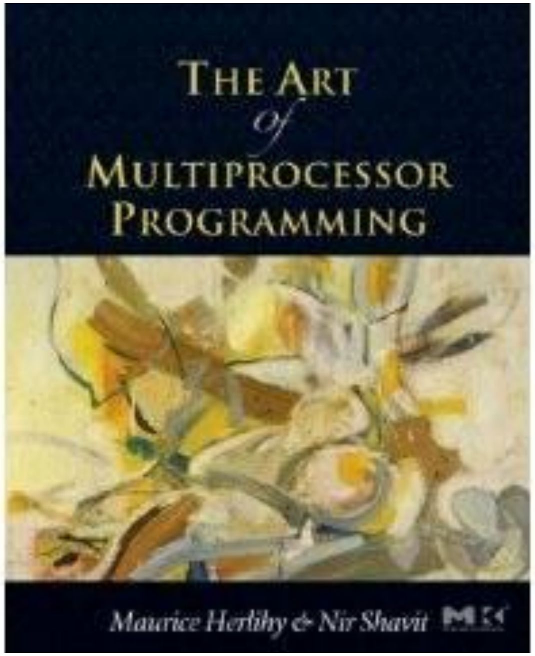
Department of Computer Science

COS 226

Chapter 9

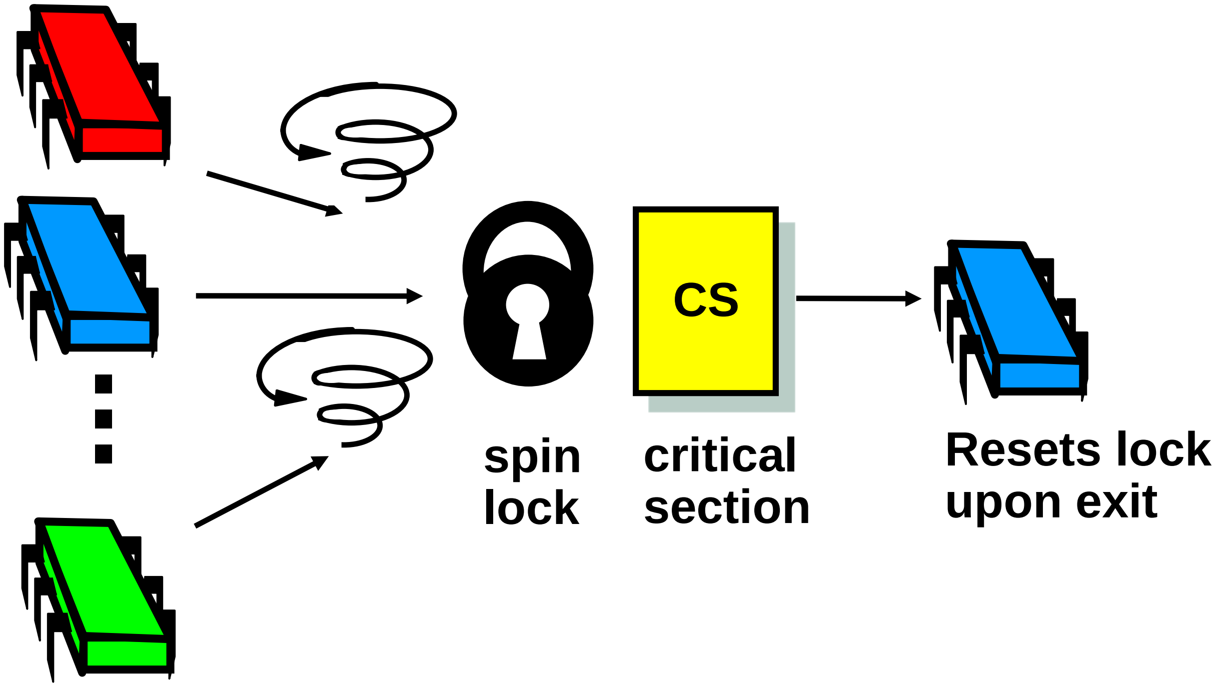
Linked Lists: The Role of Locking

Acknowledgement


THE ART of MULTIPROCESSOR PROGRAMMING
Maurice Herlihy & Nir Shavit
MK

- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Last Lecture: Spin-Locks



Spin locks

- In Chapter 7 we saw how to build scalable spin locks that provide mutual exclusion efficiently

So, how do we construct a scalable concurrent data structure?

- The most obvious solution would be to take a sequential implementation of the class, add a scalable lock and make sure that every method call acquires and releases the lock
- = coarse-grained synchronization
- What is the potential problem with this?

Problem

- A class that uses a single lock to mediate all its method calls is not always scalable
- Coarse-grained synchronization works well when the level of concurrency is low
- However when too many thread tries to acquire the lock, it forms a bottleneck

This Chapter

- Introduce four “patterns”
  - Bag of tricks …
  - Methods that work more than once …
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput

First:
Fine-Grained Synchronization

- Instead of using a single lock …
- Split object into
  - Independently-synchronized components
- Methods calls interfere only when the 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
- Lazy synchronization splits it into these two removal phases

Fourth:
Lock-Free Synchronization

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

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-Based Sets

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

Returns true if x was not already in set

List-Based Sets

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

Returns true if x was in set

List-Based Sets

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

Returns true if x was in set

List-Based Sets

- A set is implemented as a linked list of nodes
- Node<T> has three fields:
  - Item – actual item
  - Key – item’s hash code, nodes are sorted according to key
  - Next – reference to next node in list

List Node

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

List-Based Sets

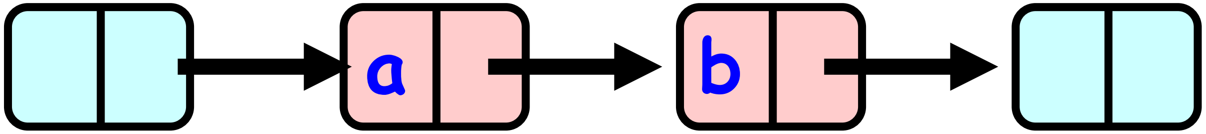
- Lists has two types of nodes:
  - Regular nodes – hold items
  - Sentinel nodes – head and tail
- Each thread that traverses through the list use:
  - curr – a “pointer” to the current node
  - pred – a “pointer” to the node’s predecessor

Freedom from interference

- We assume that add(), remove() and contains() are the only methods that can modify nodes
- We also assume that sentinel nodes cannot be added or removed
- And nodes are sorted by keys and keys are unique

Reasoning about Concurrent objects

- Concrete representation:

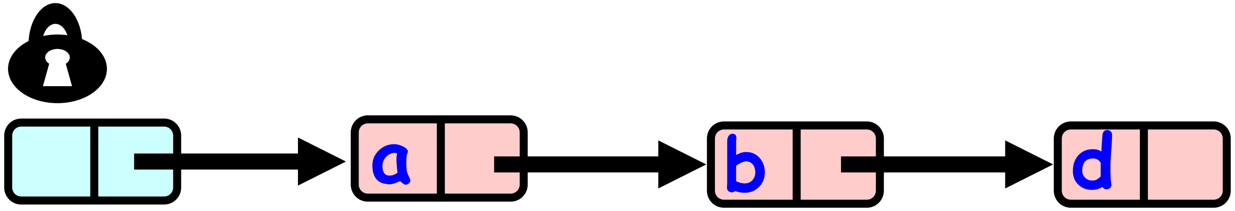


- Abstract Value:
  - {a, b}

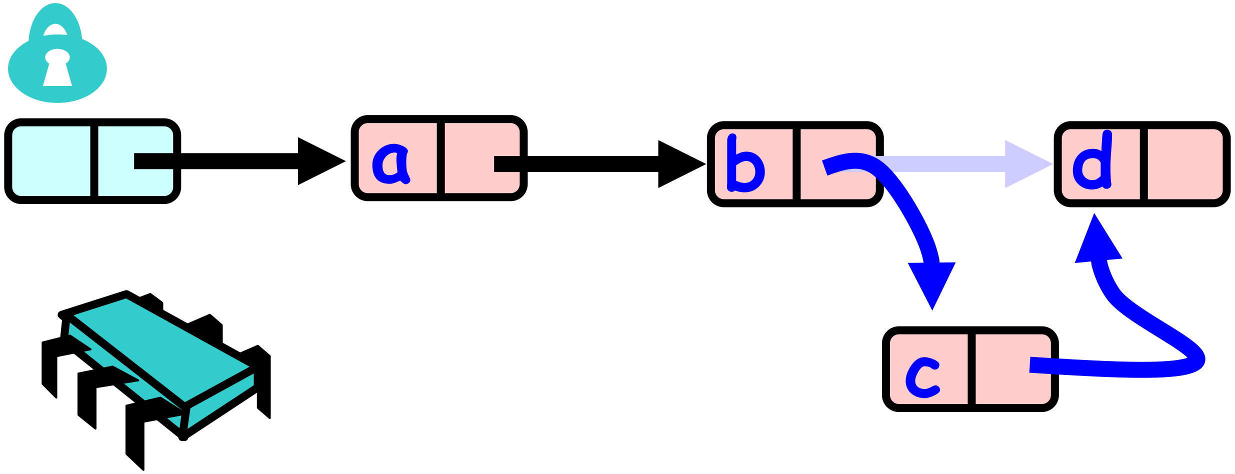
Safety and Liveness?

- Safety:
    - Linearizability
- Liveness
    - Deadlock-free
    - Starvation-free
    - Nonblocking?

Coarse Grained Locking



Coarse Grained Locking



Coarse-grained synchronization

- One concurrent data structure
- One lock
    - Method acquires and releases lock with each access
- Multiple threads

Coarse-grained synchronization

```
public class CoarseList<T> {
  private Node head;
  private Lock lock = new ReentrantLock();
  public CoarseList() {
     head = new Node(Integer.MIN_VALUE);
     head.next = new
     Node(Integer.MAX_VALUE);
  }
```

```
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 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();
   }
```

Acquire lock

```
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();
   }
```

Starting positions for pred and curr

```
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();
   }
```

Traverse through list to find correct position

```
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();
   }
```

If element already exists
return false

```
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();
  }
```

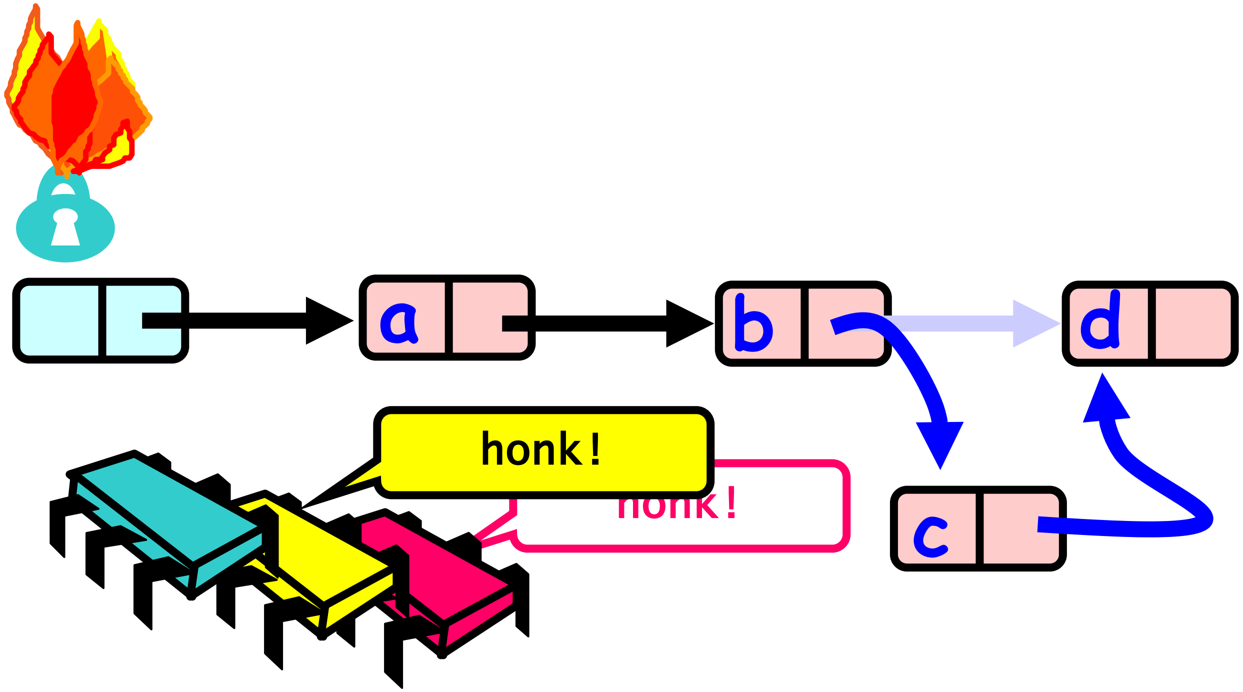
Create node and insert in list

```
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();
   }
```

Release lock

```
public boolean remove(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) {
             pred.next = curr.next;
             return true;
       } else
             return false;
   } finally {
       lock.unlock();
   }
```

Coarse Grained Locking



Simple but hotspot + bottleneck

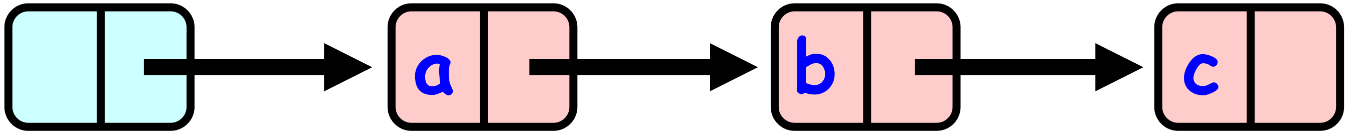
Coarse-grained synchronization

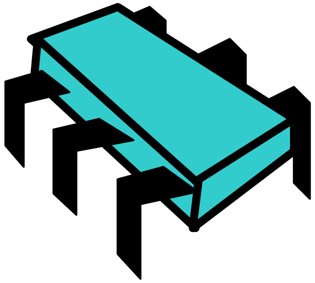
- Easy to implement
- Simple, clear and correct
  - Deserves respect!
- But, works poorly with high contention
  - Queue locks help
  - But bottlenecks still an issue

Fine-grained synchronization

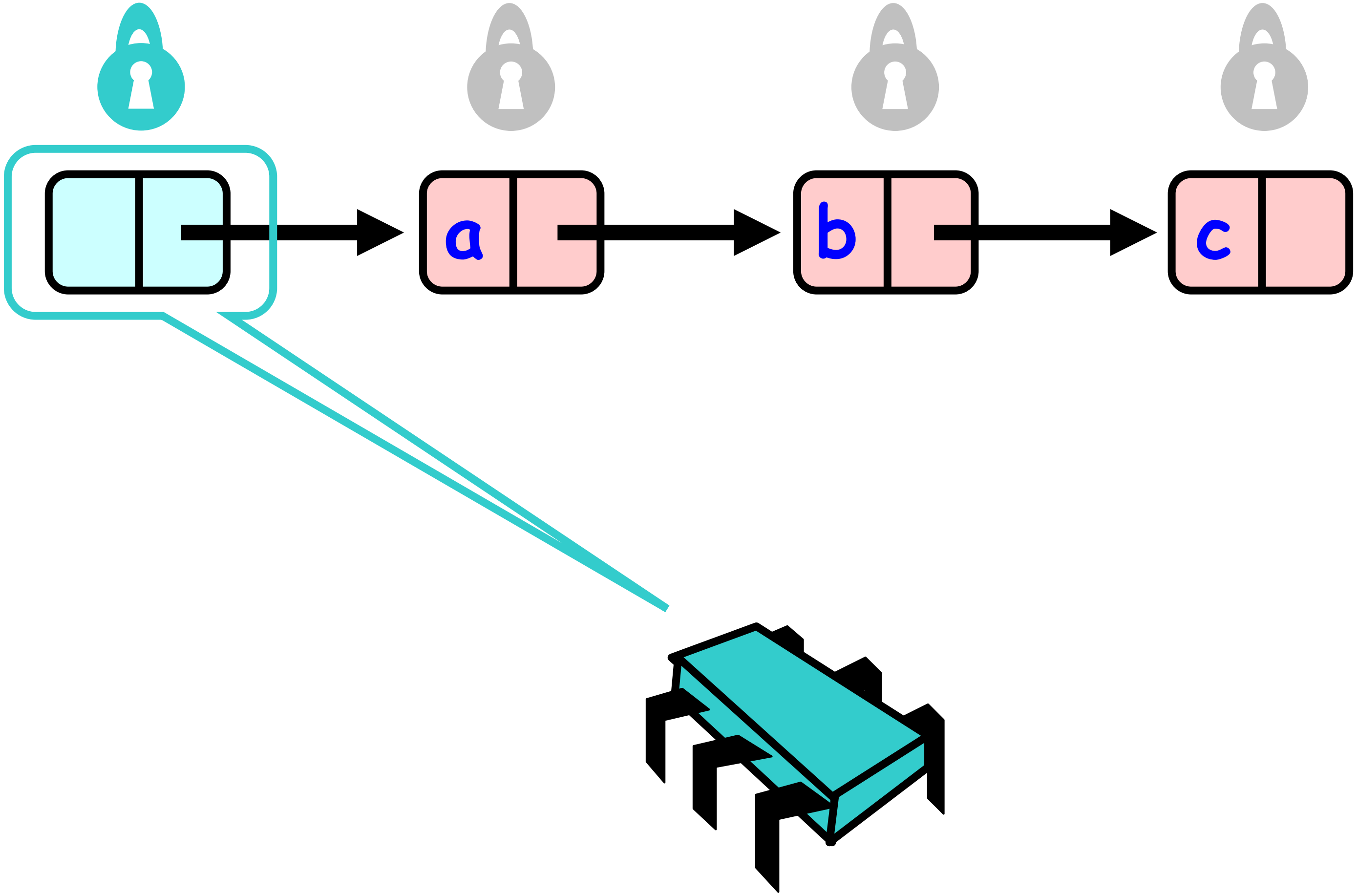
- Instead of locking the list as a whole, place a lock on each entry
- Split object into pieces
    - Each piece has own lock
    - As thread traverses list, he locks each entry with its first visit and unlocks it later
- Concurrent threads can now traverse the list together

Fine-grained synchronization

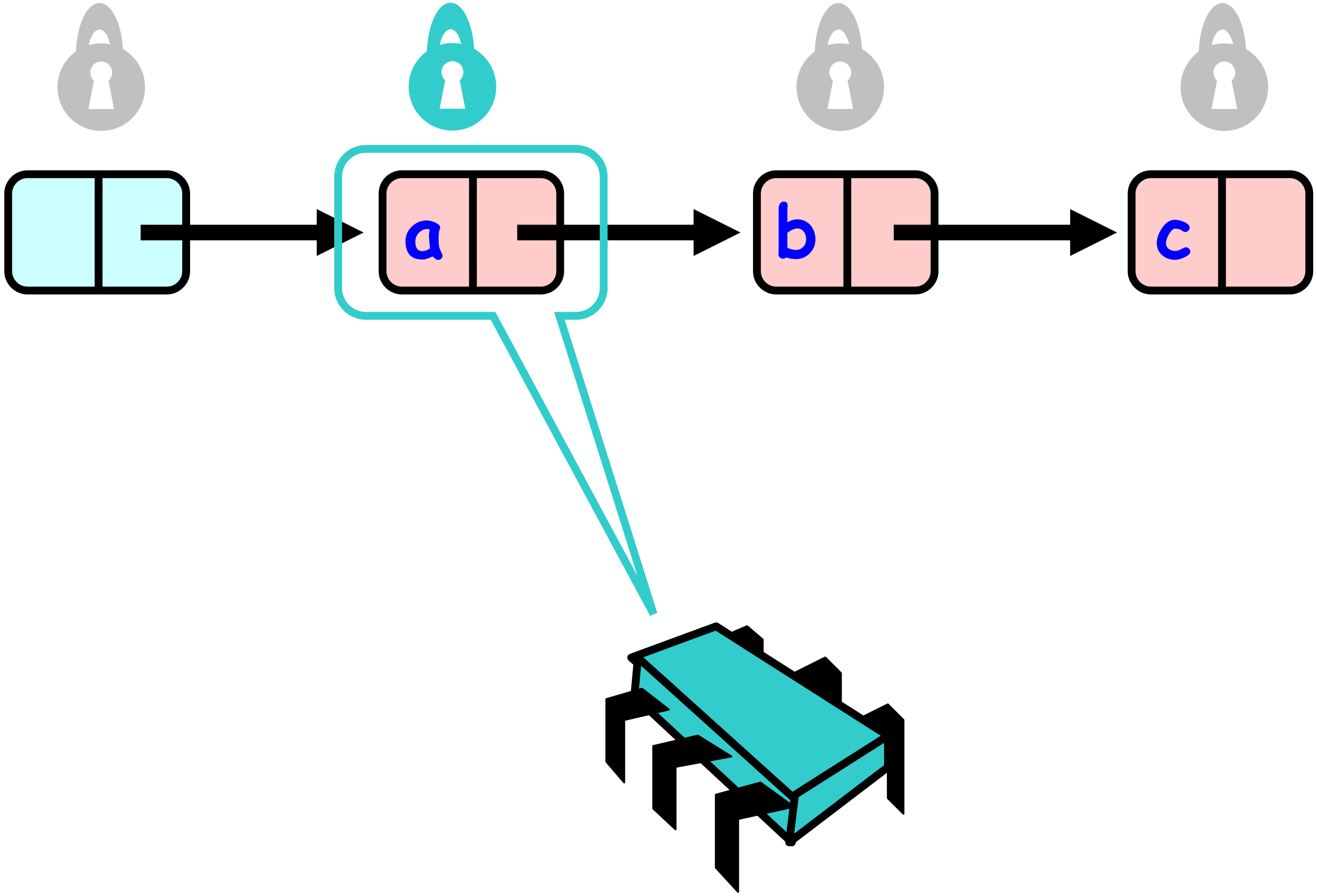




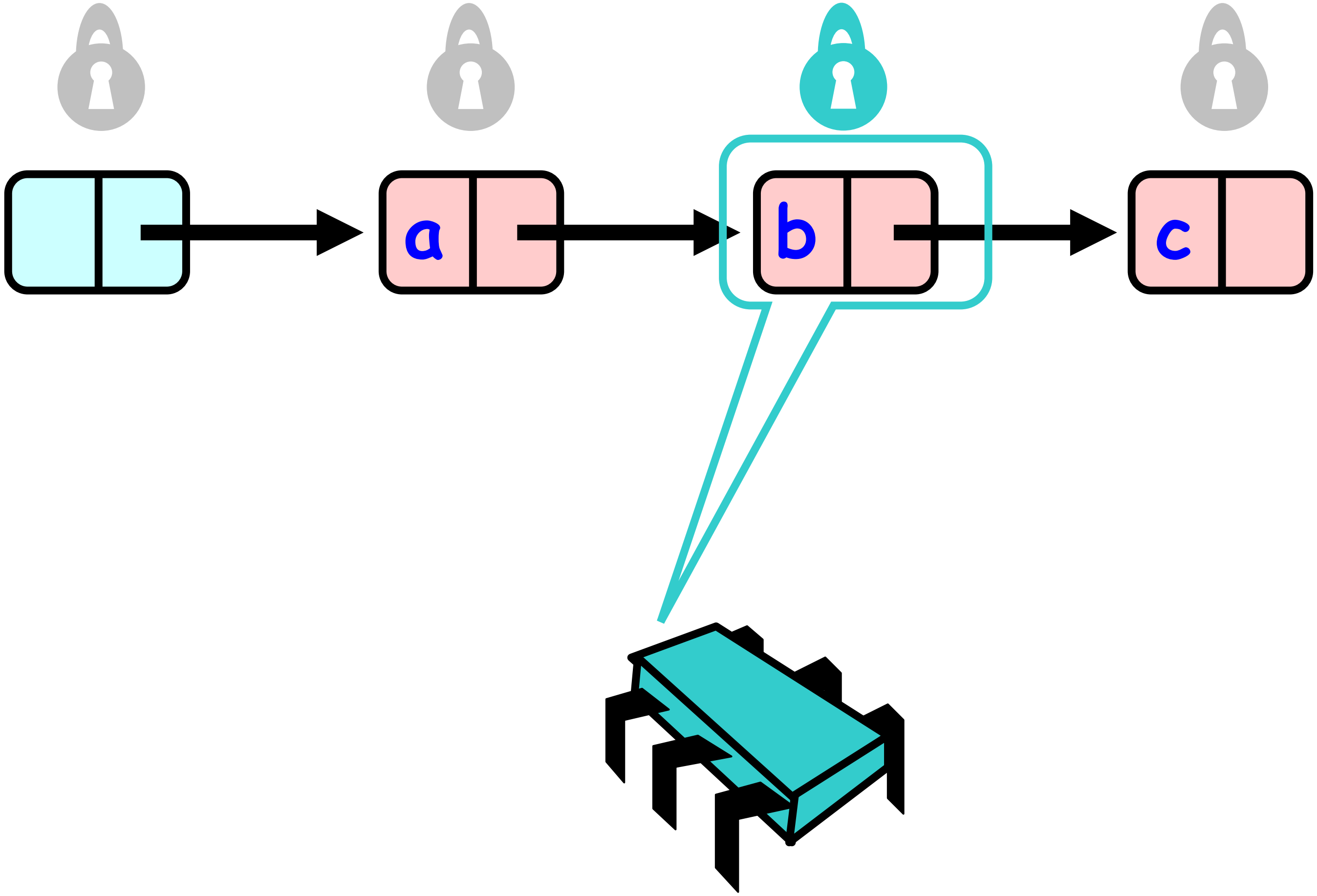
Fine grained synchronization



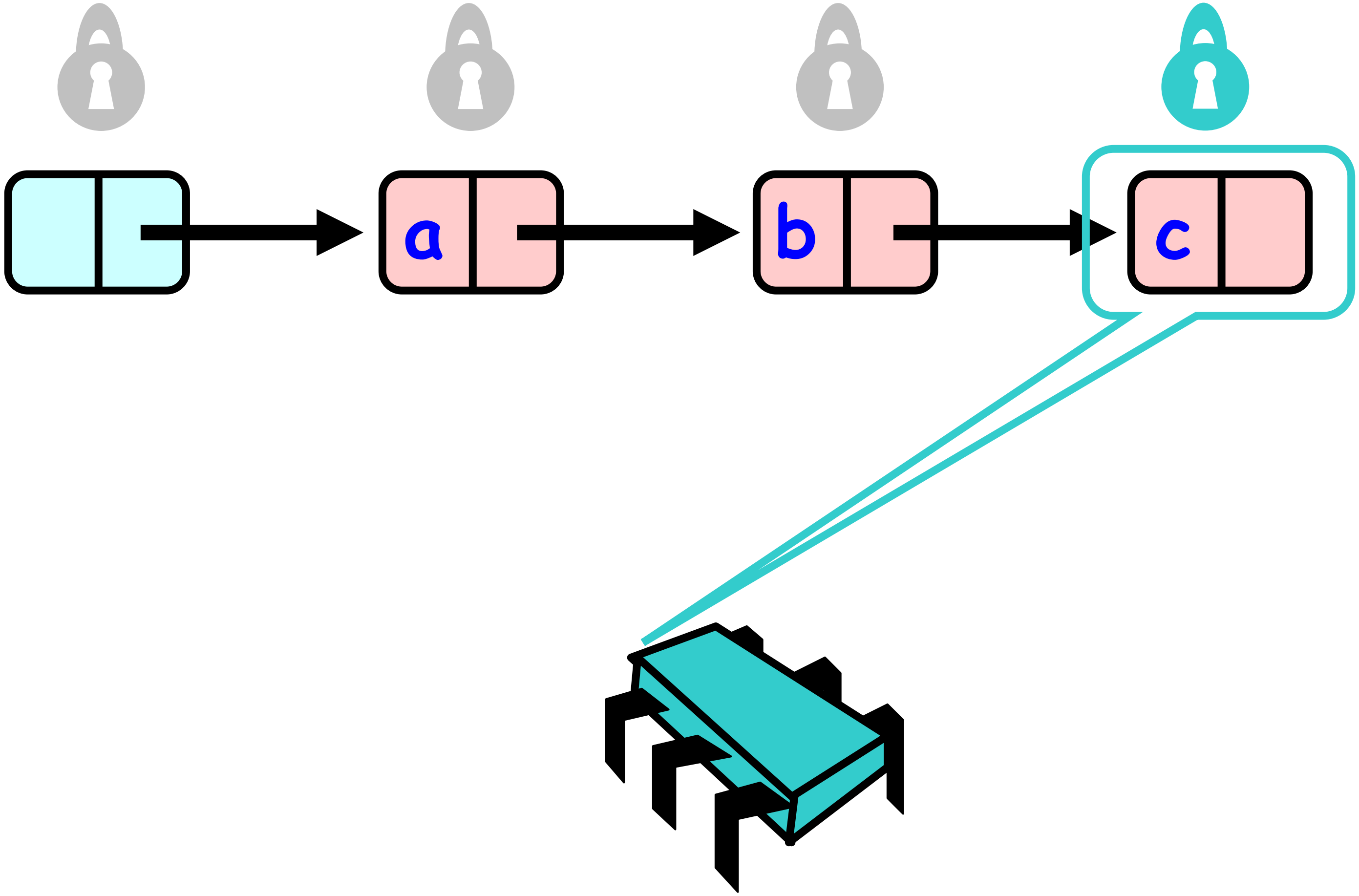
Hand-over-Hand locking



Fine-grained synchronization



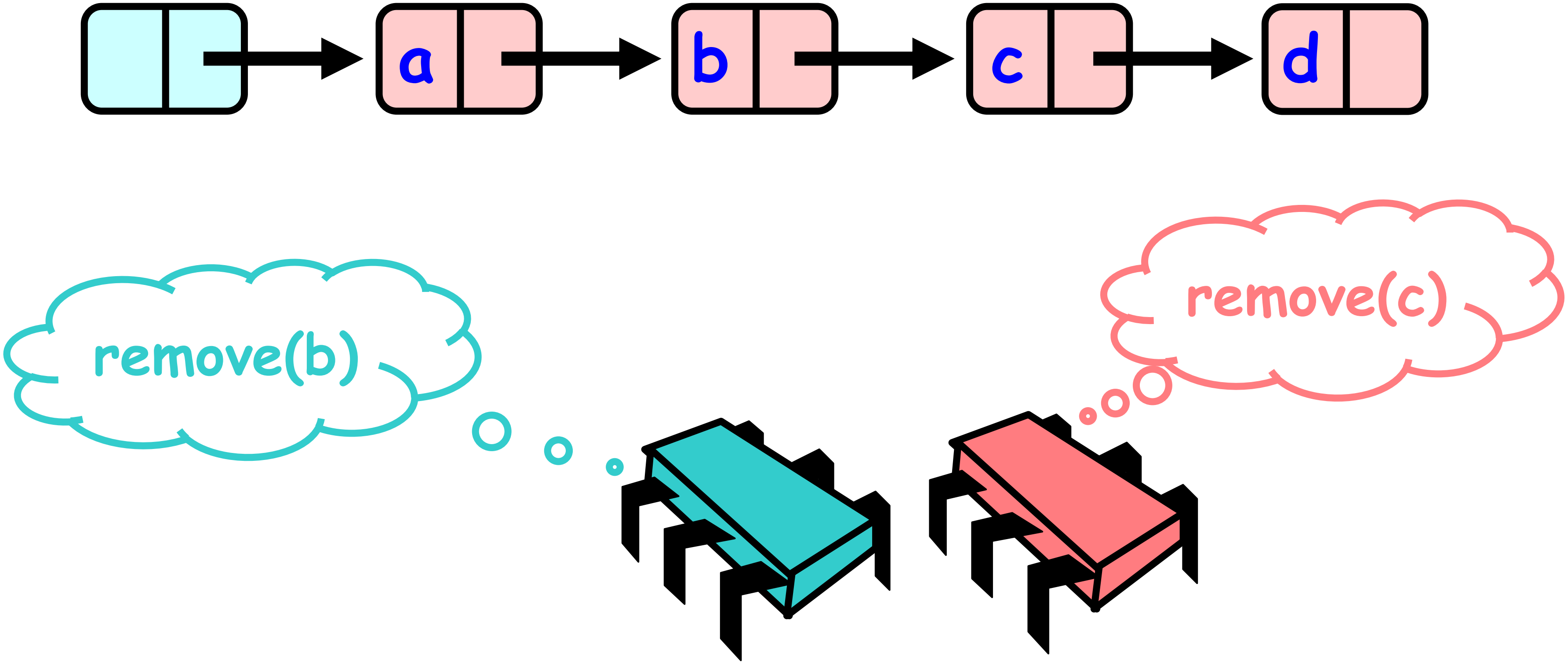
Fine-grained synchronization



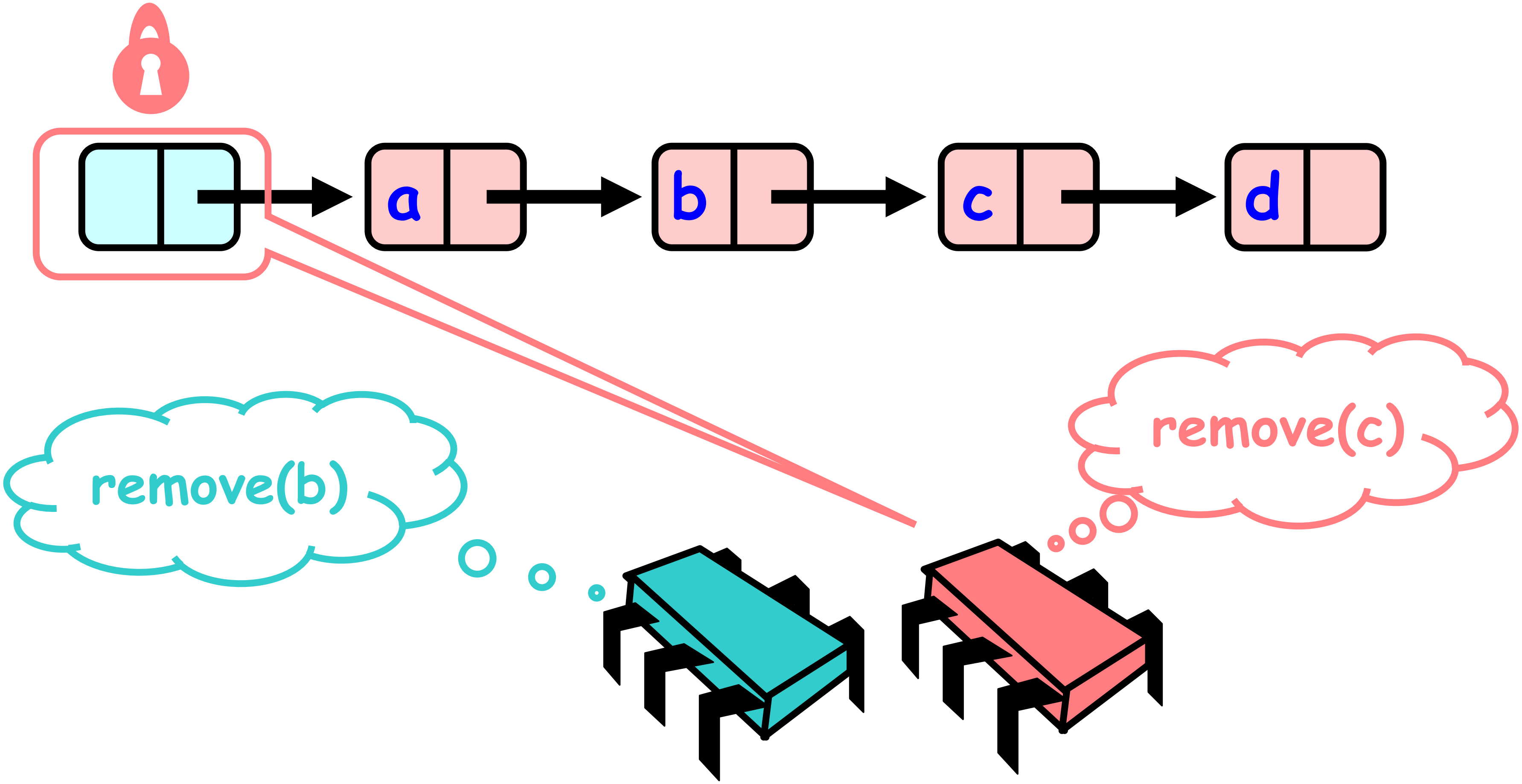
Fine-grained synchronization

- However, it is unsafe to unlock a before locking b

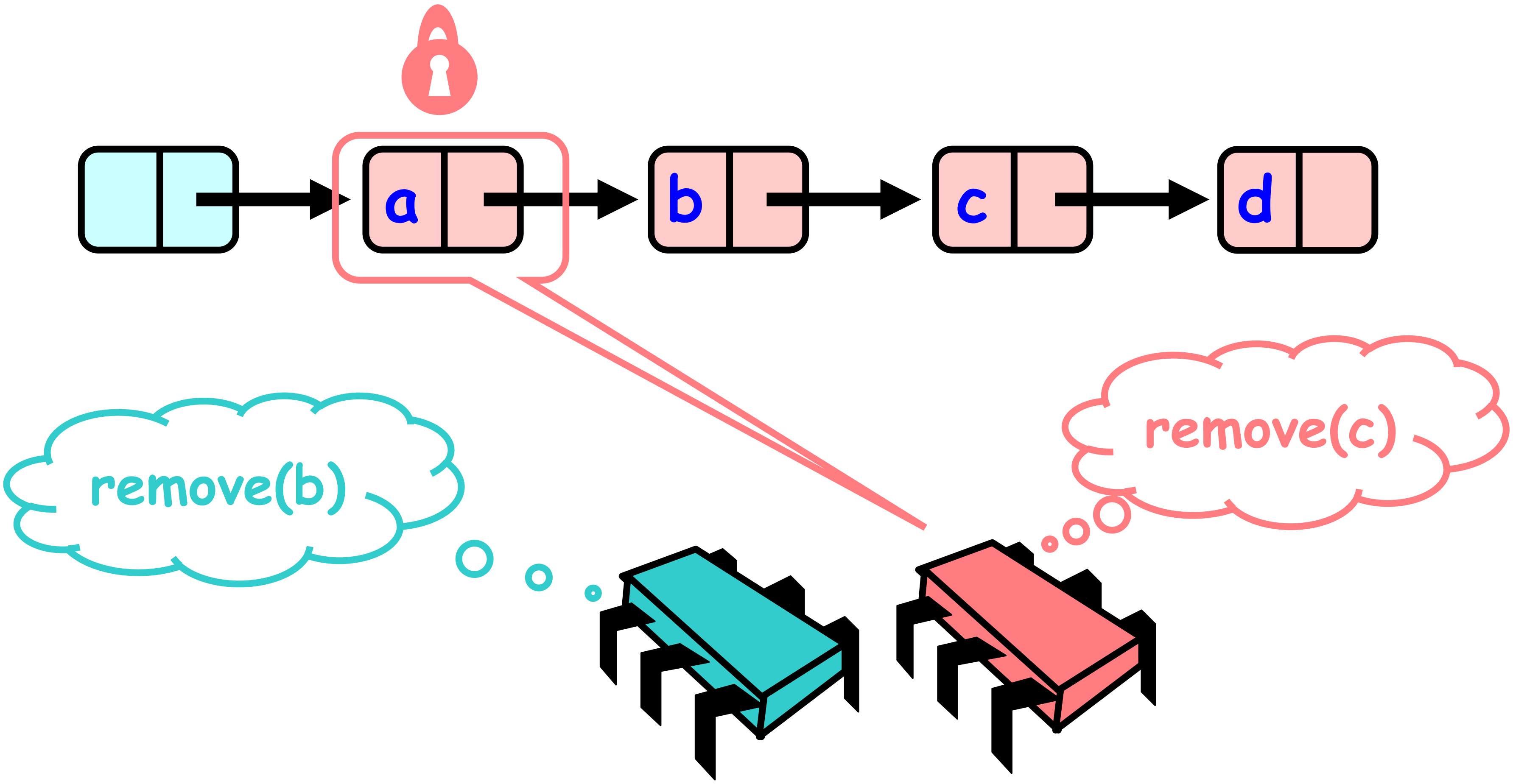
Concurrent Removes



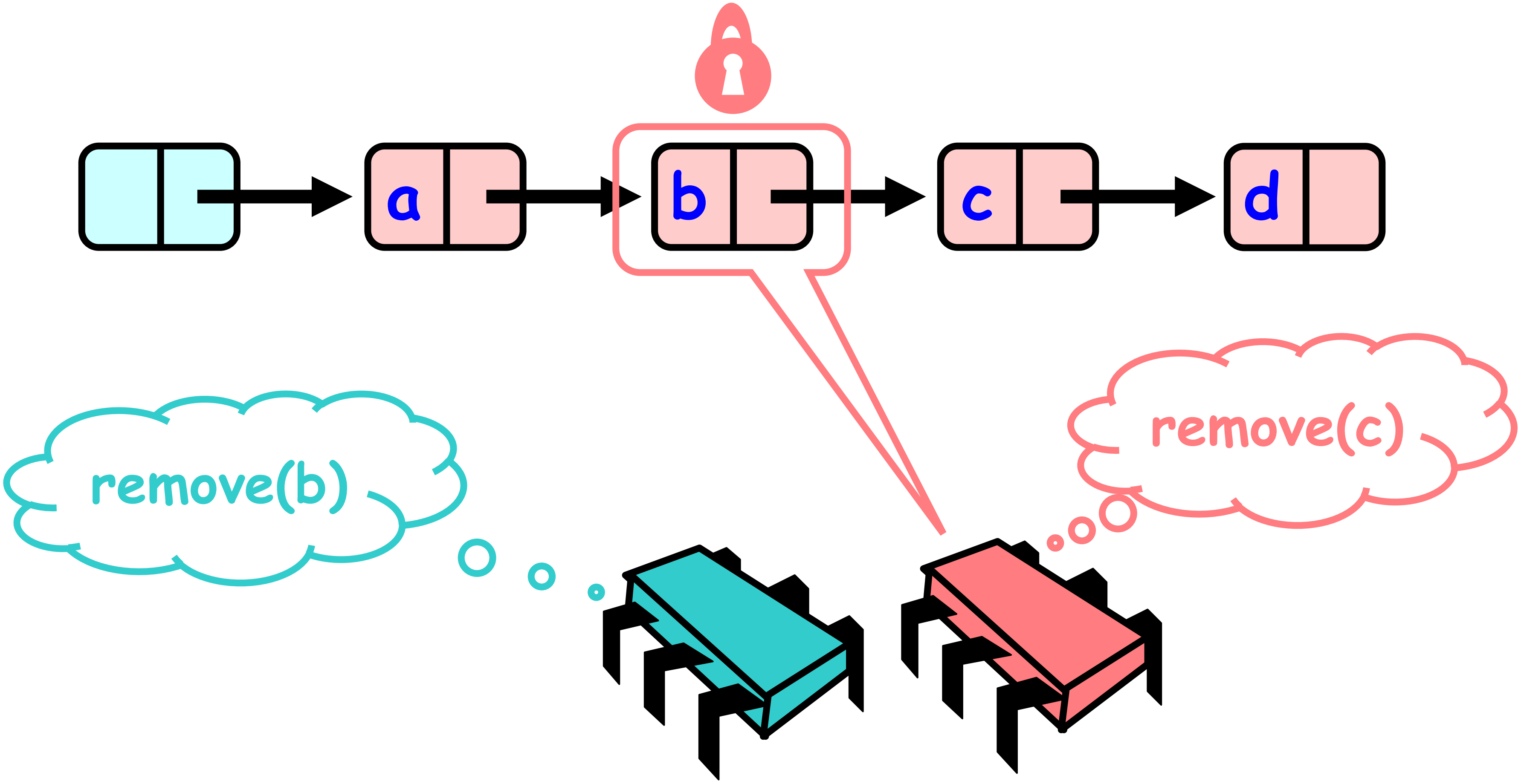
Concurrent Removes



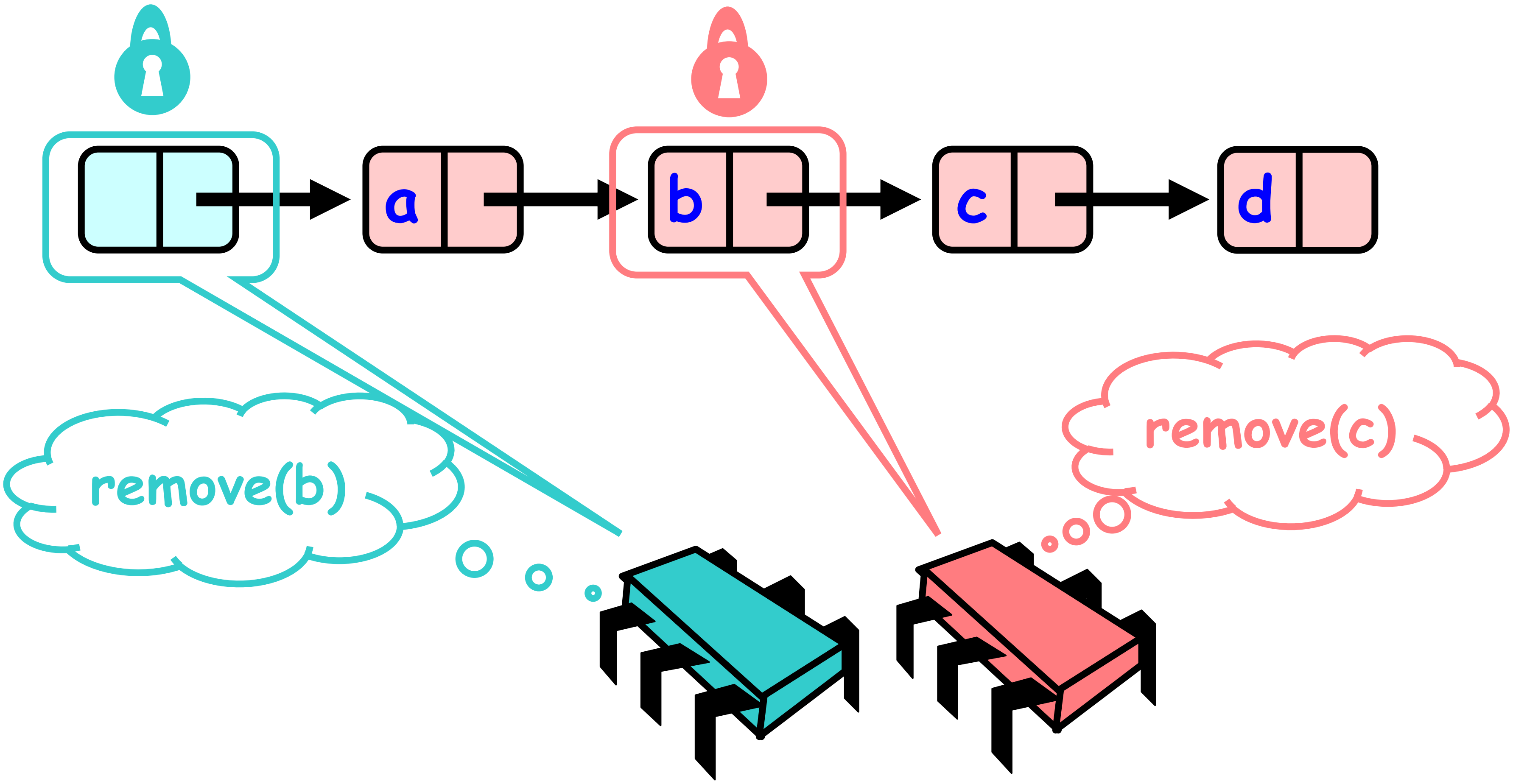
Concurrent Removes



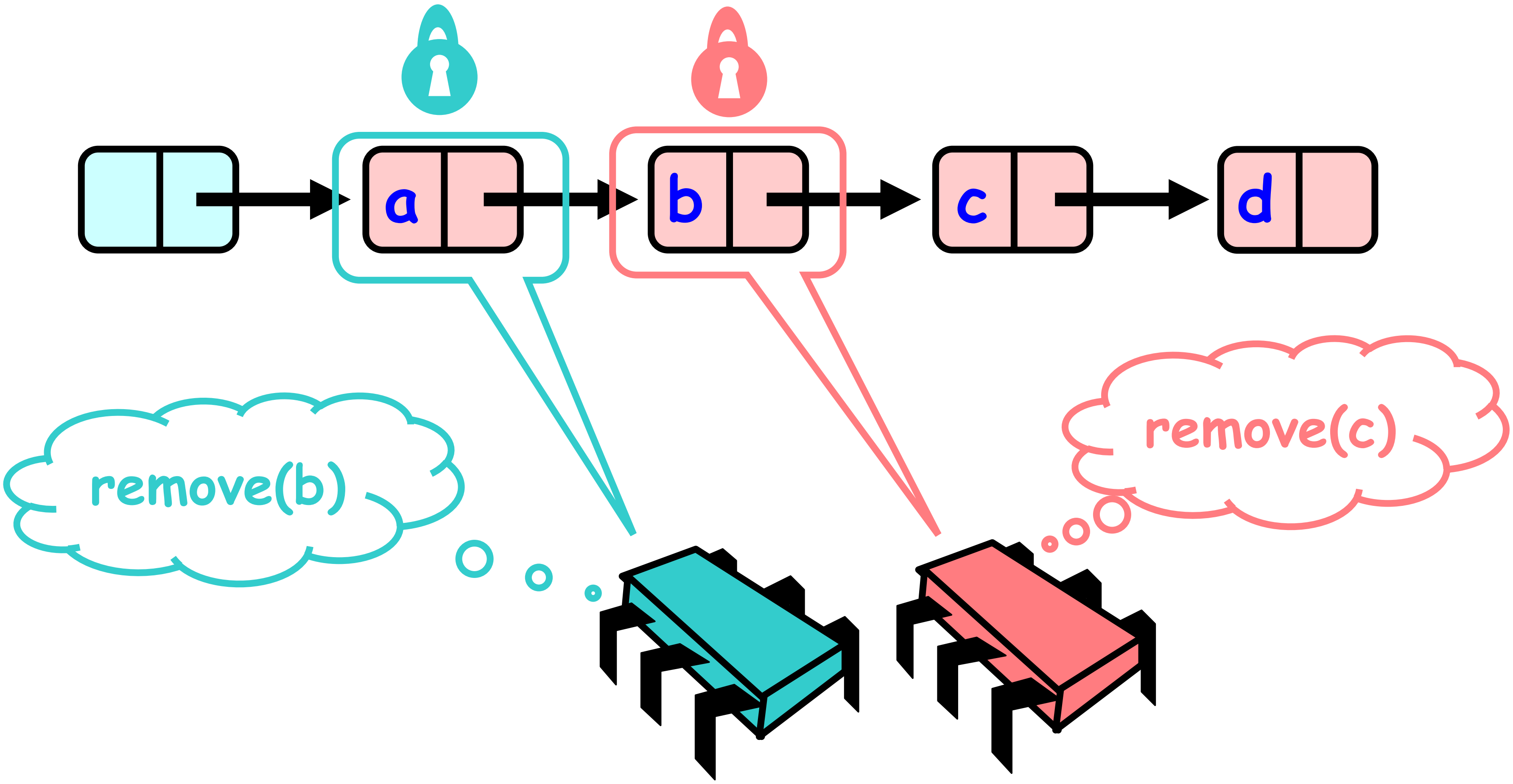
Concurrent Removes



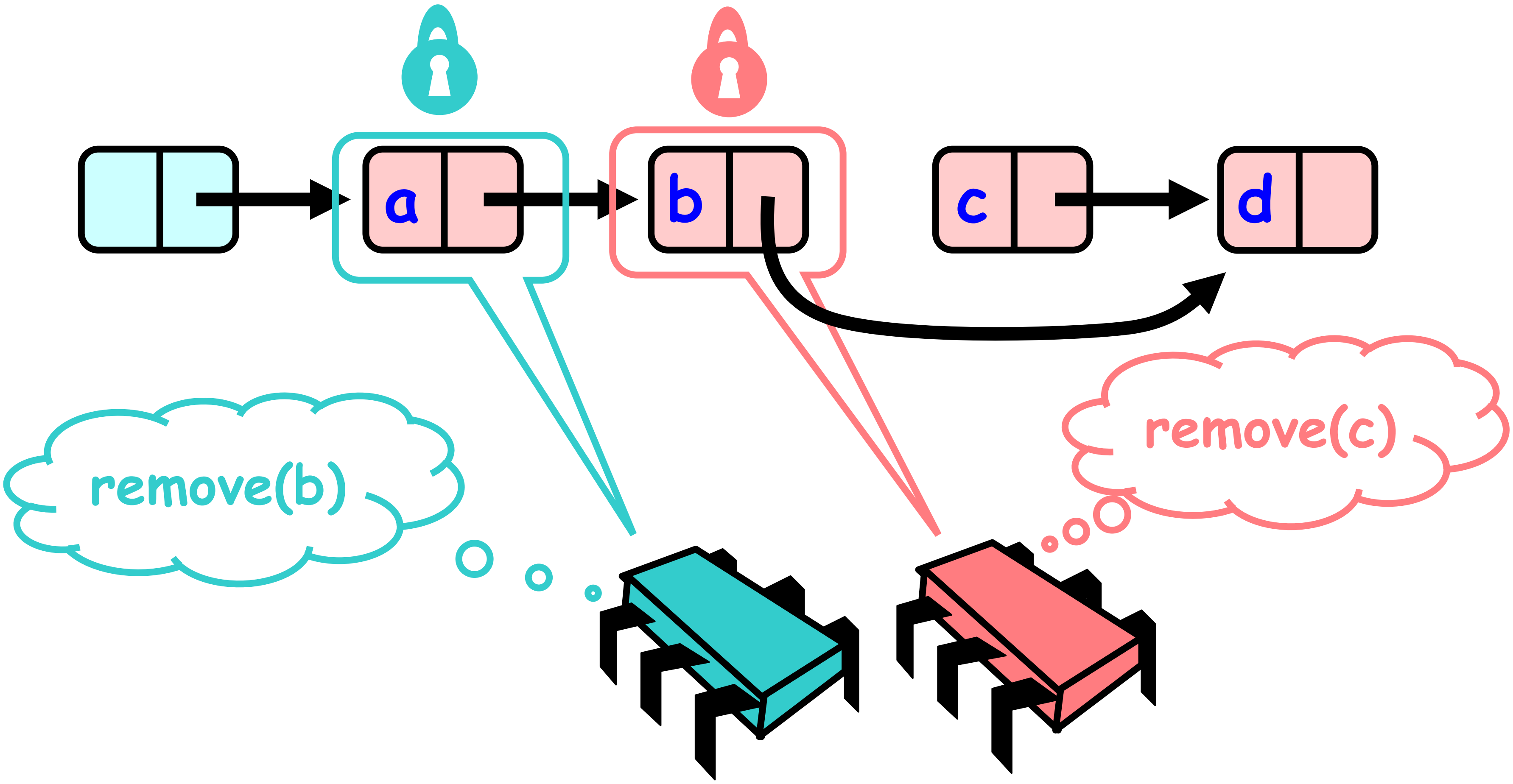
Concurrent Removes



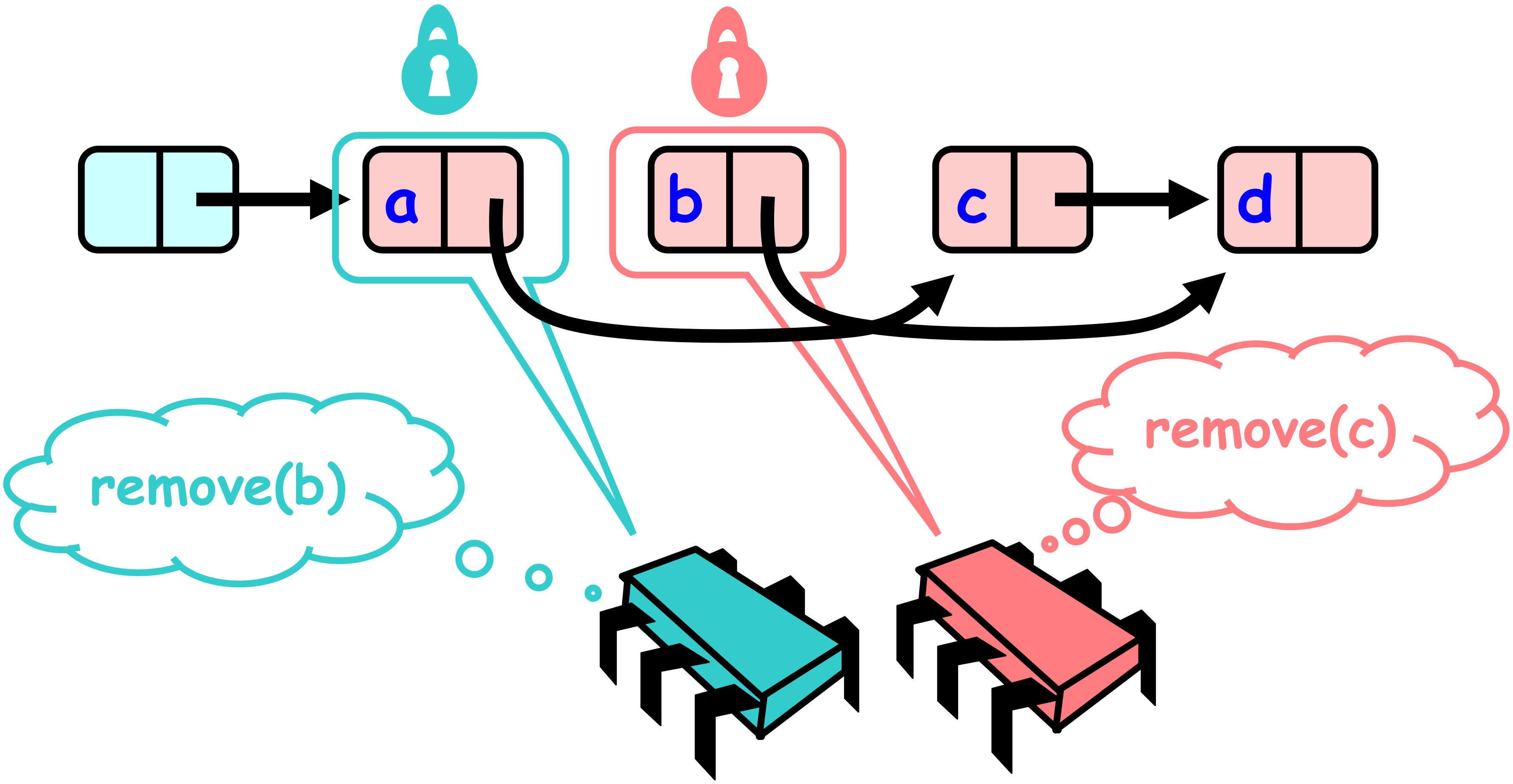
Concurrent Removes



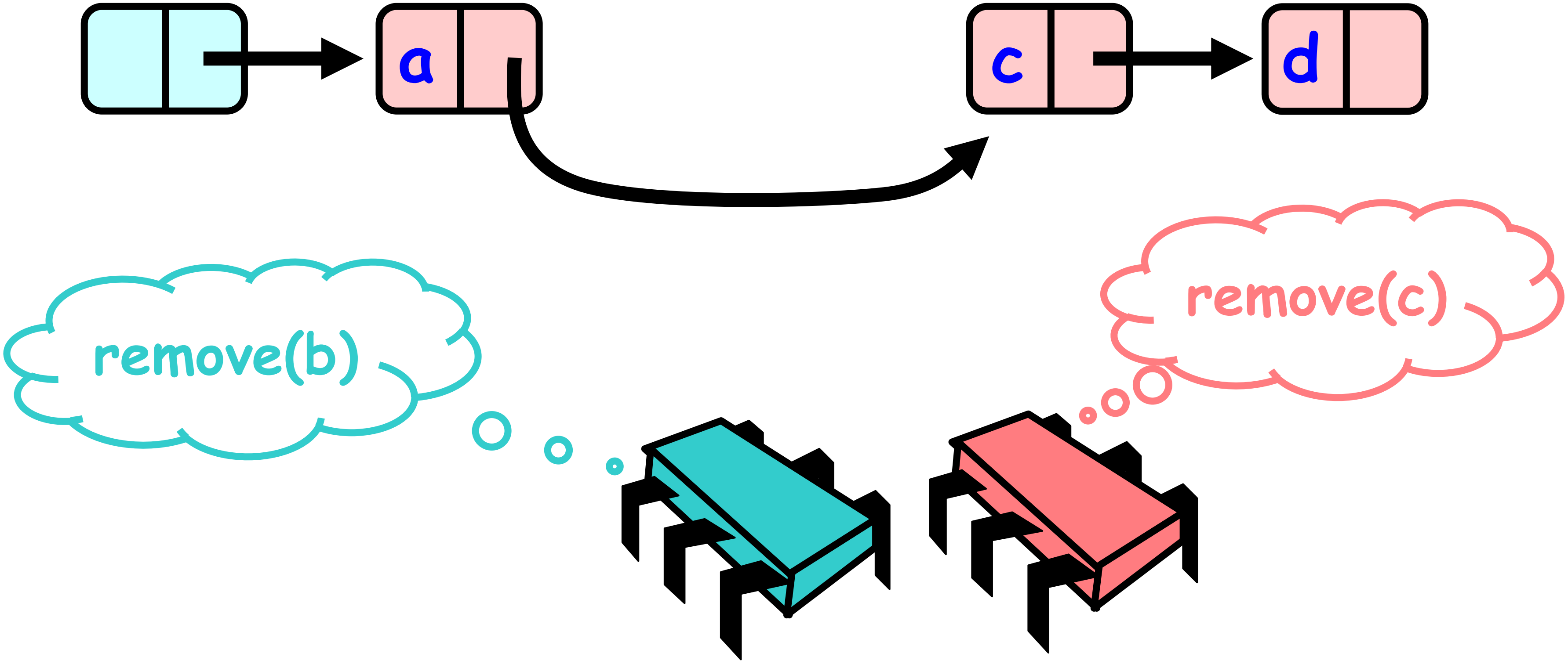
Concurrent Removes



Concurrent Removes

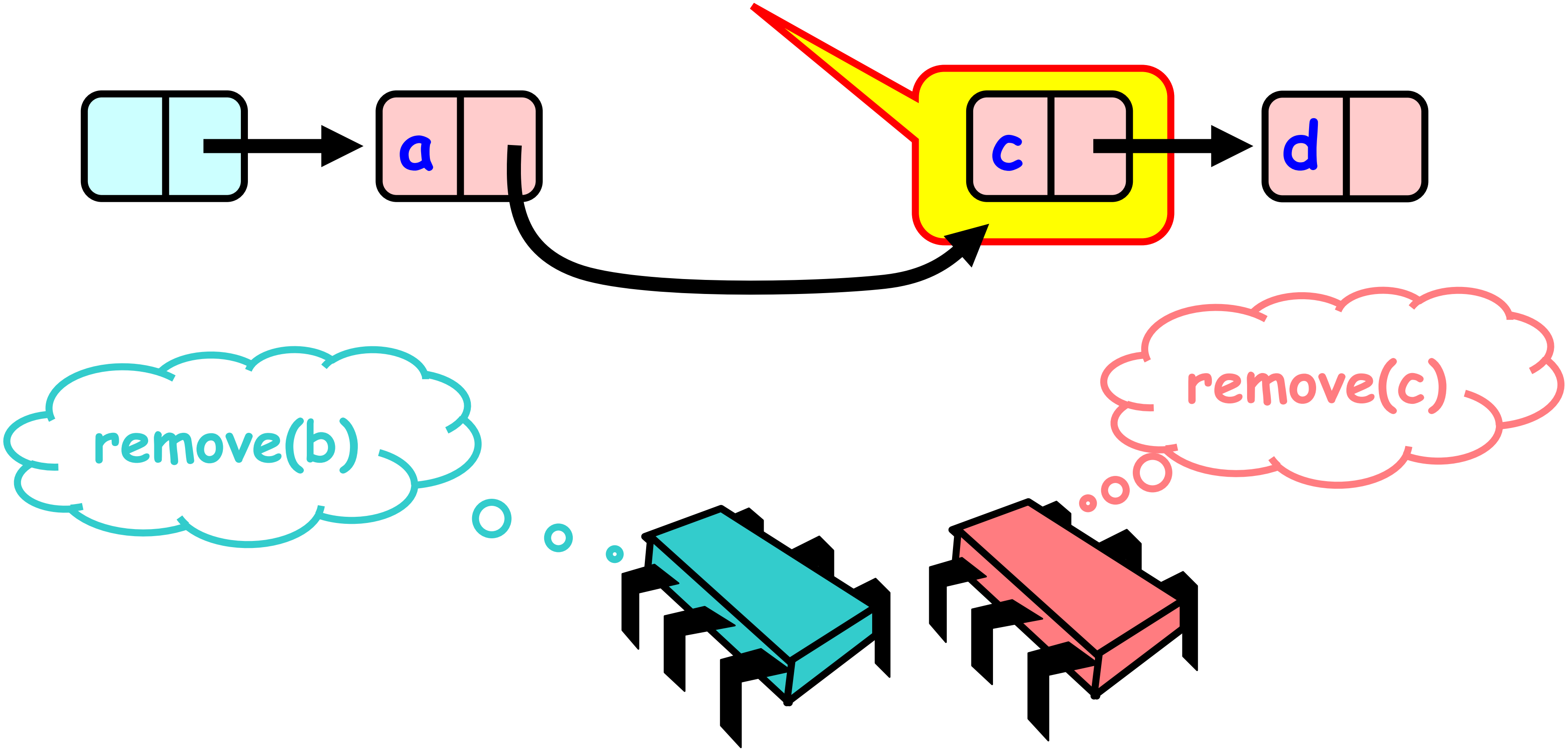


Uh, Oh



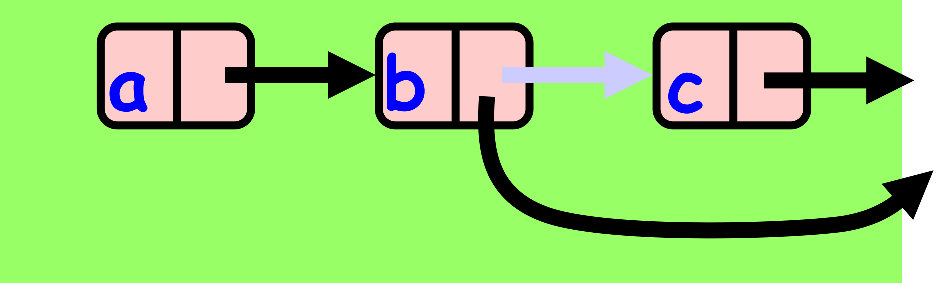
Uh, Oh

Bad news, c not removed

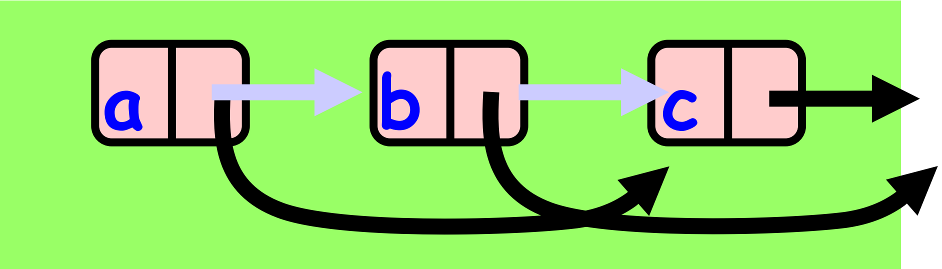


Problem

- To delete node c
  - Swing node b's next field to d



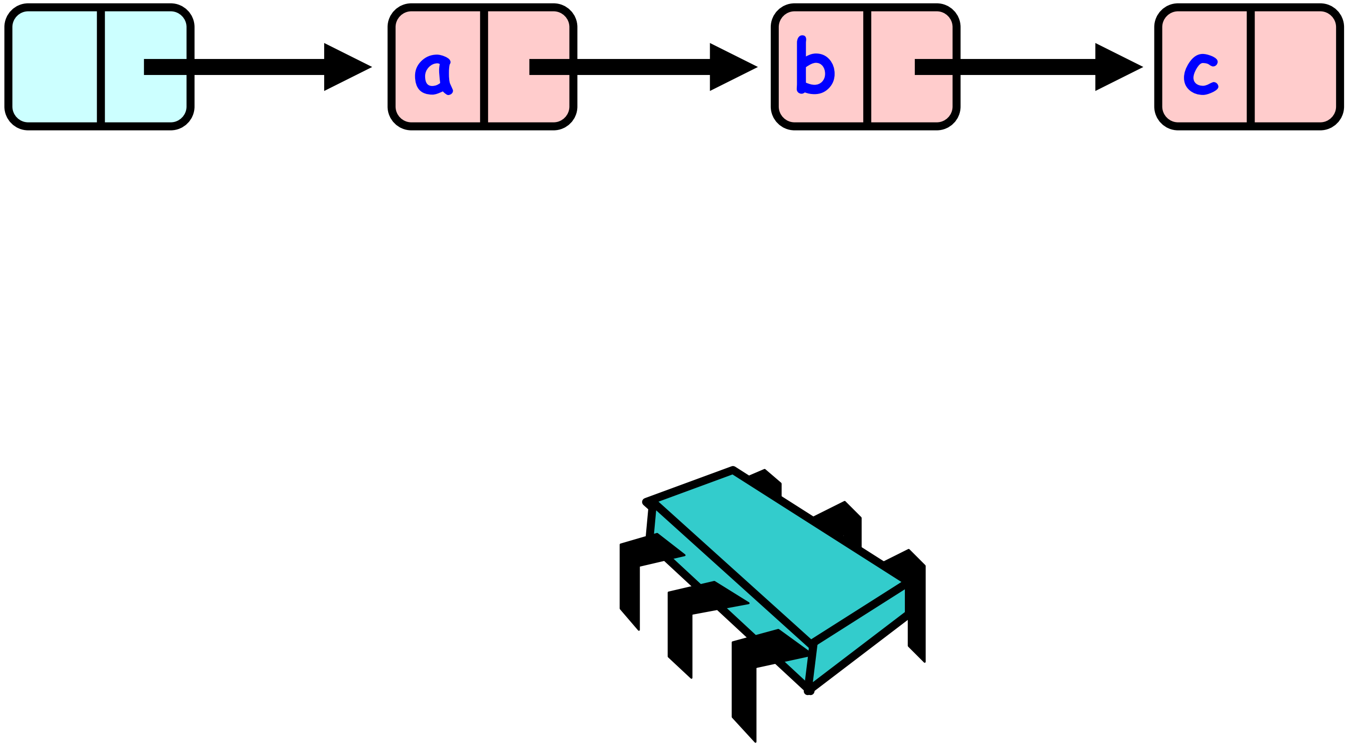
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



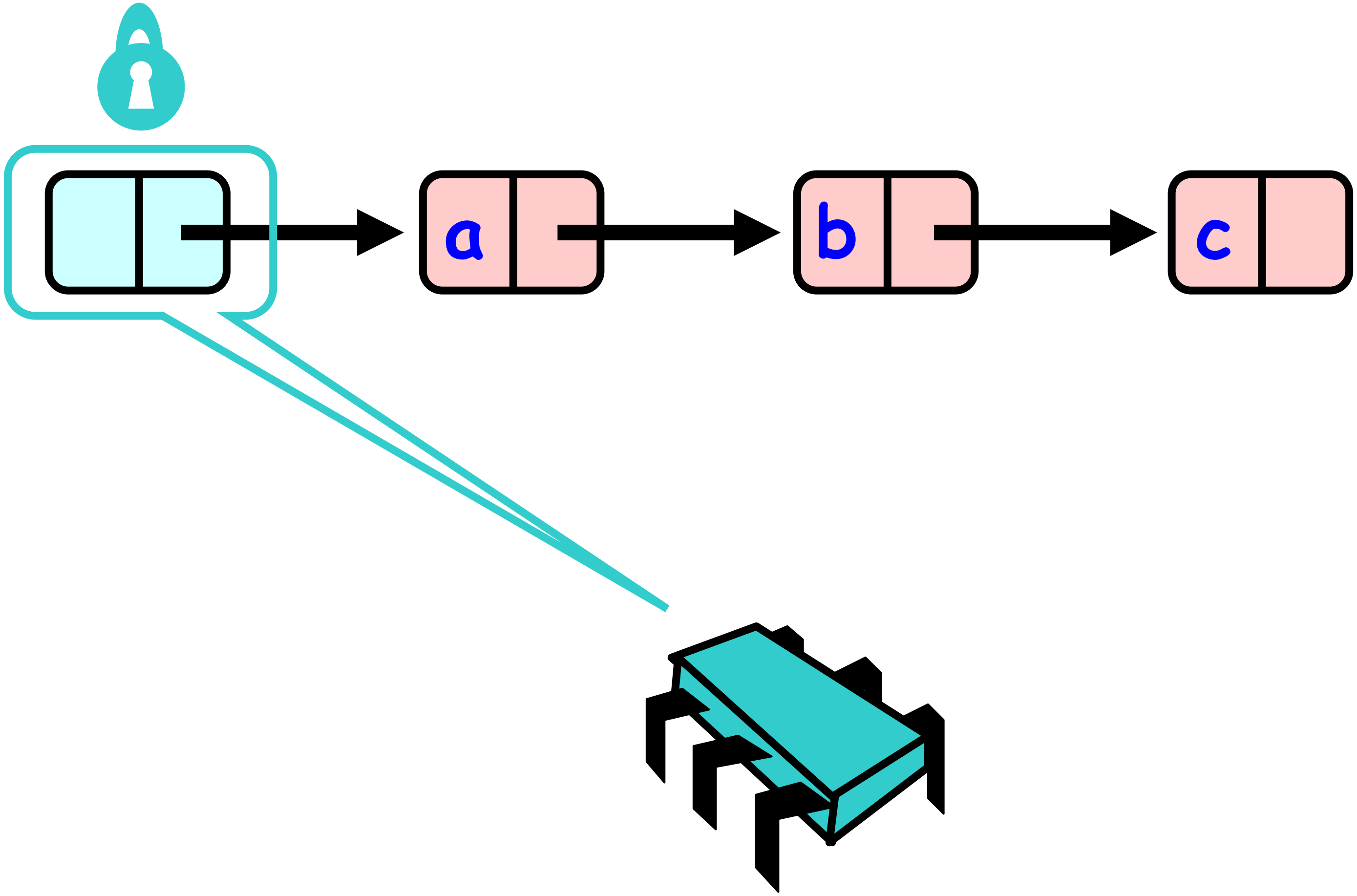
Solution

- Hand-over-hand locking
  - Except for the initial head sentinel node, acquire the next lock while holding the previous lock
  - In other words, hold two locks at a time

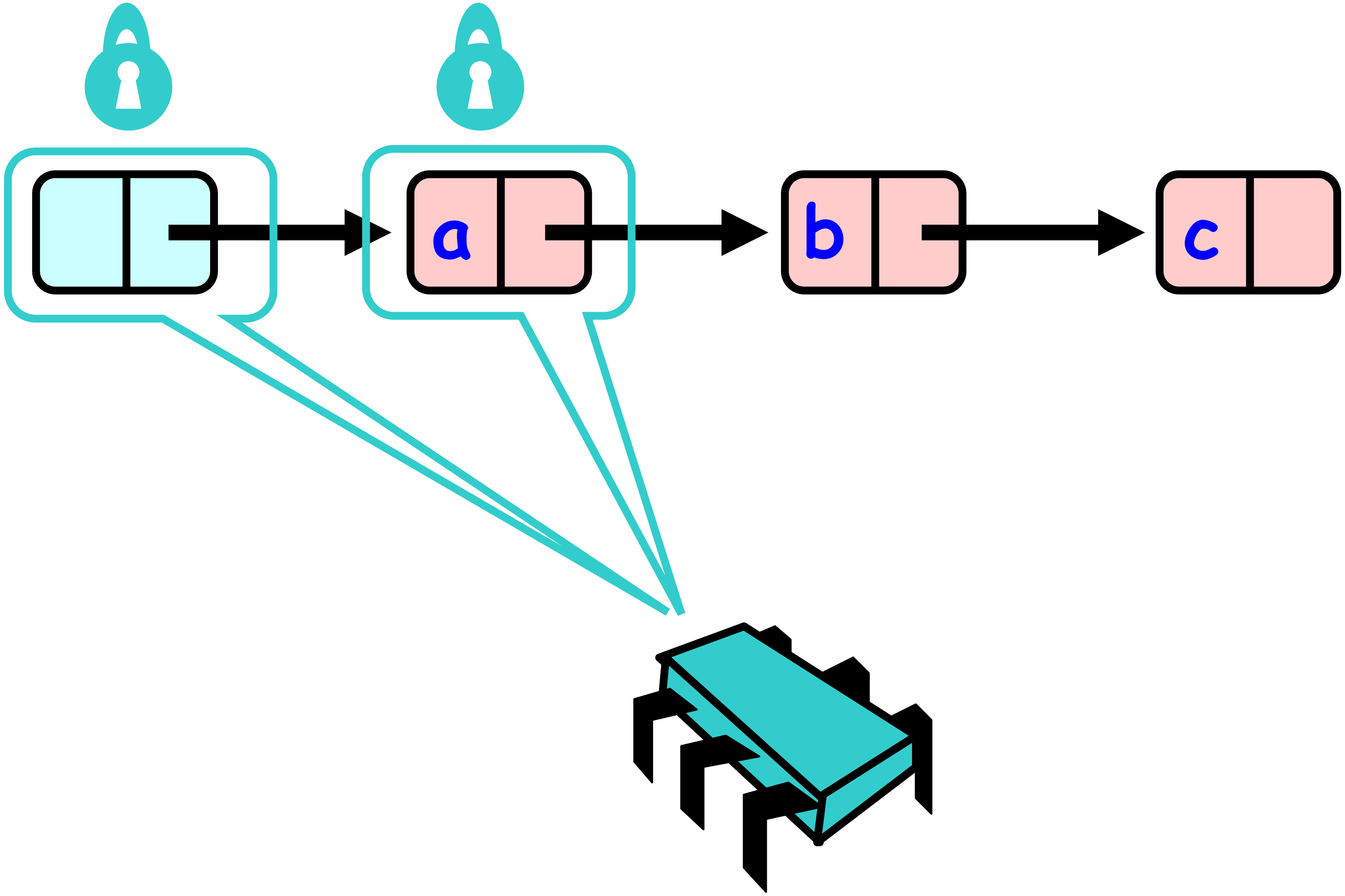
Hand-over-Hand locking



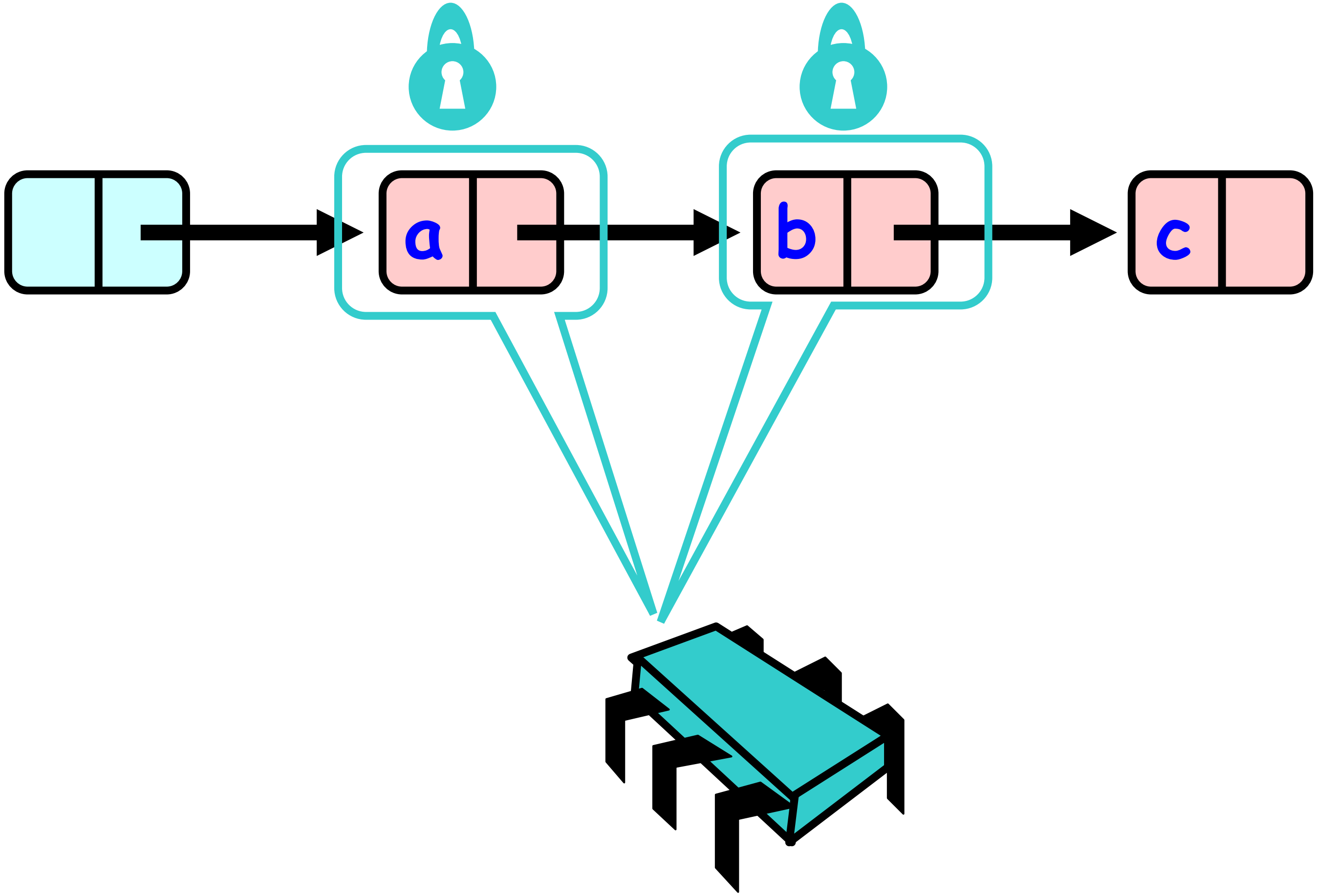
Hand-over-Hand locking



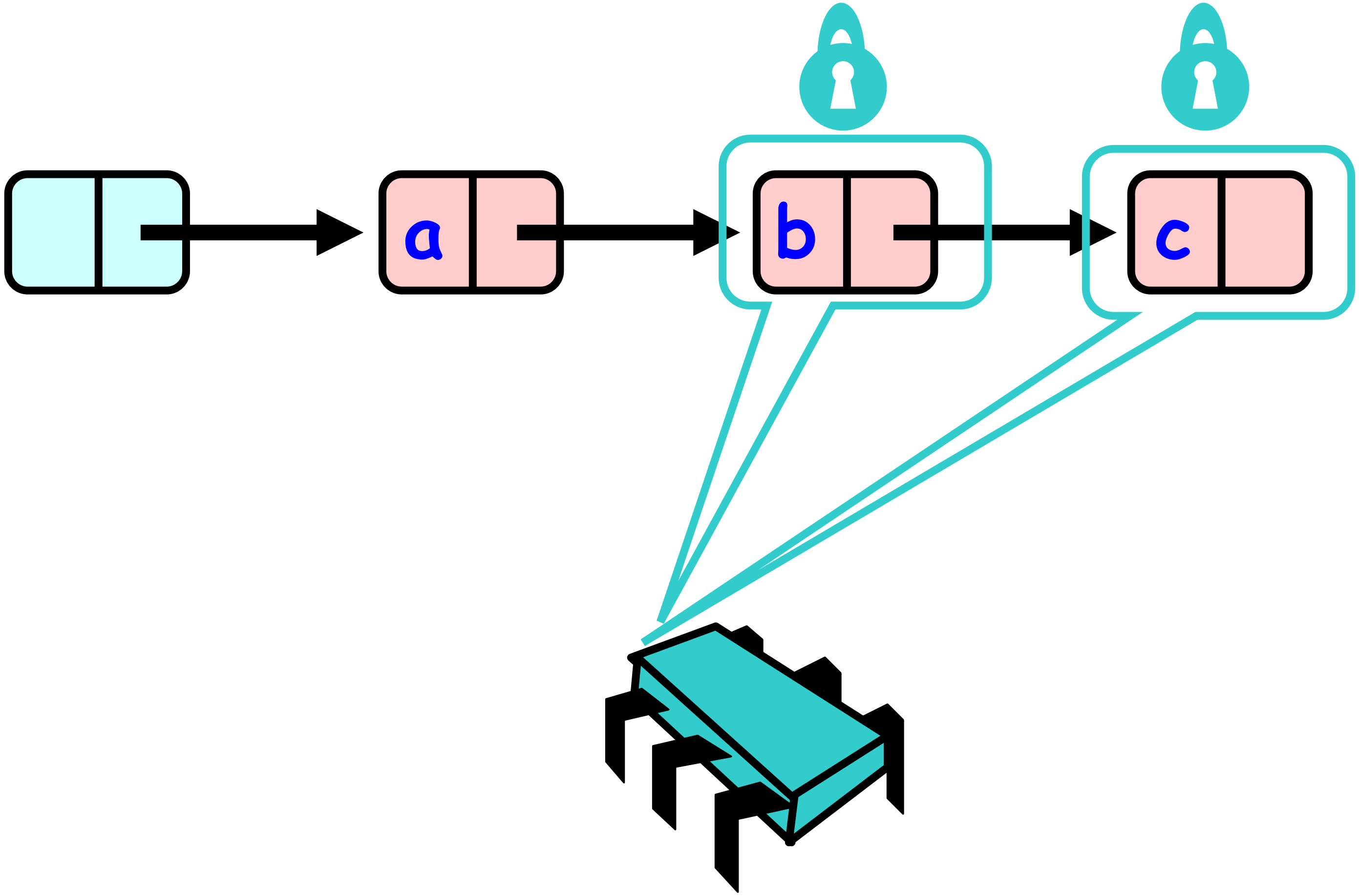
Hand-over-Hand locking



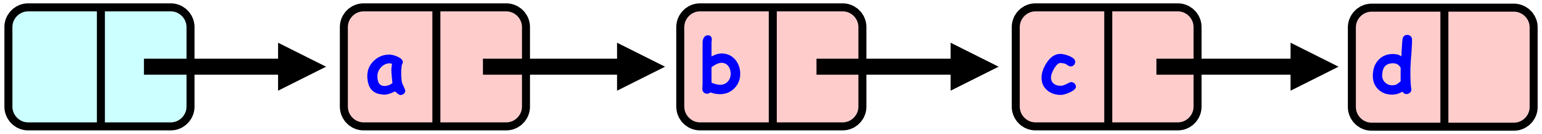
Hand-over-Hand locking

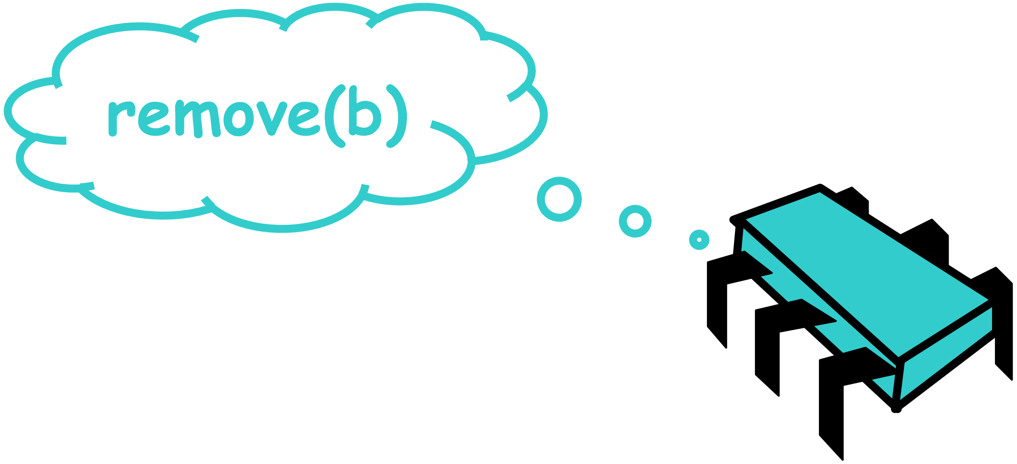


Hand-over-Hand locking

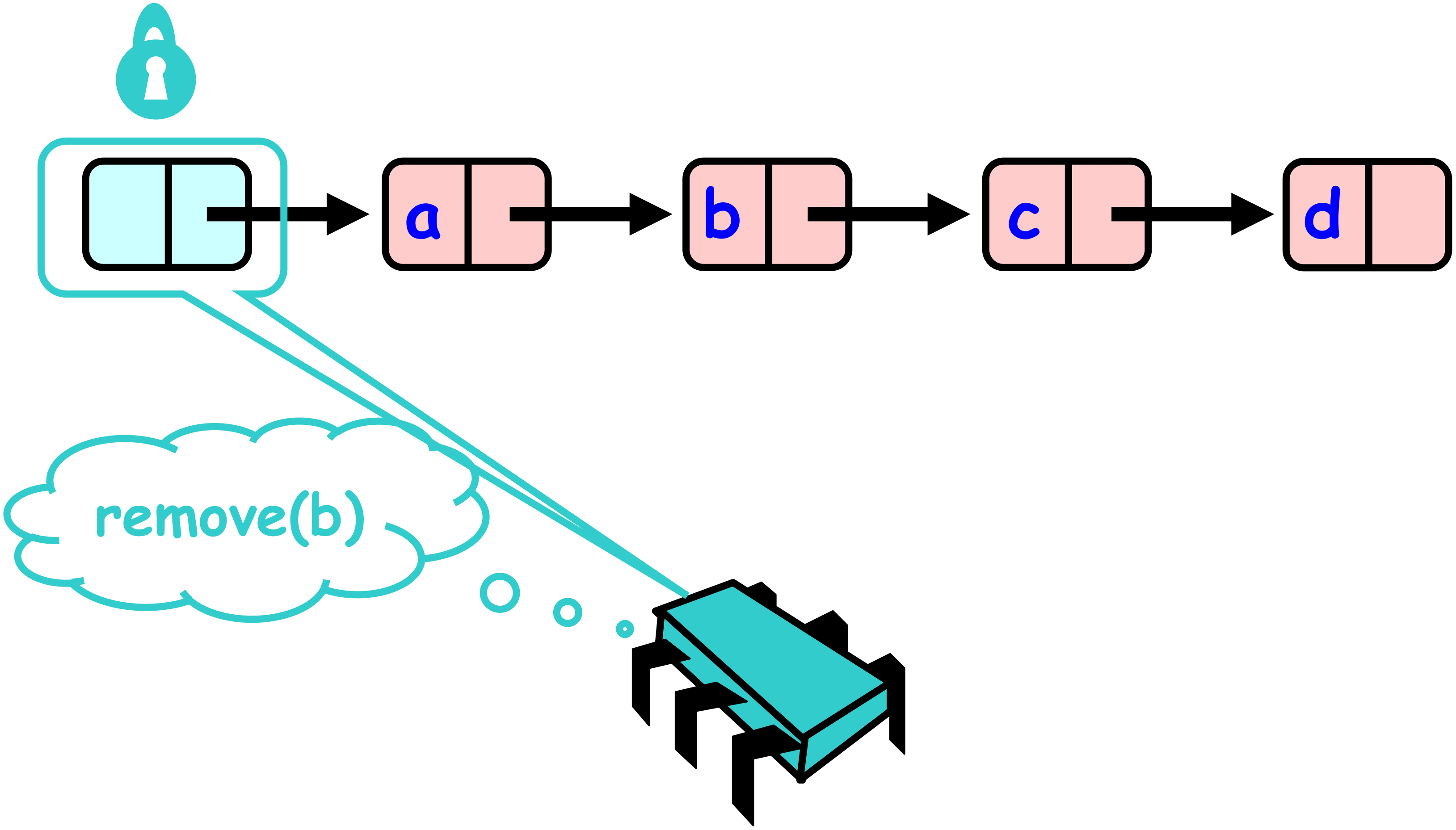


Removing a Node

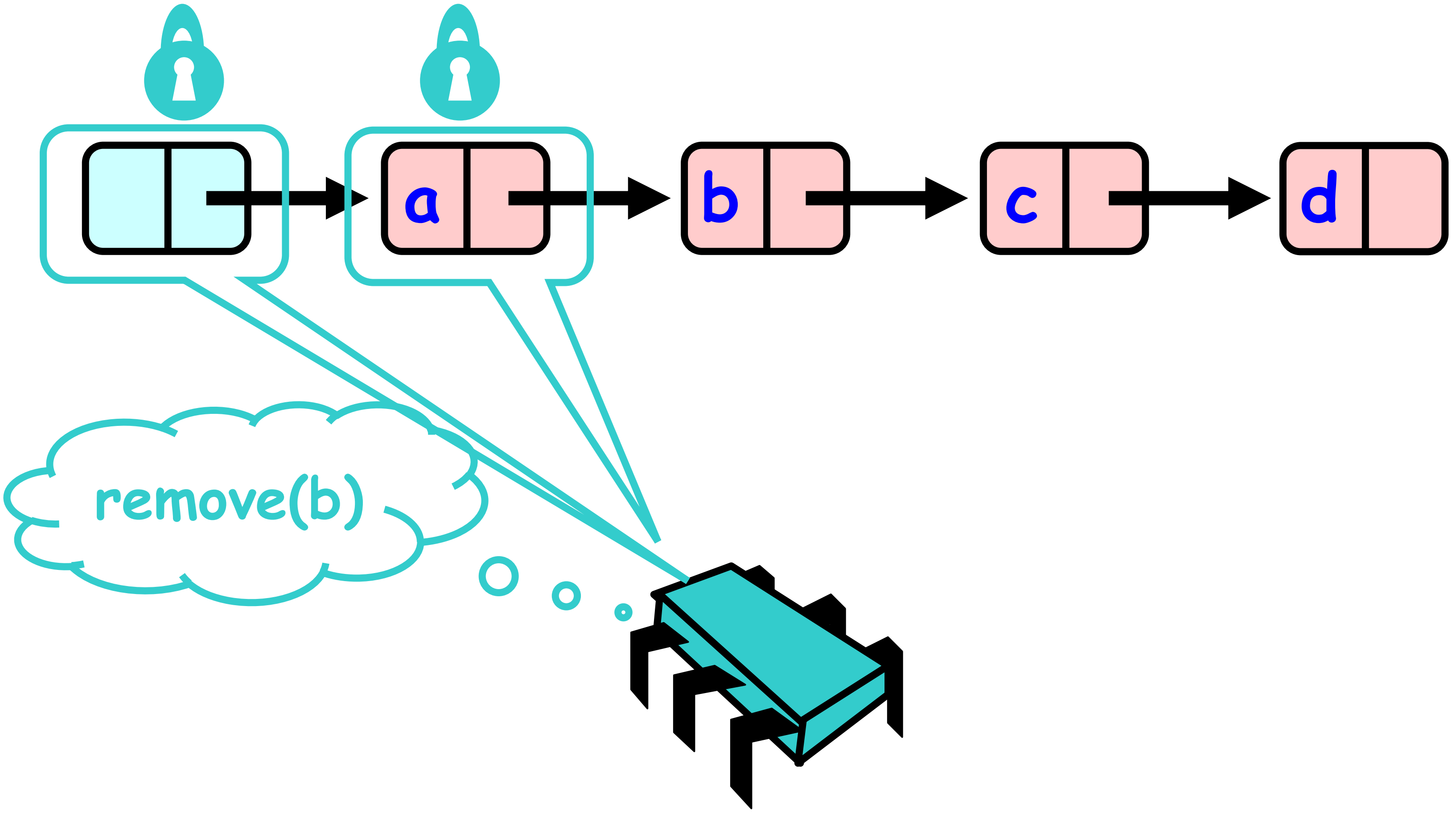




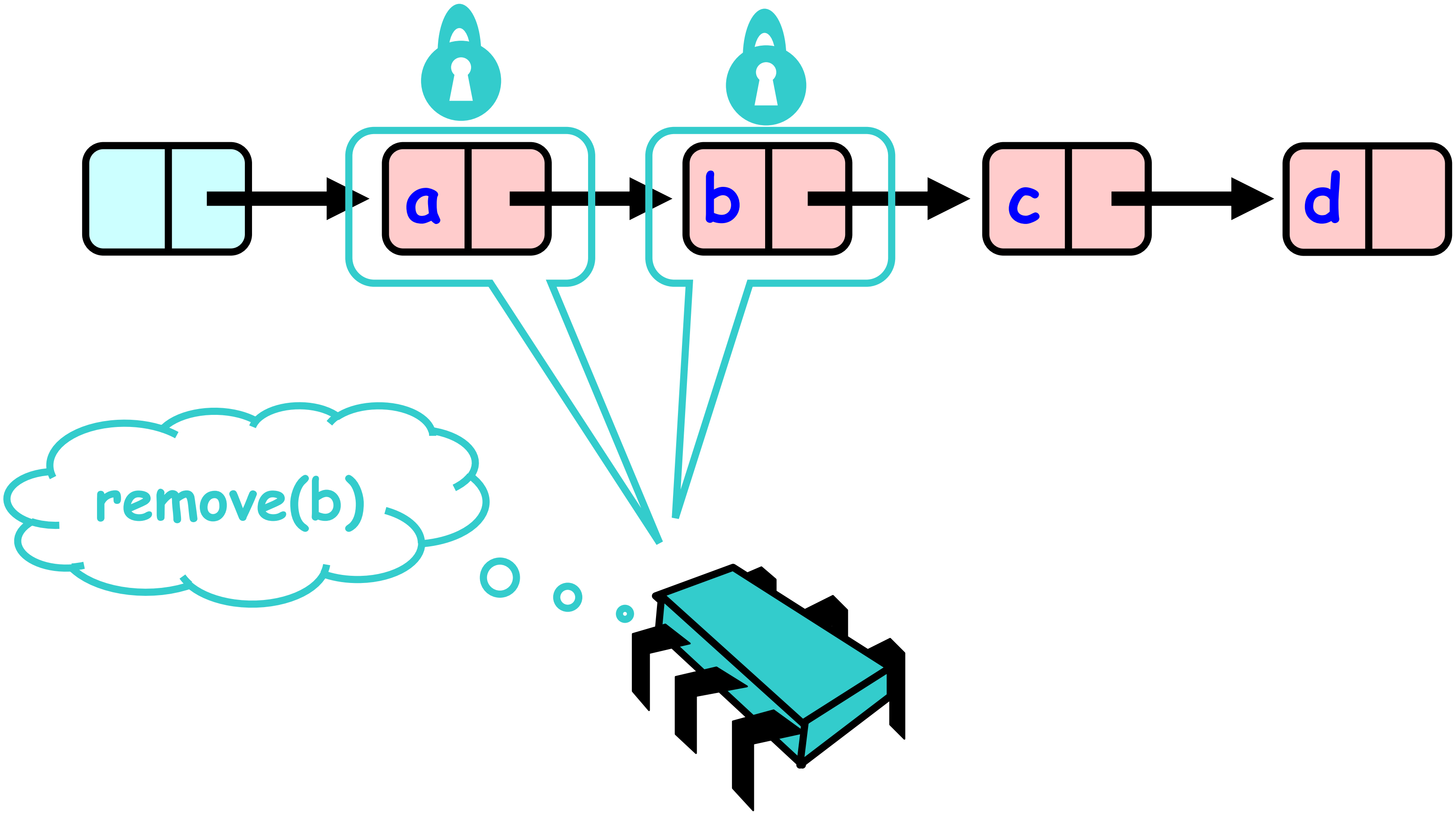
Removing a Node



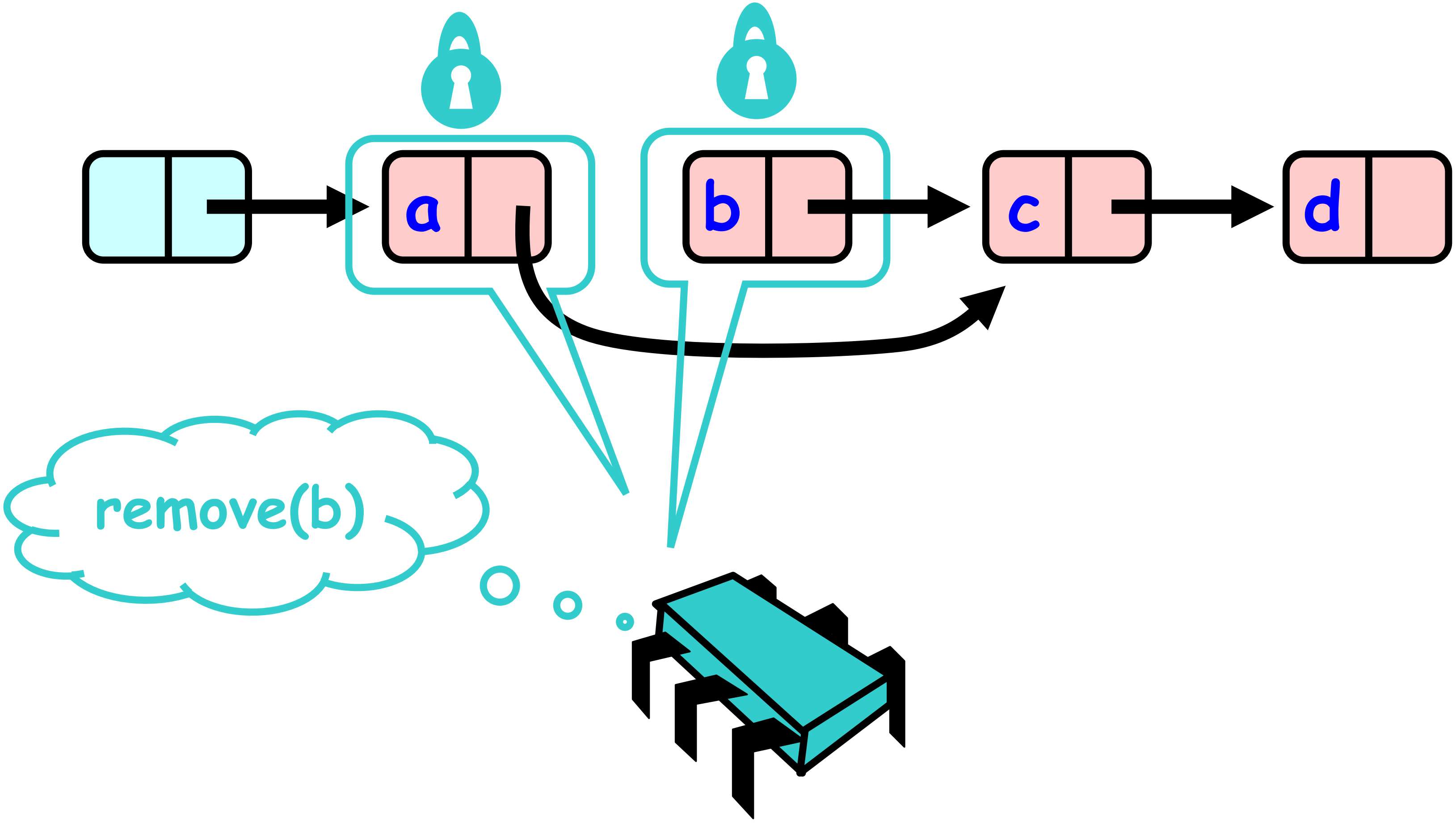
Removing a Node



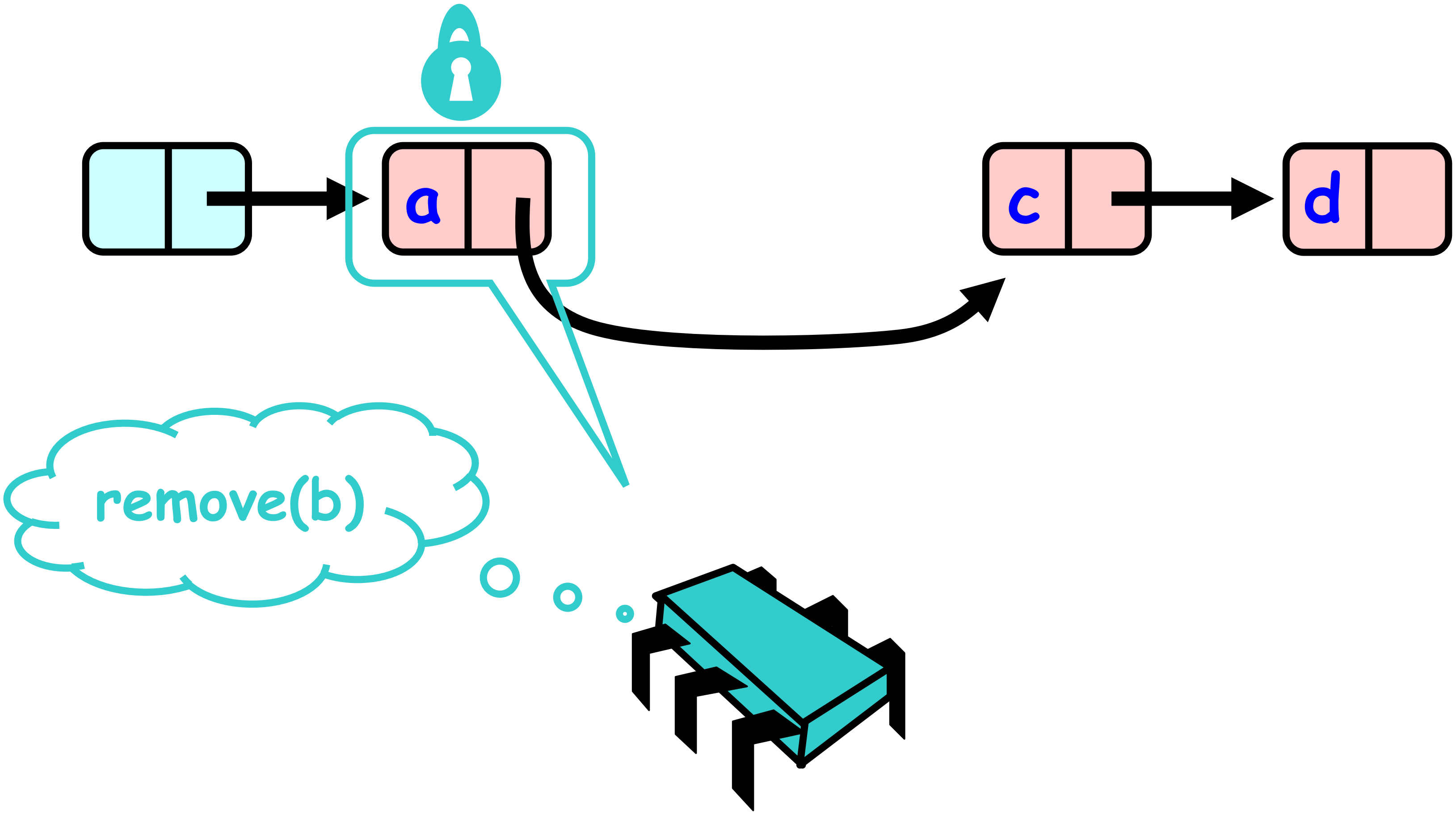
Removing a Node



Removing a Node



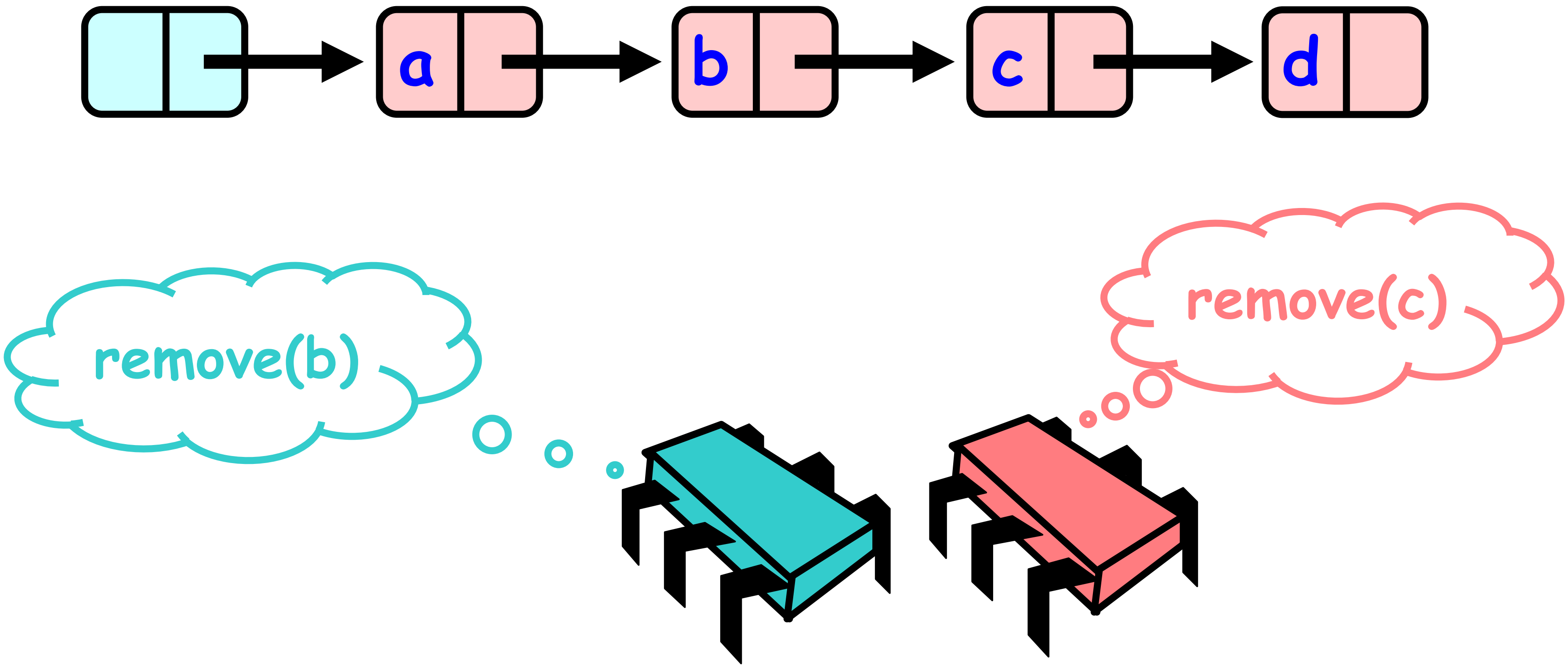
Removing a Node



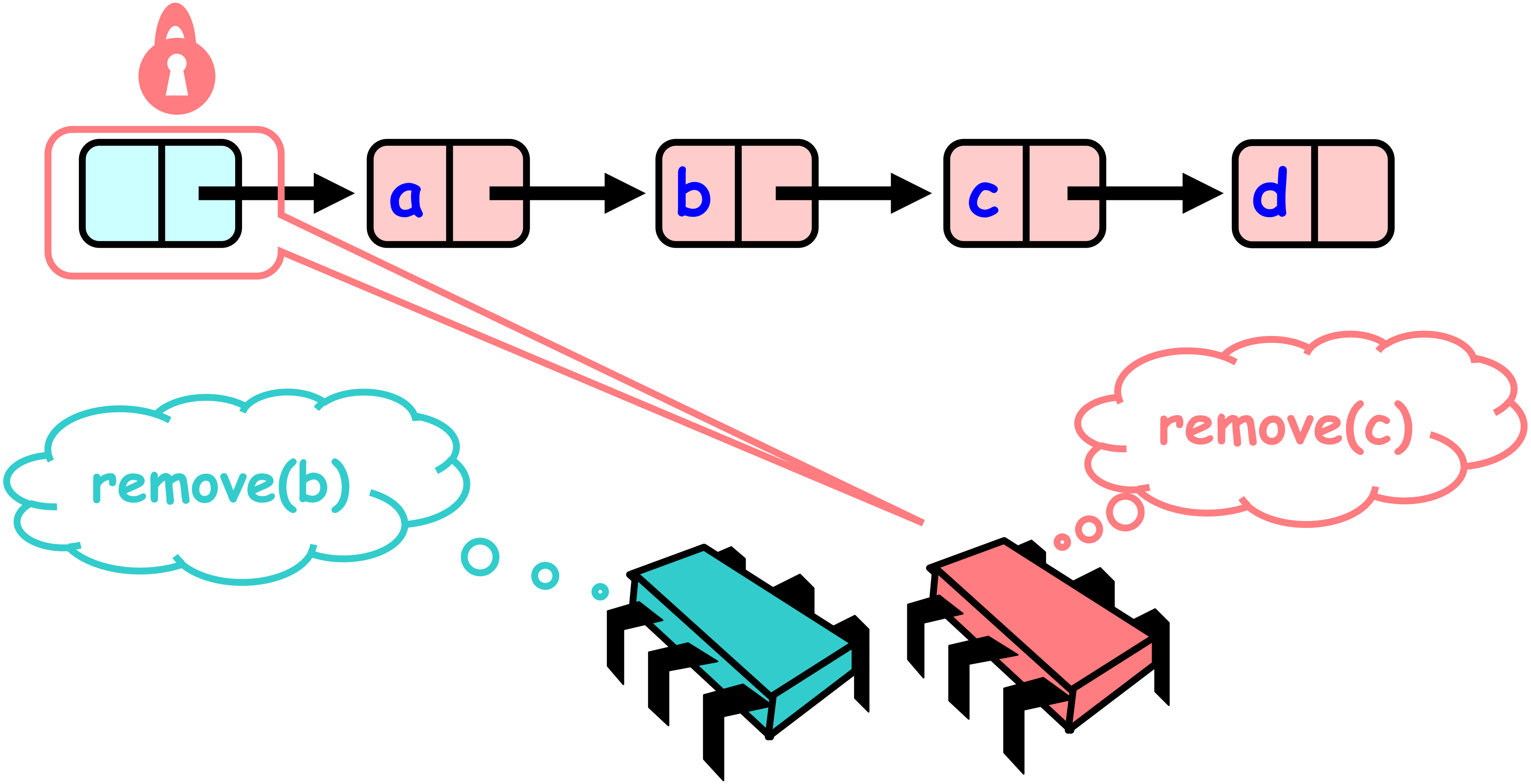
Hand-over-hand

- Does it solve our problem?

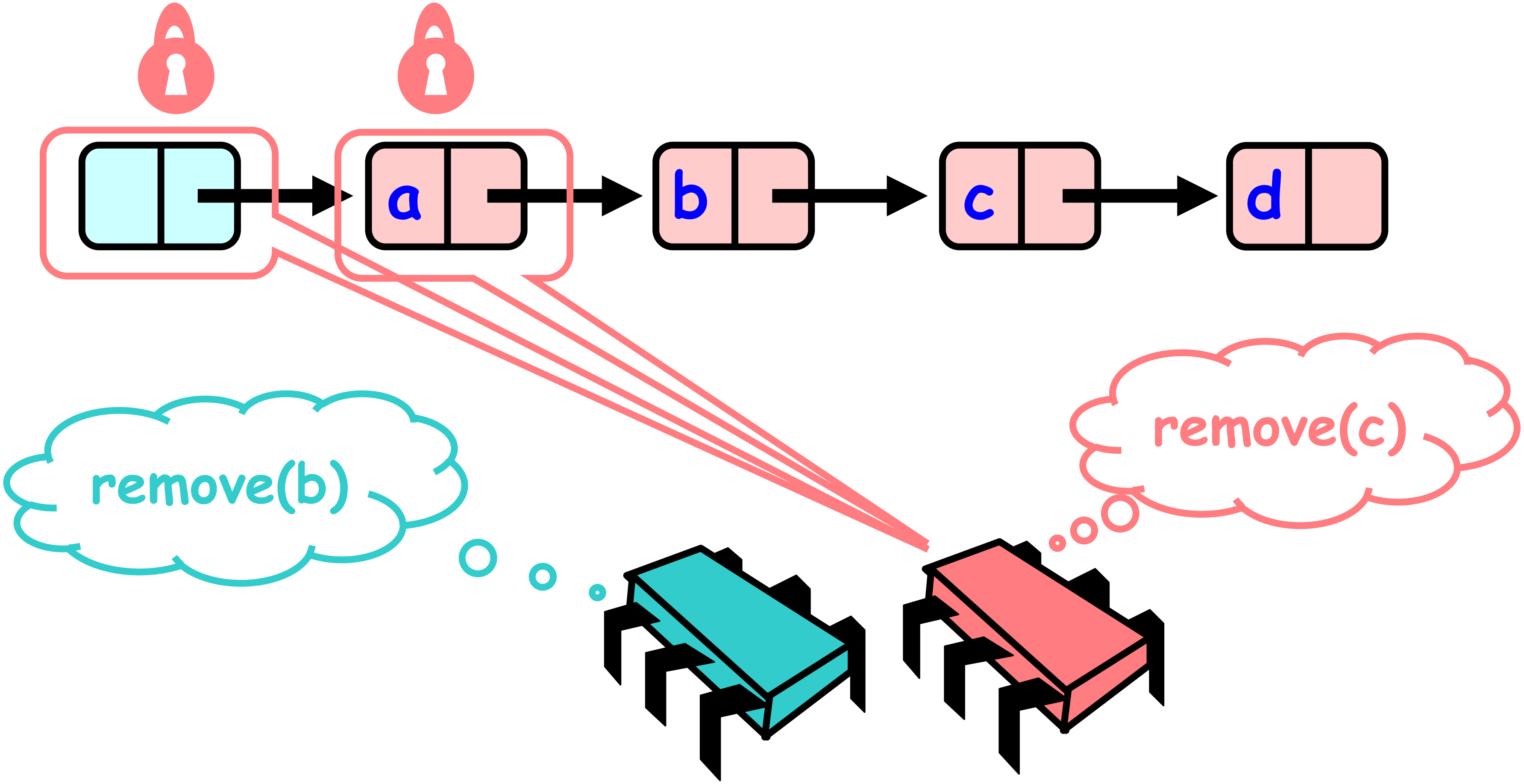
Removing a Node



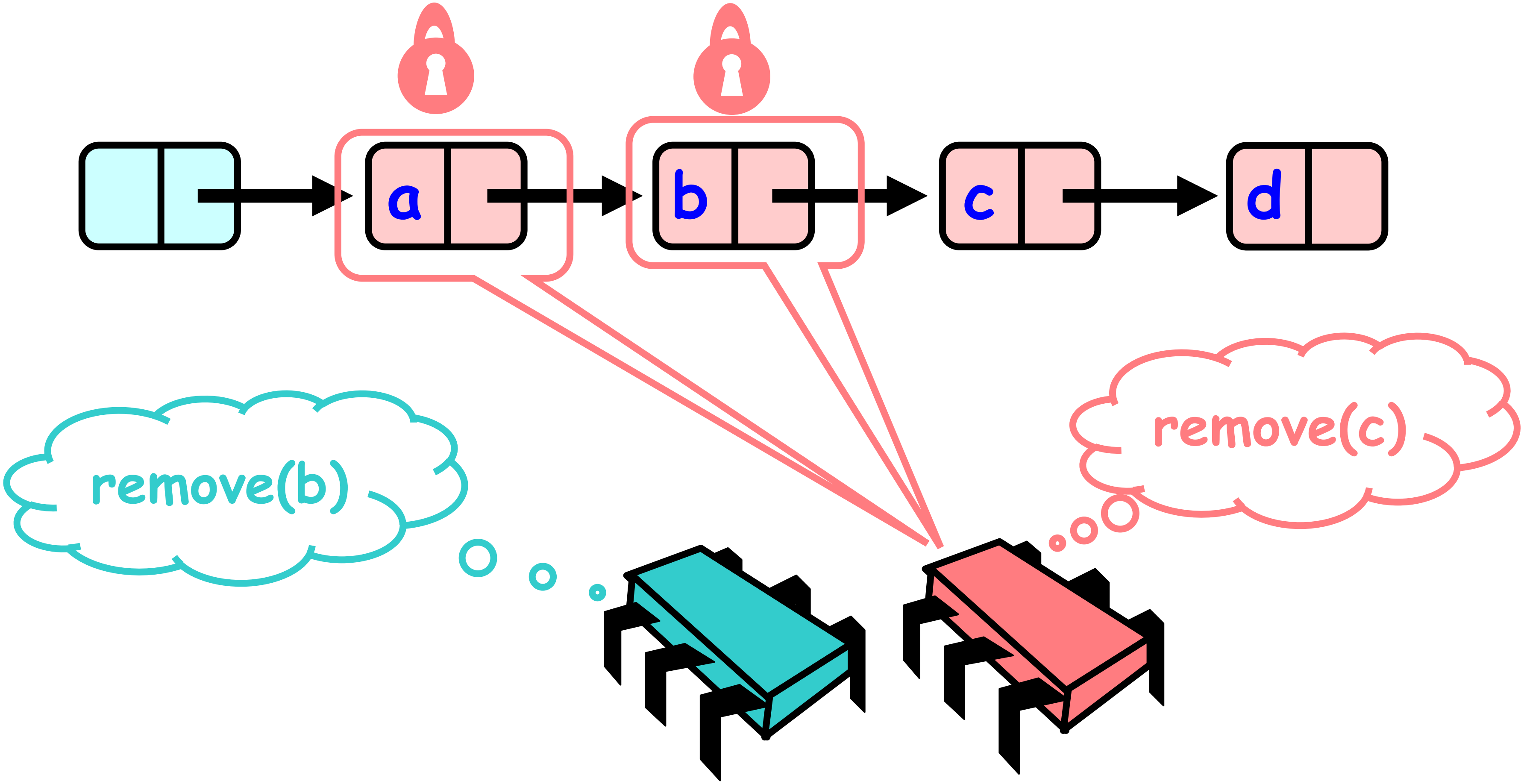
Removing a Node



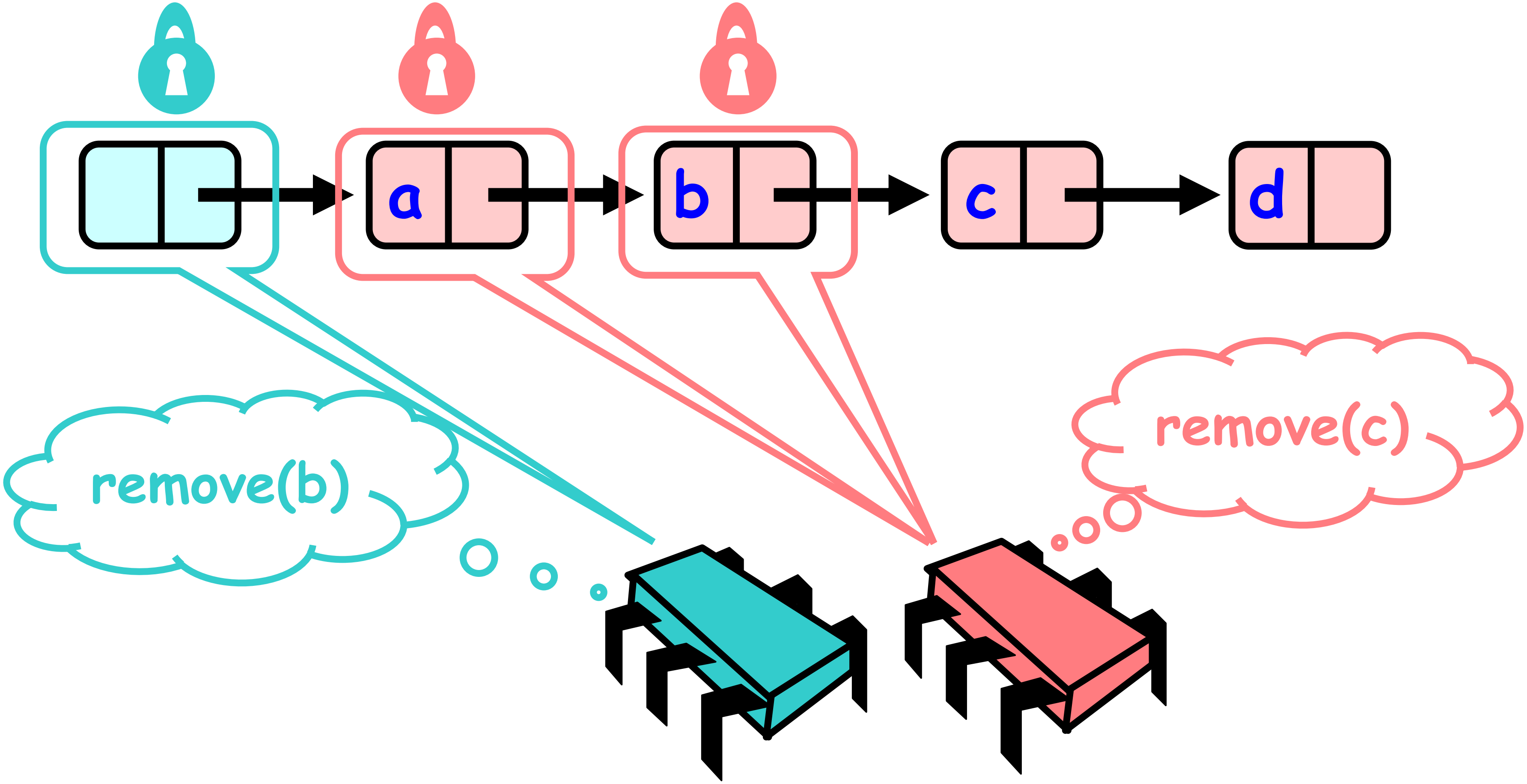
Removing a Node



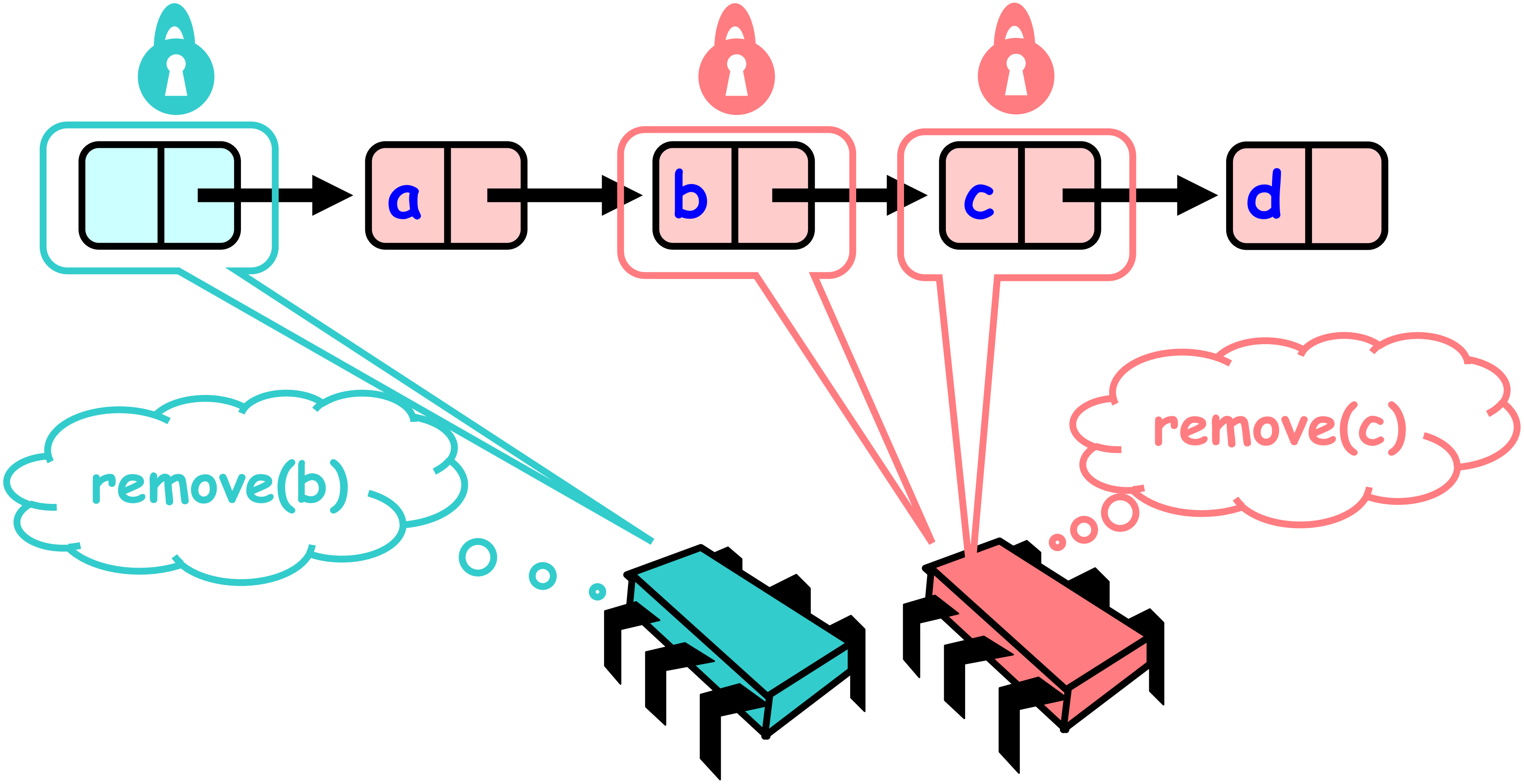
Removing a Node



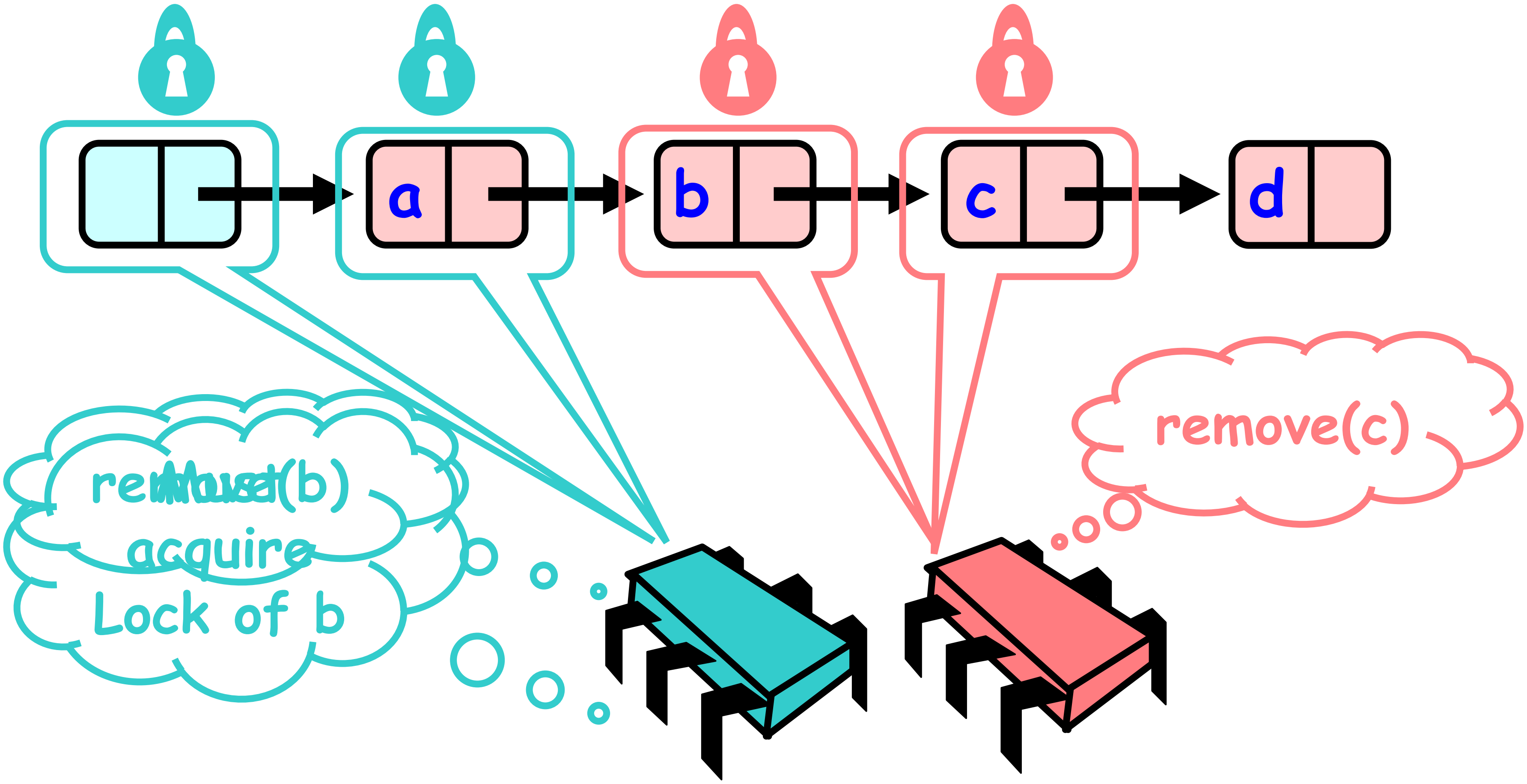
Removing a Node



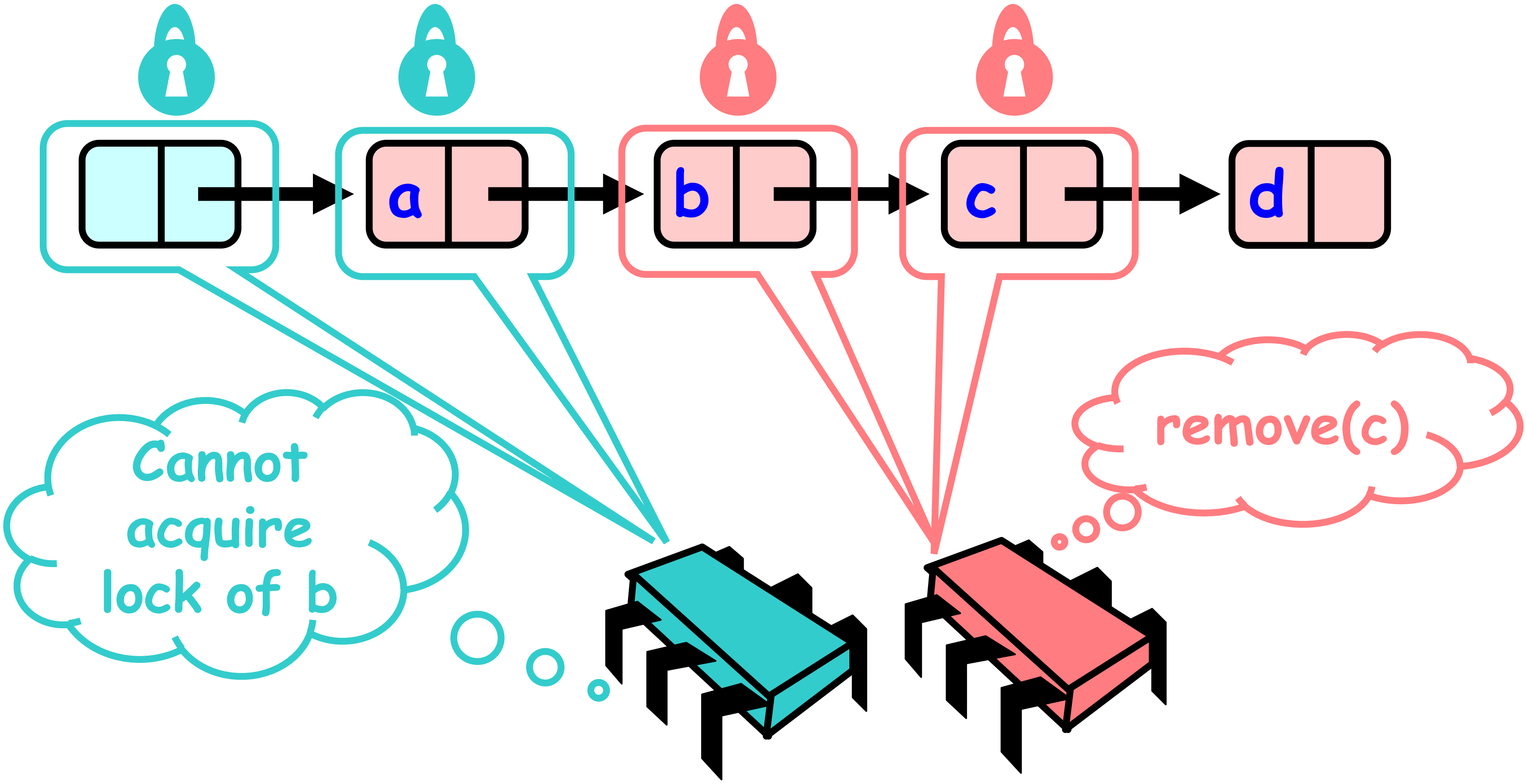
Removing a Node



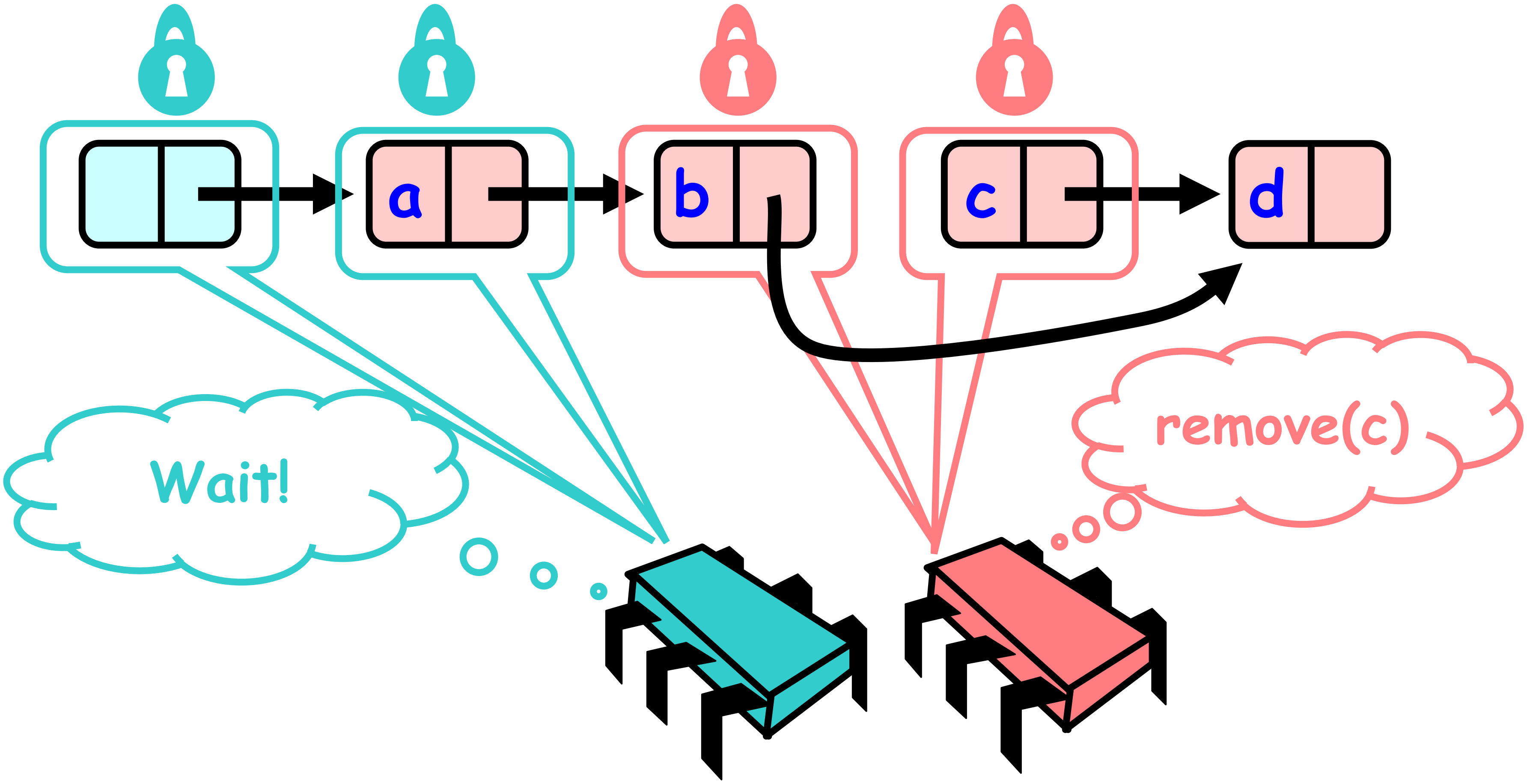
Removing a Node



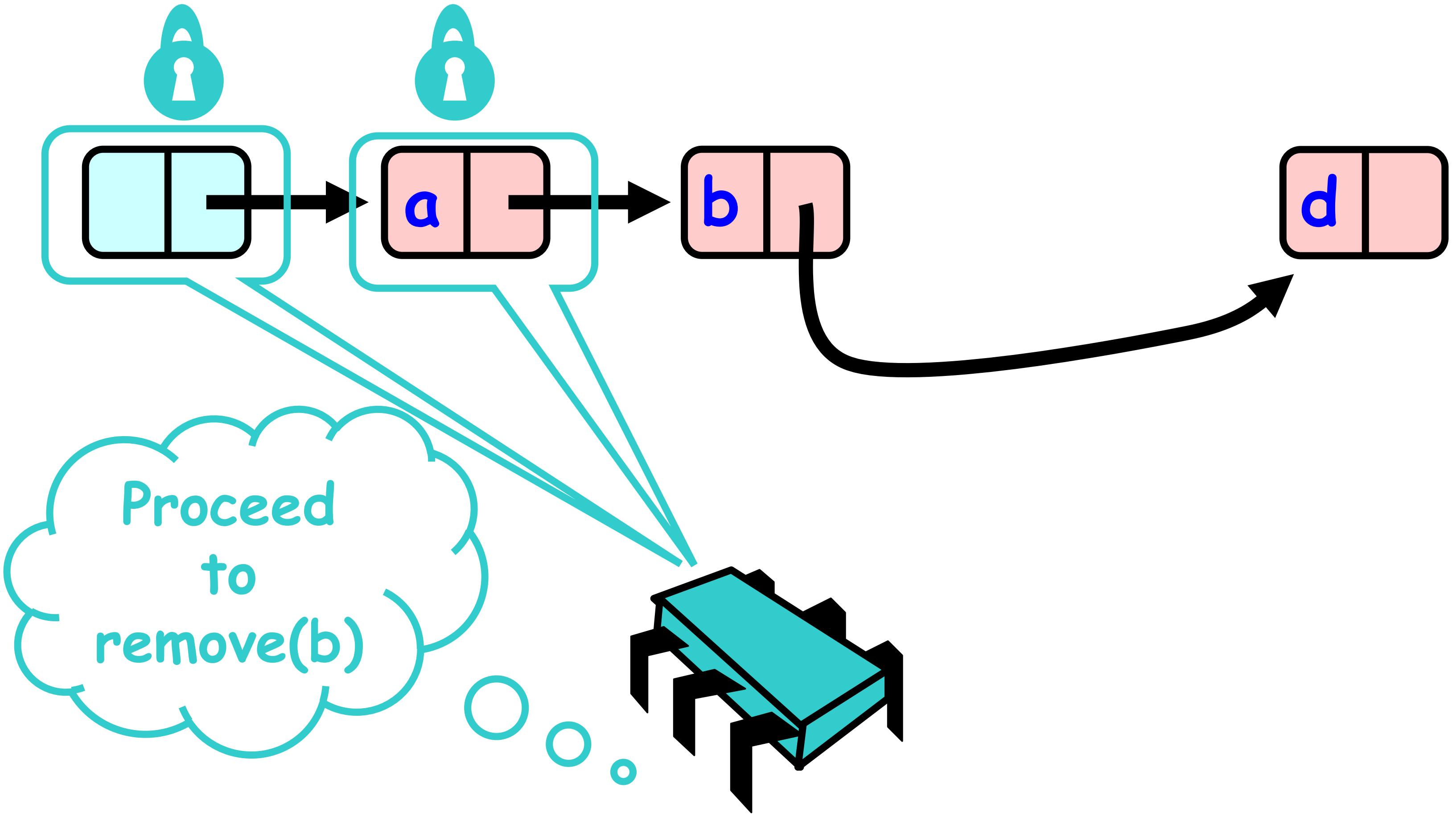
Removing a Node



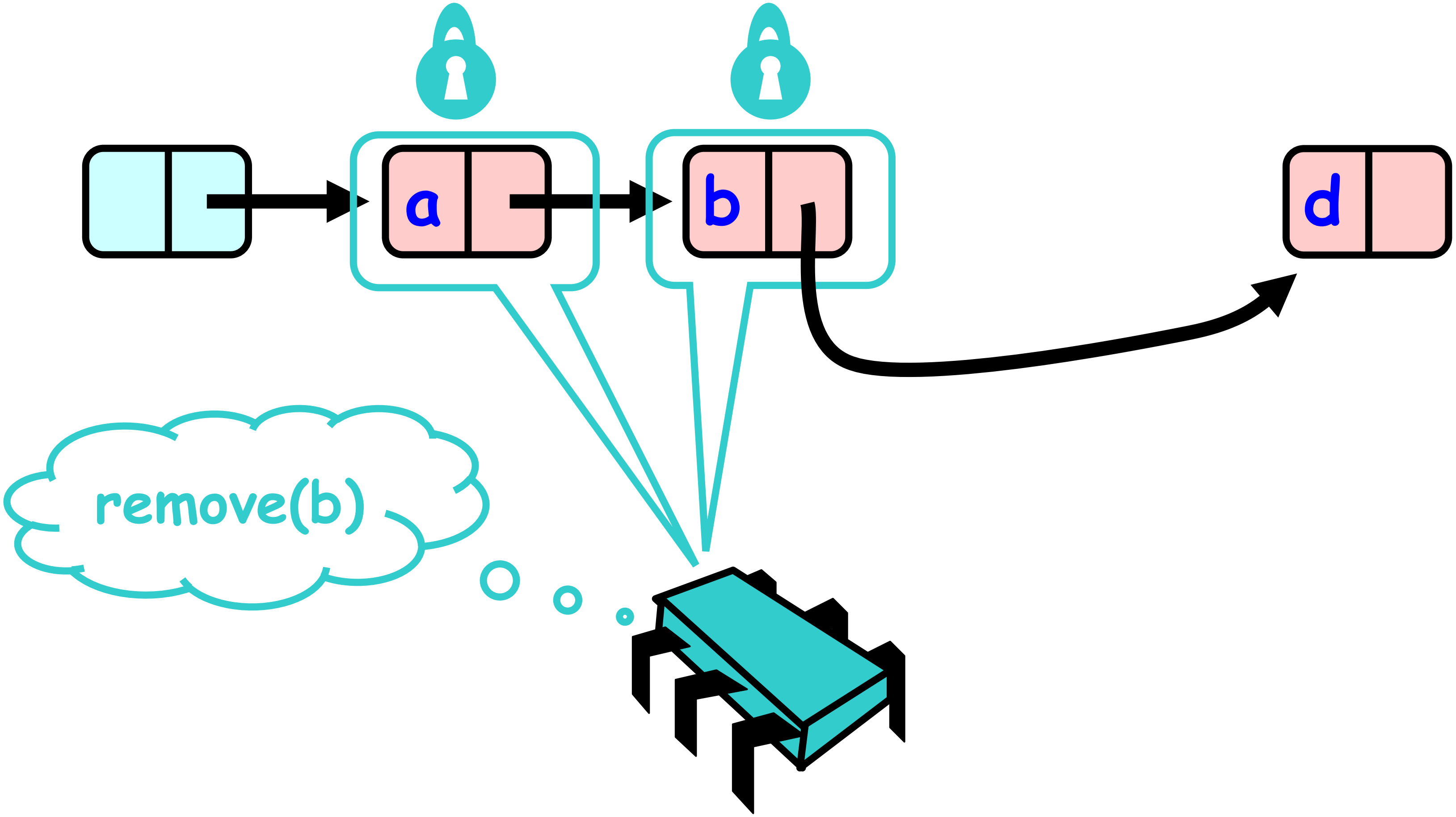
Removing a Node



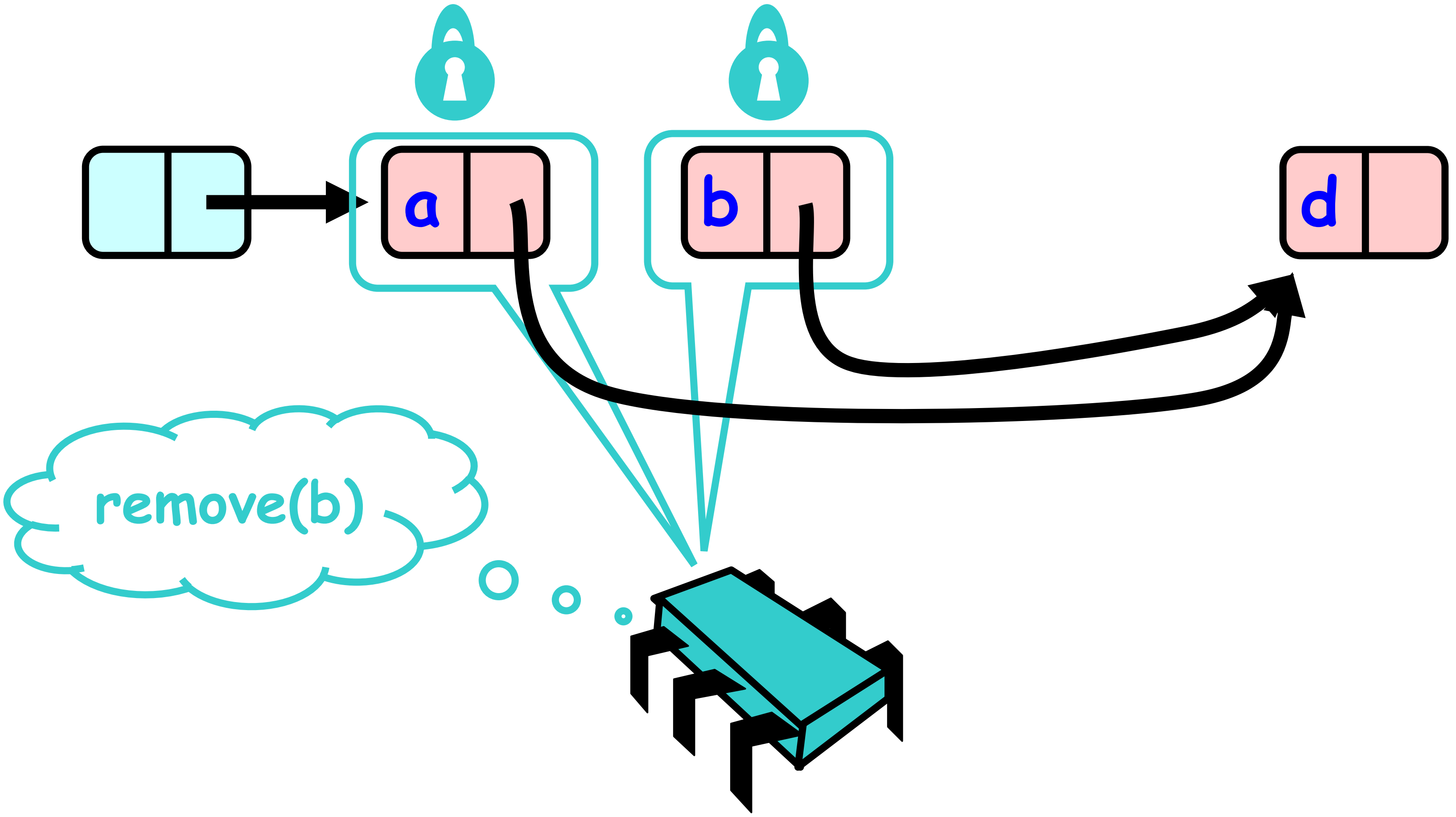
Removing a Node



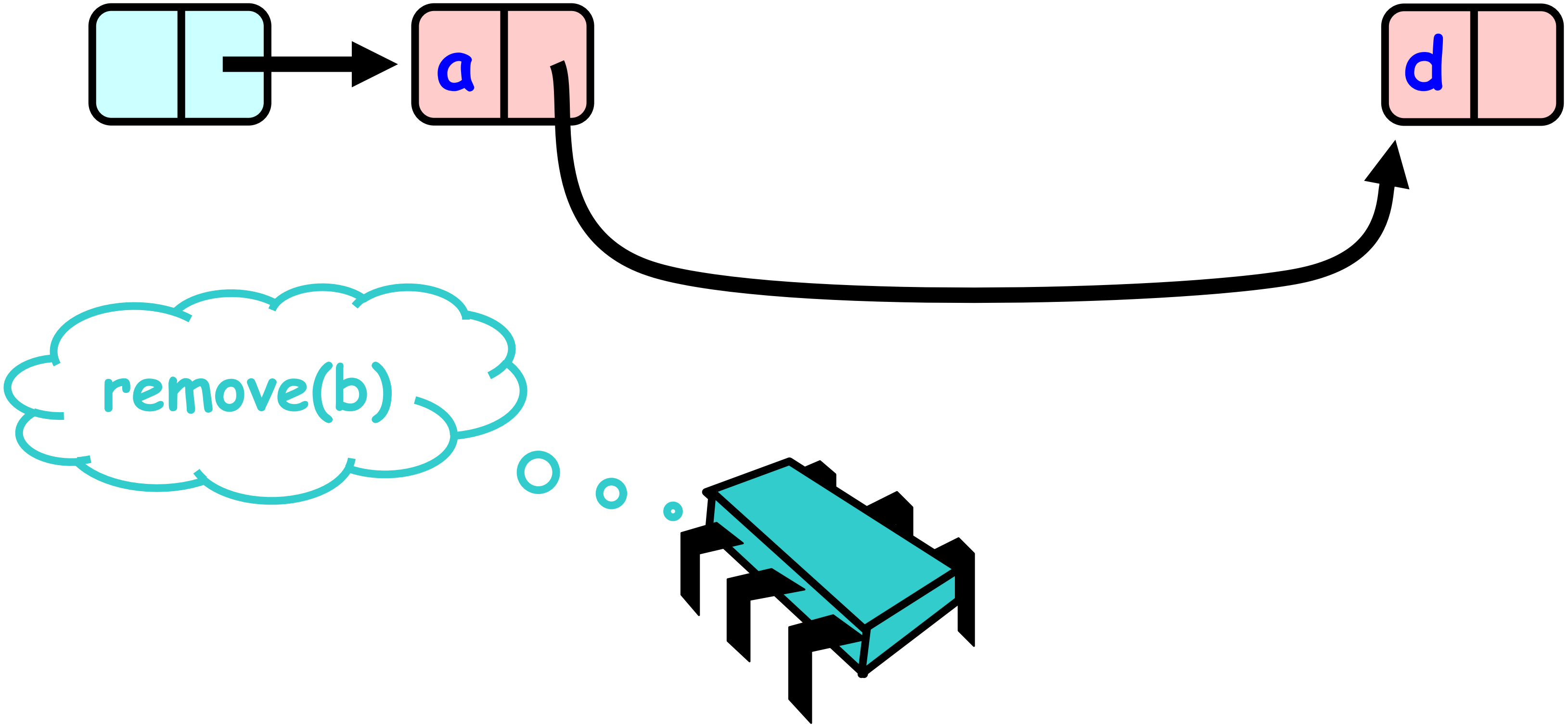
Removing a Node



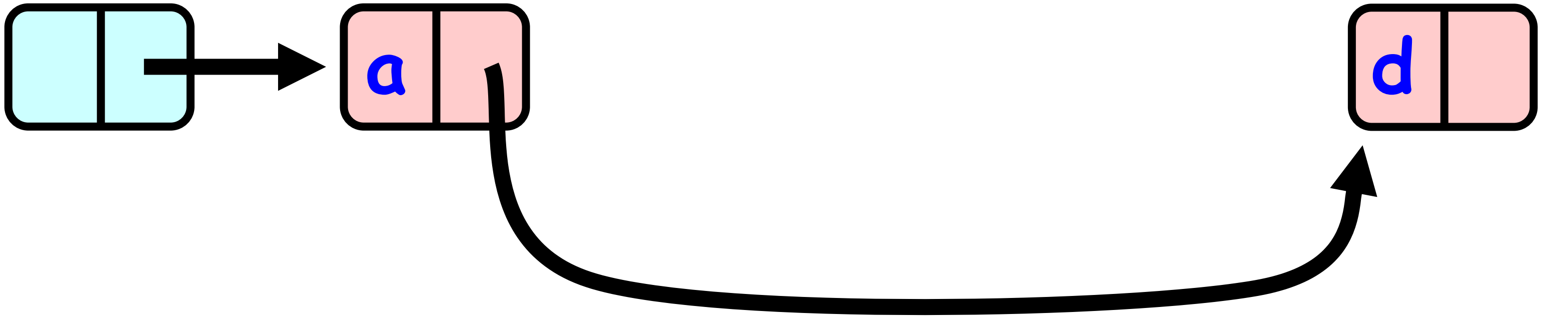
Removing a Node



Removing a Node



Removing a Node



Remove method

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

Remove method

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

Key used to order node

Remove method

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

Predecessor and current nodes

Remove method

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

Make sure locks released

Remove method

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

Everything else

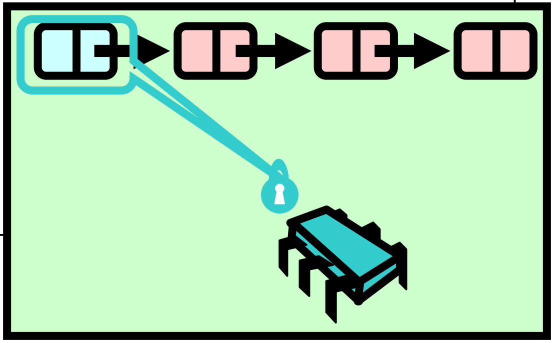
Remove method

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

Remove method

lock pred == head

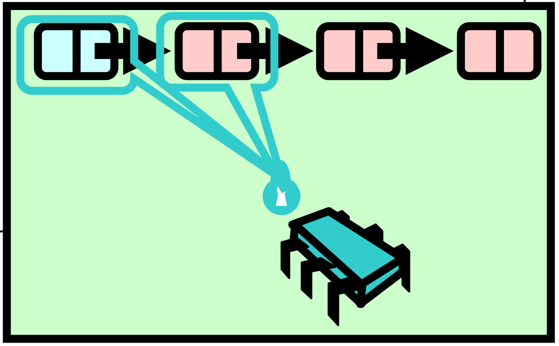
```
try {
  pred = head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```



Remove method

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

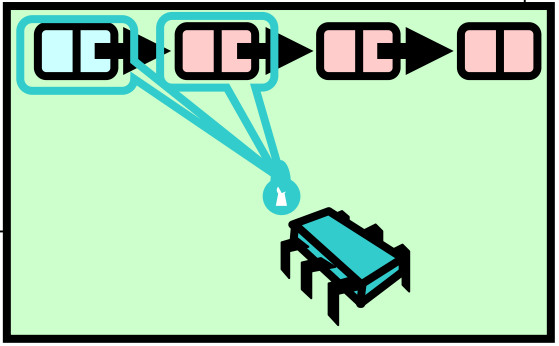
Lock current



Remove method

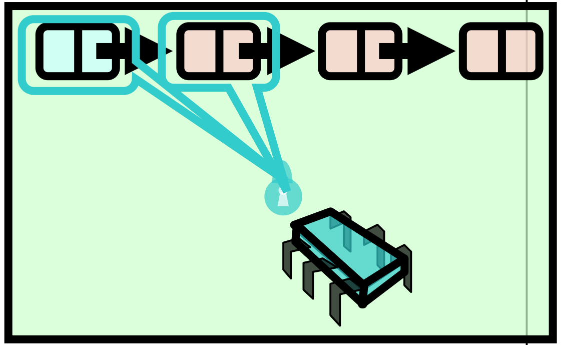
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

Traversing list



Remove: searching

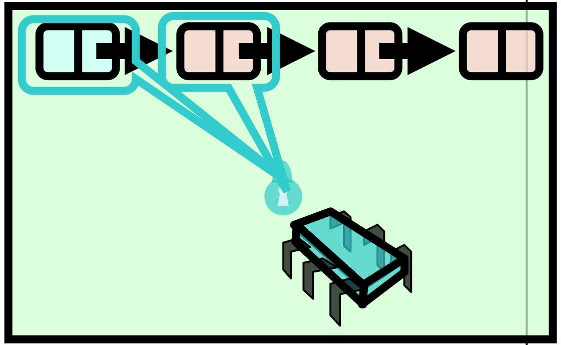
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

```
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;
```

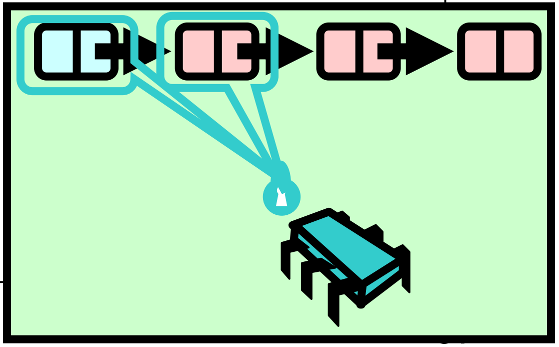
Search key range



Remove: searching

```
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;
```

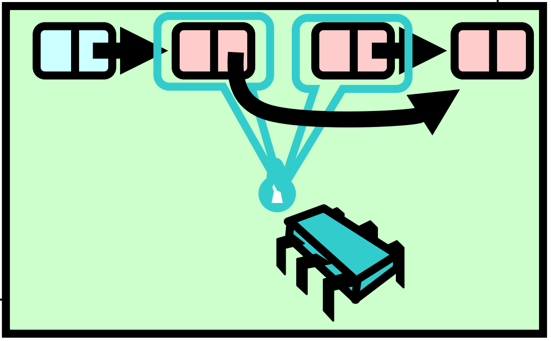
At start of each loop: curr and pred locked



Remove: searching

```
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;
```

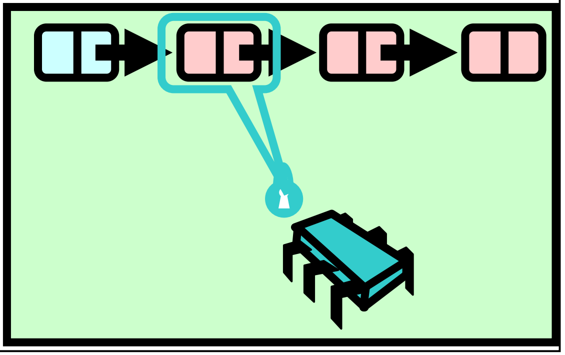
If item found, remove node



Remove: searching

Unlock predecessor

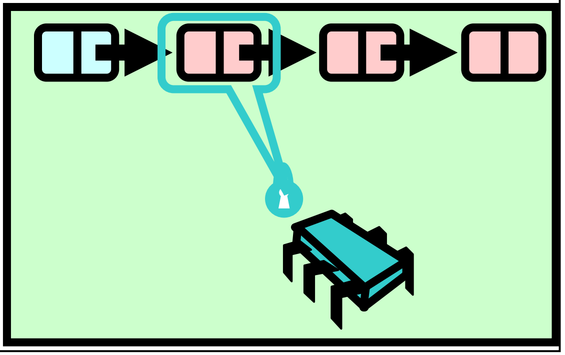
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

```
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;
```

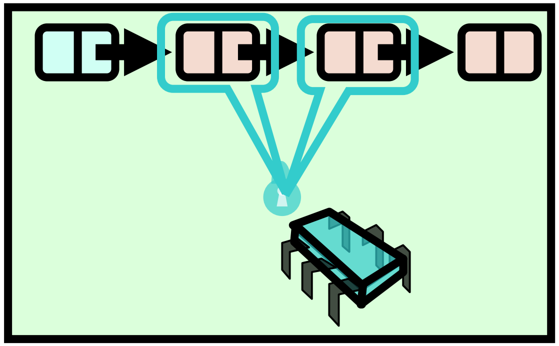
Promote predecessor



Remove: searching

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

Find and lock new current



Remove: searching

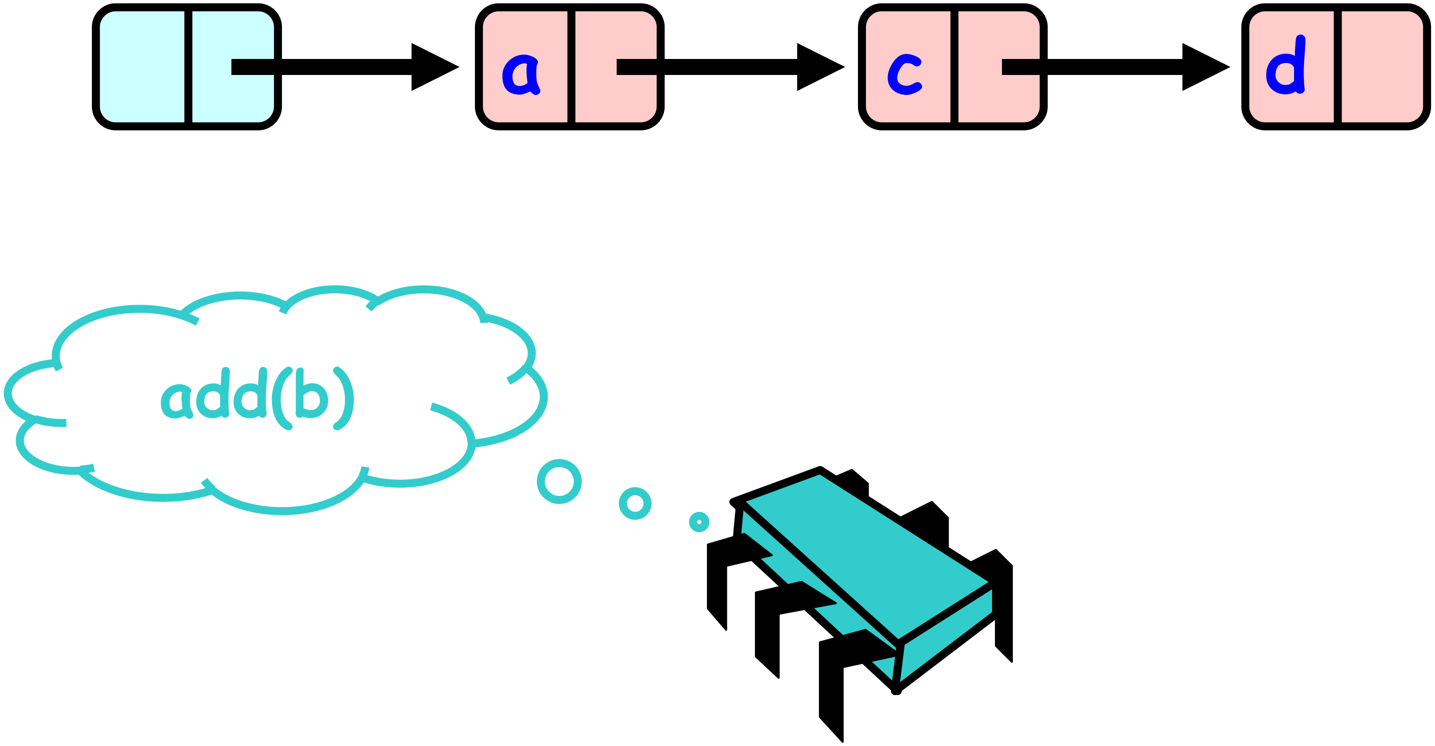
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, not present

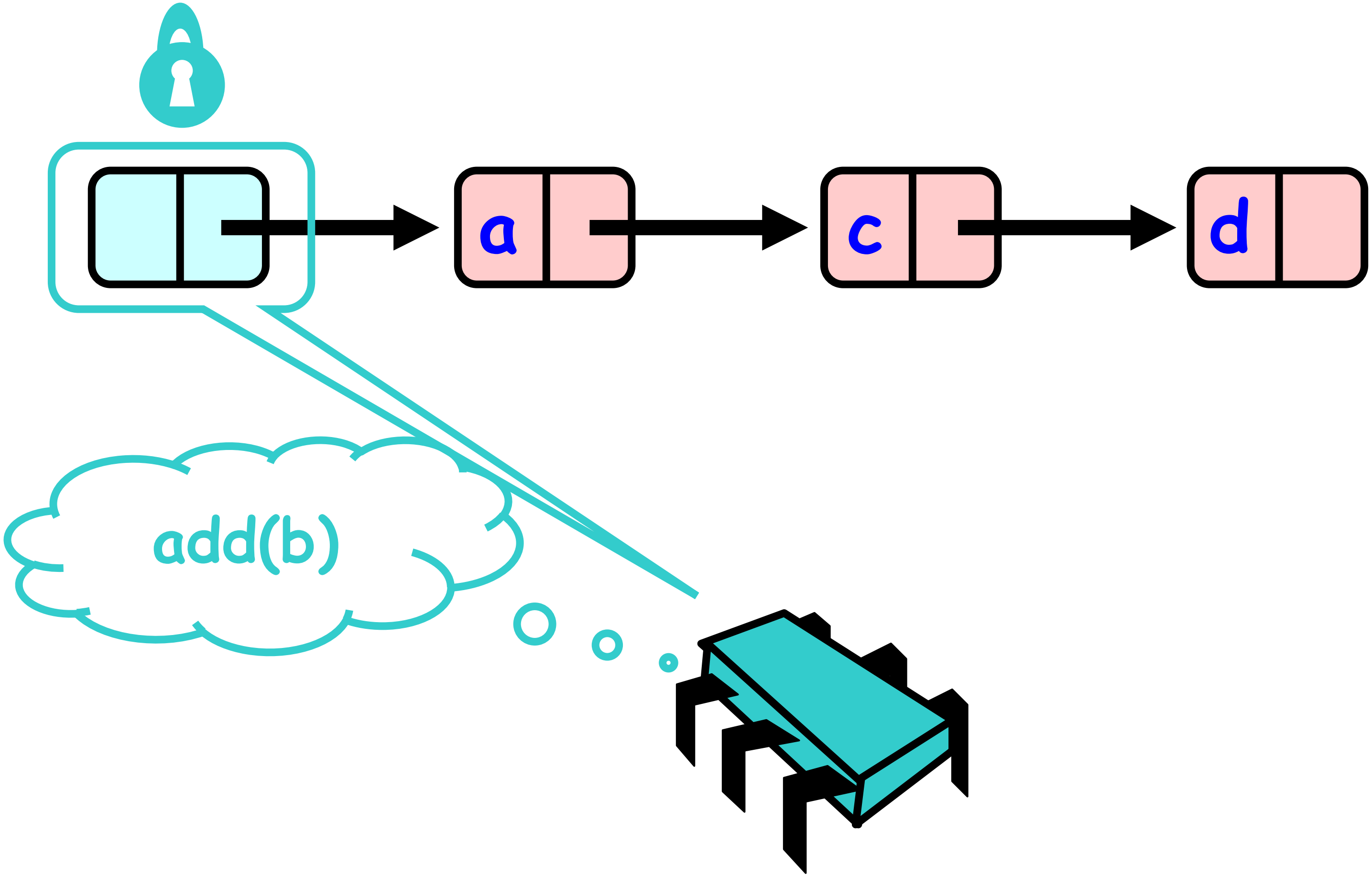
Why does this work?

- To remove node e
  - Must lock e
  - Must lock e’s predecessor
- Therefore, if you lock a node
  - It can’t be removed
  - And neither can its successor

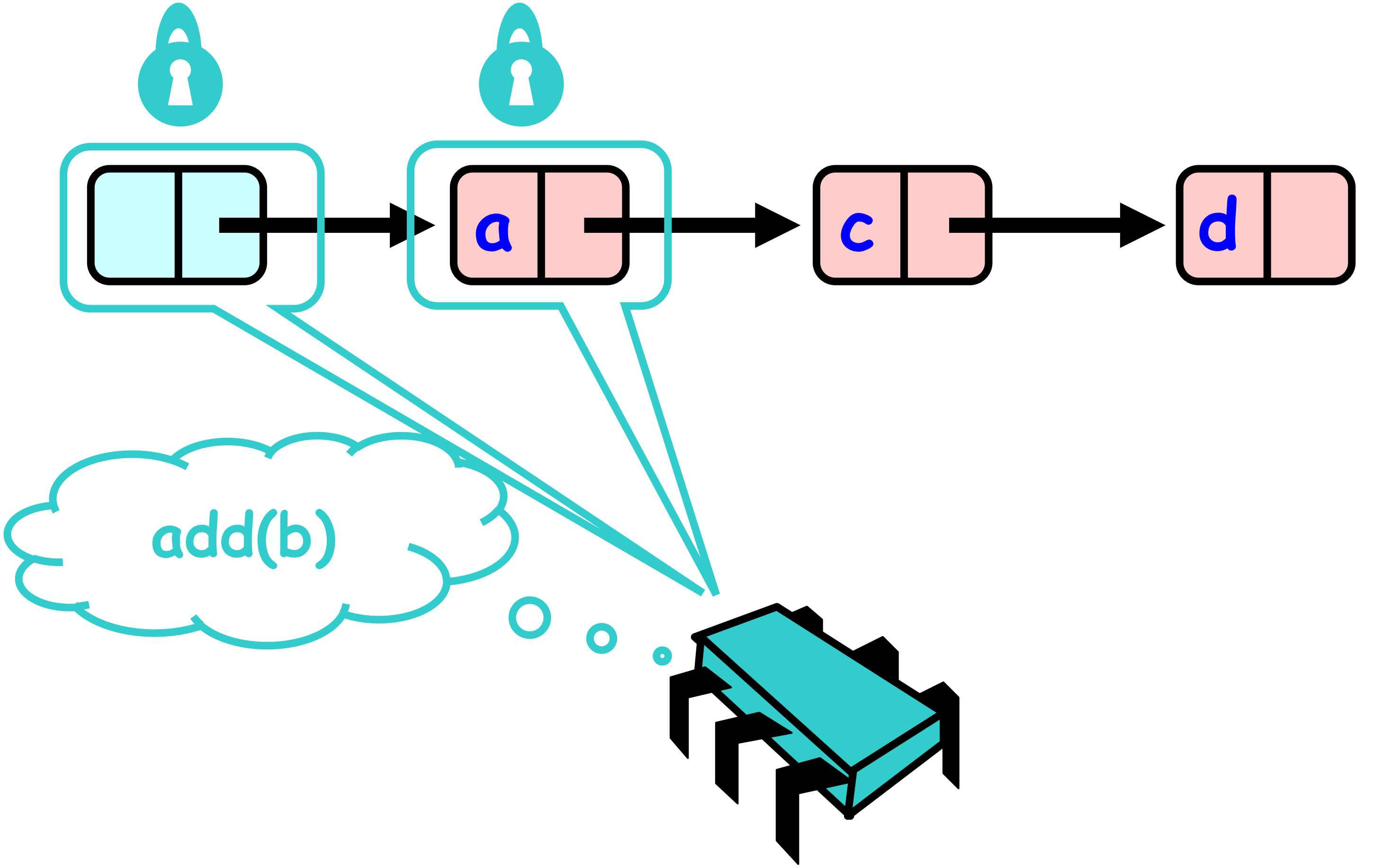
Adding a node



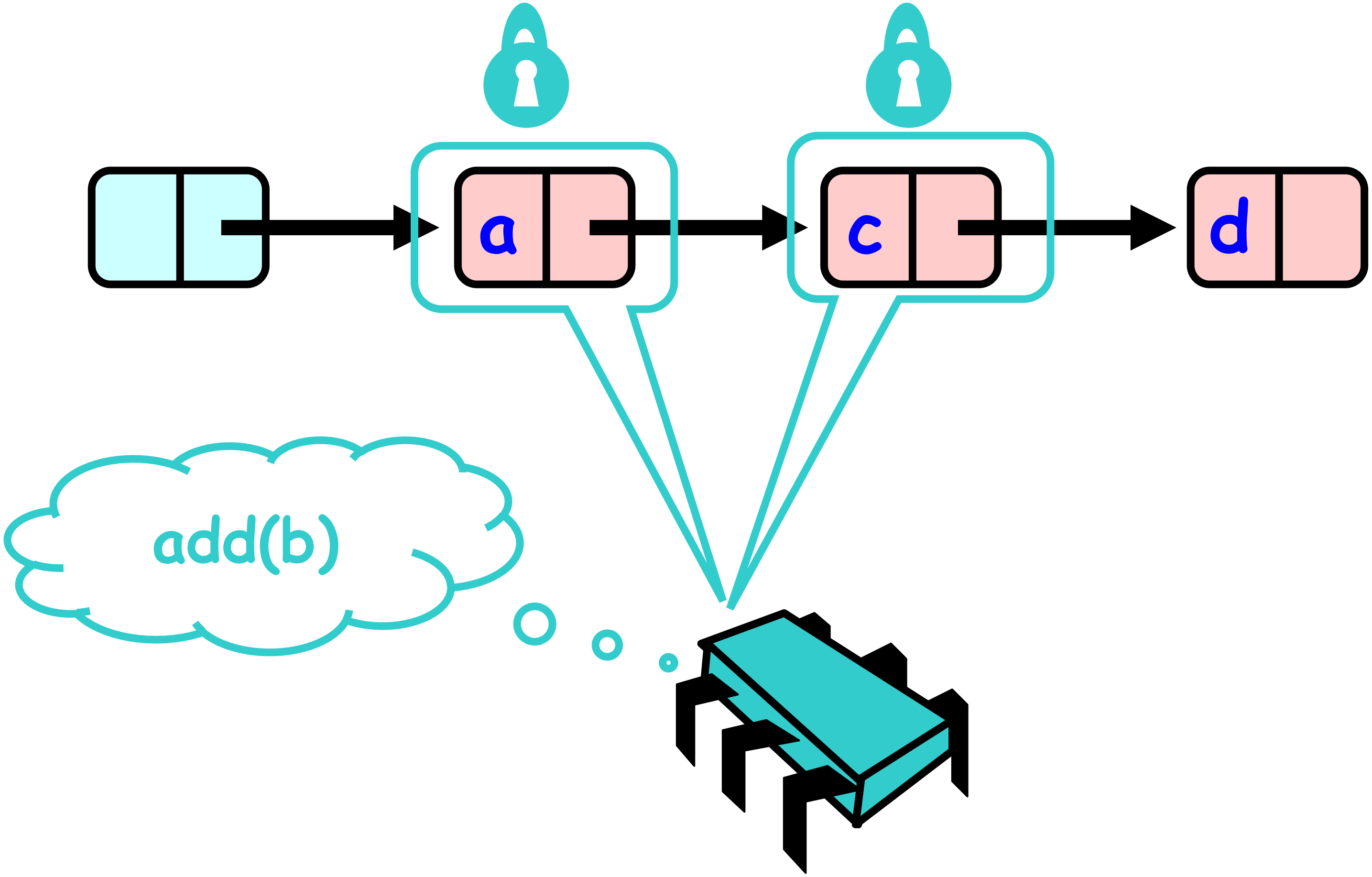
Adding a node



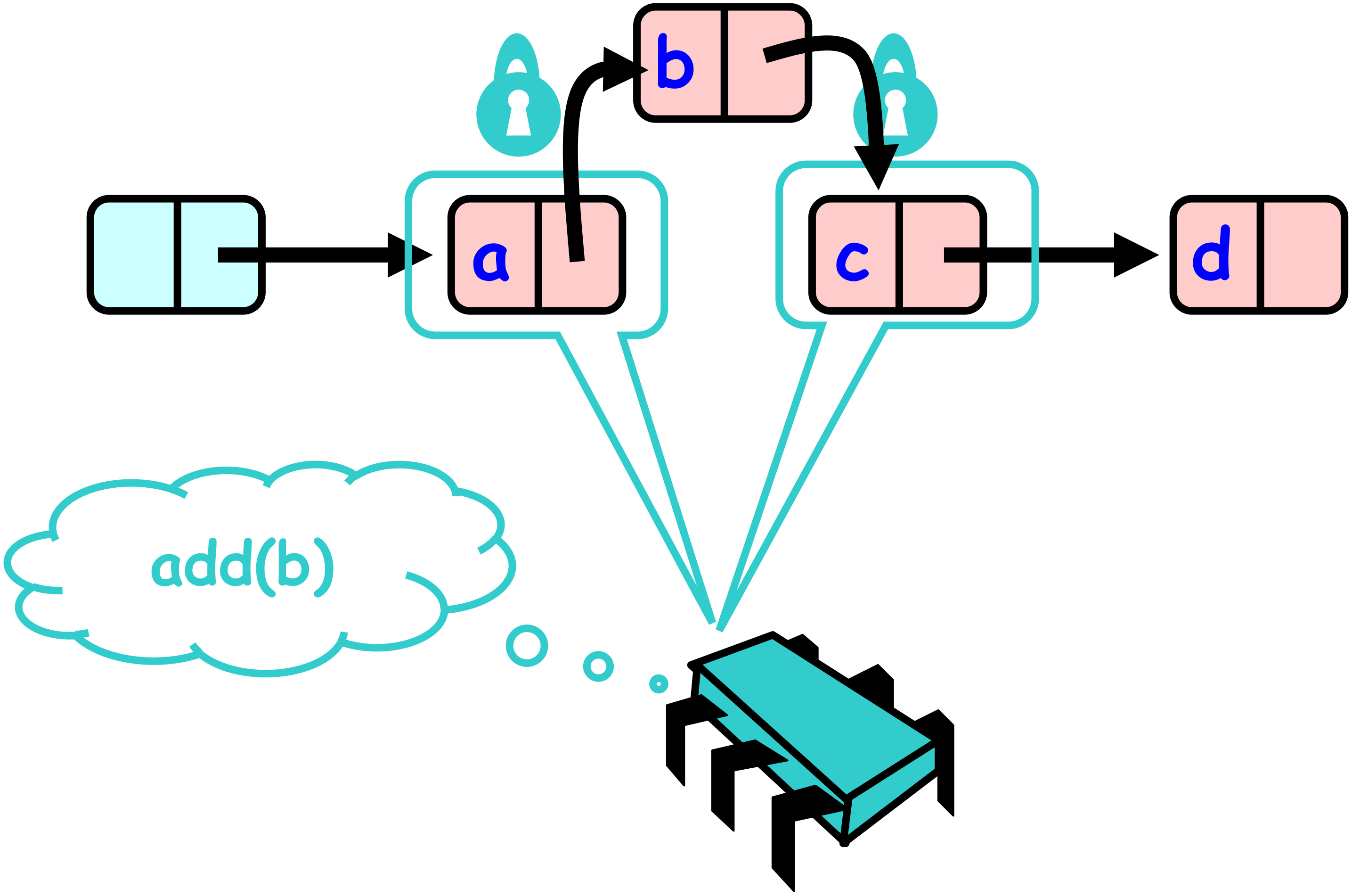
Adding a node



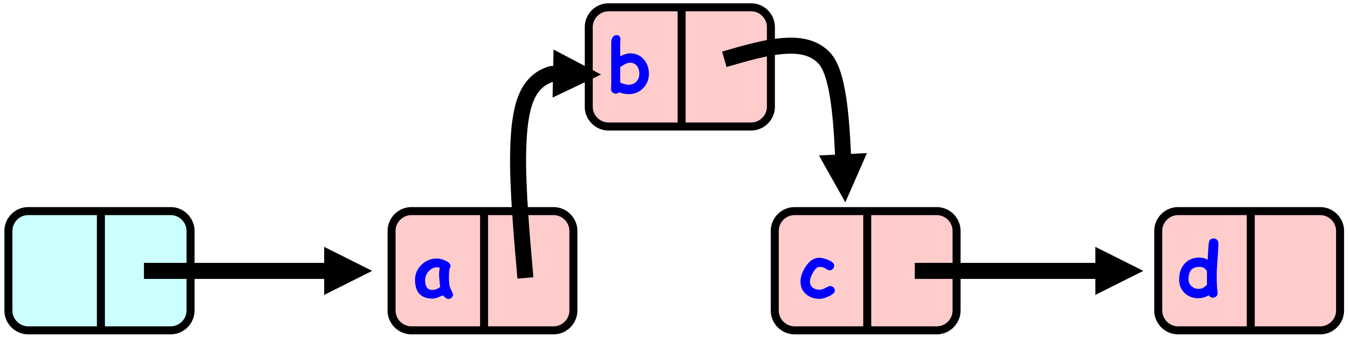
Adding a node



Adding a node



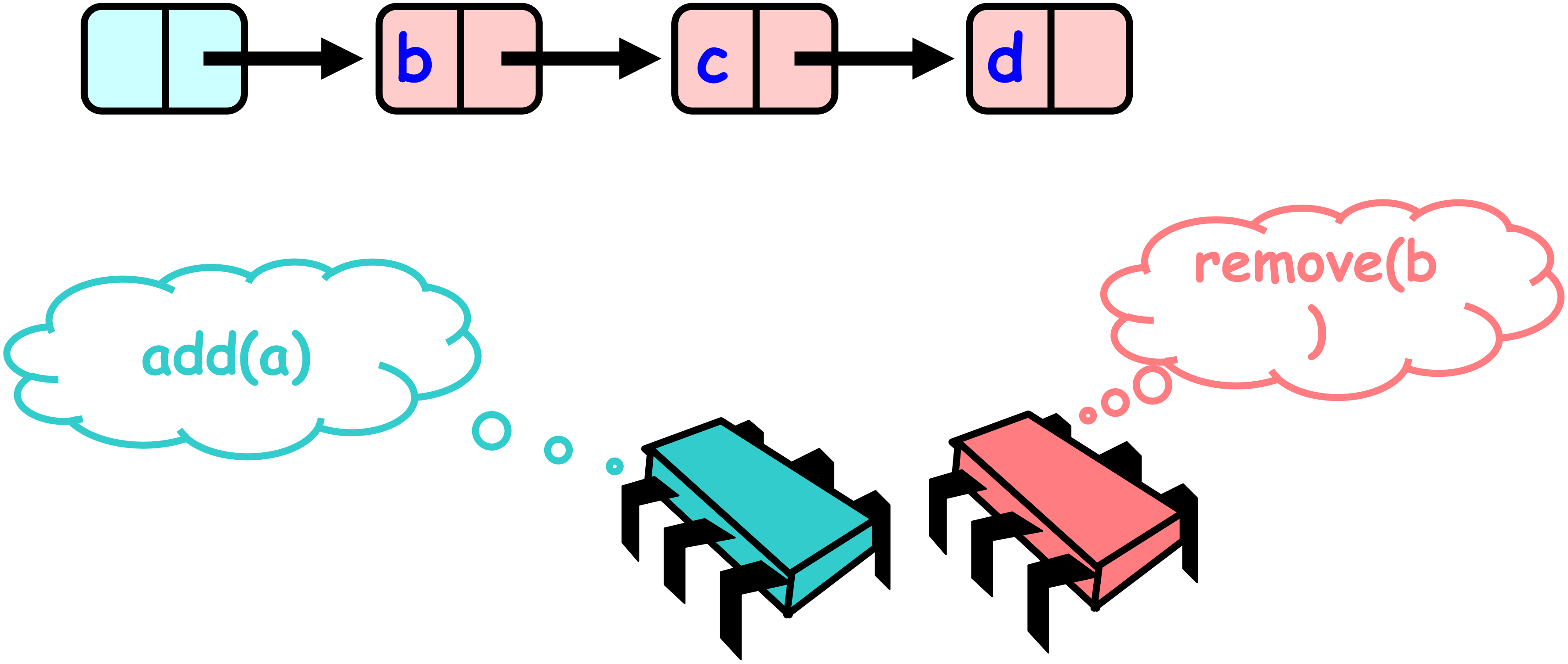
Adding a node



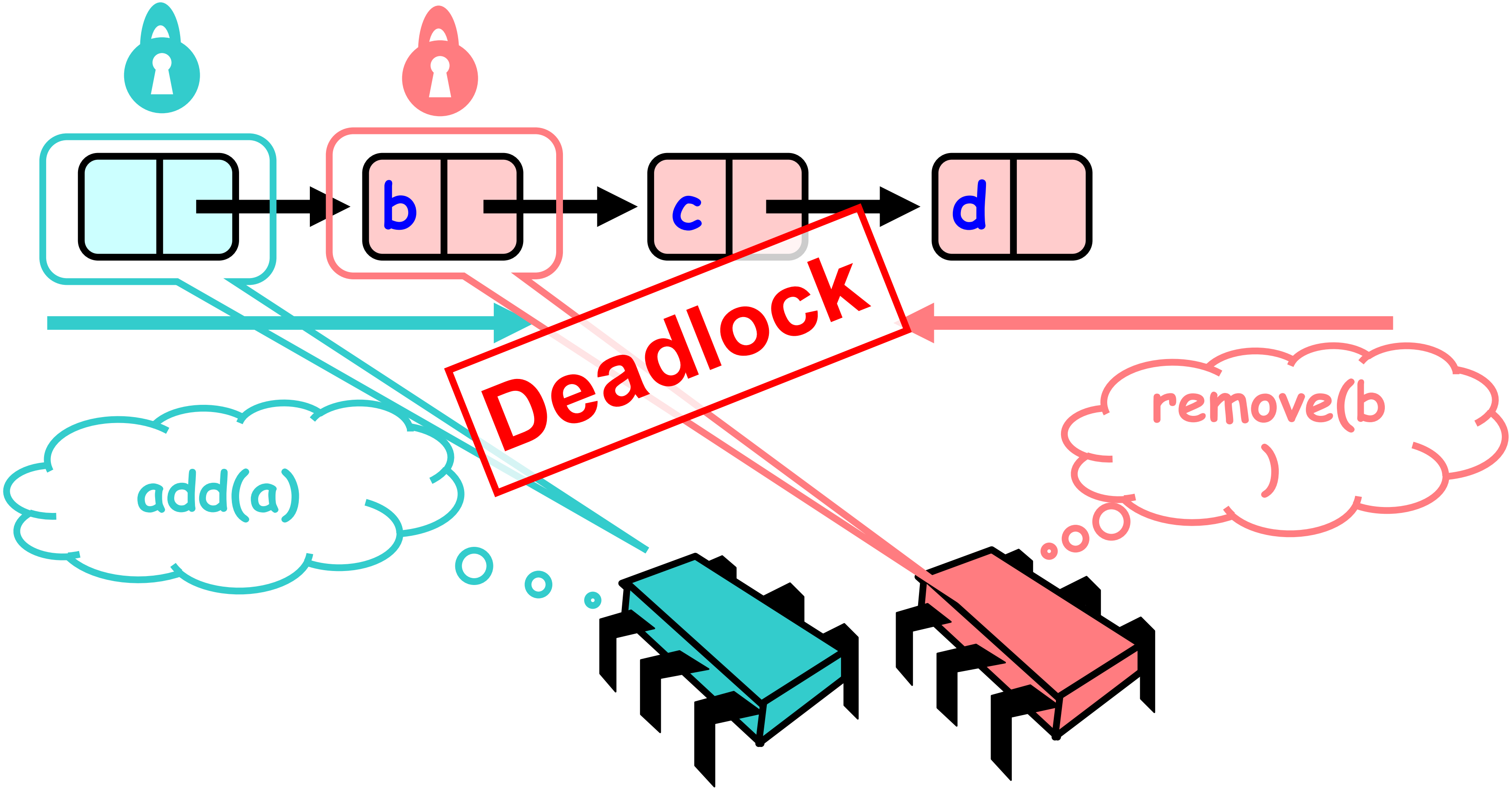
Hand-over-hand locking

- Does it matter whether threads acquire locks in the same order?
- What happens when it does not happen in the same order?

Hand-over-hand locking



Hand-over-hand locking



Fine-grained synchronization

- Although fine-grained synchronization is an improvement over coarse-grained synchronization it is still a potentially long sequence of locks acquisitions and releases
- The algorithm is blocking

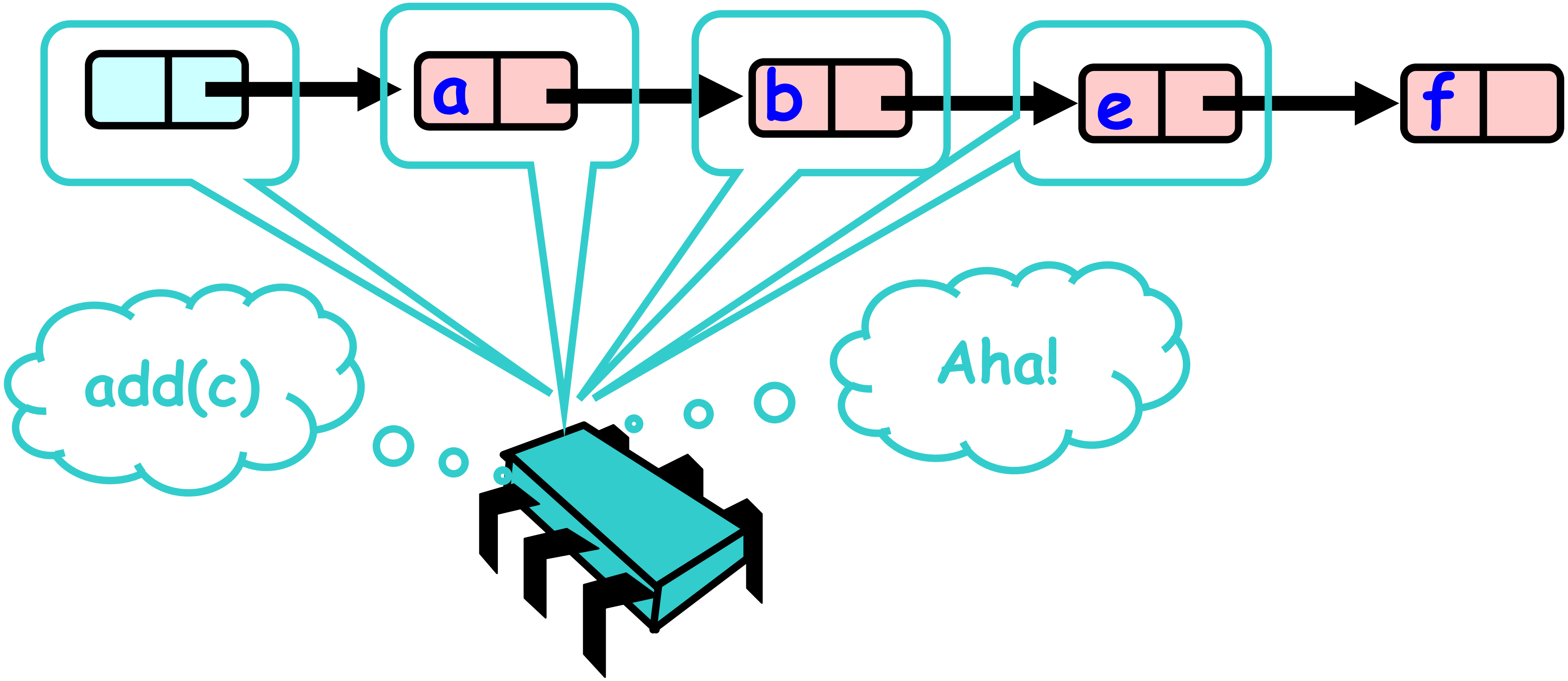
Fine-grained synchronization

- For example:
  - A thread removing the second item in the list still blocks all concurrent threads searching for later nodes
- Possible solution:
  - To take a chance

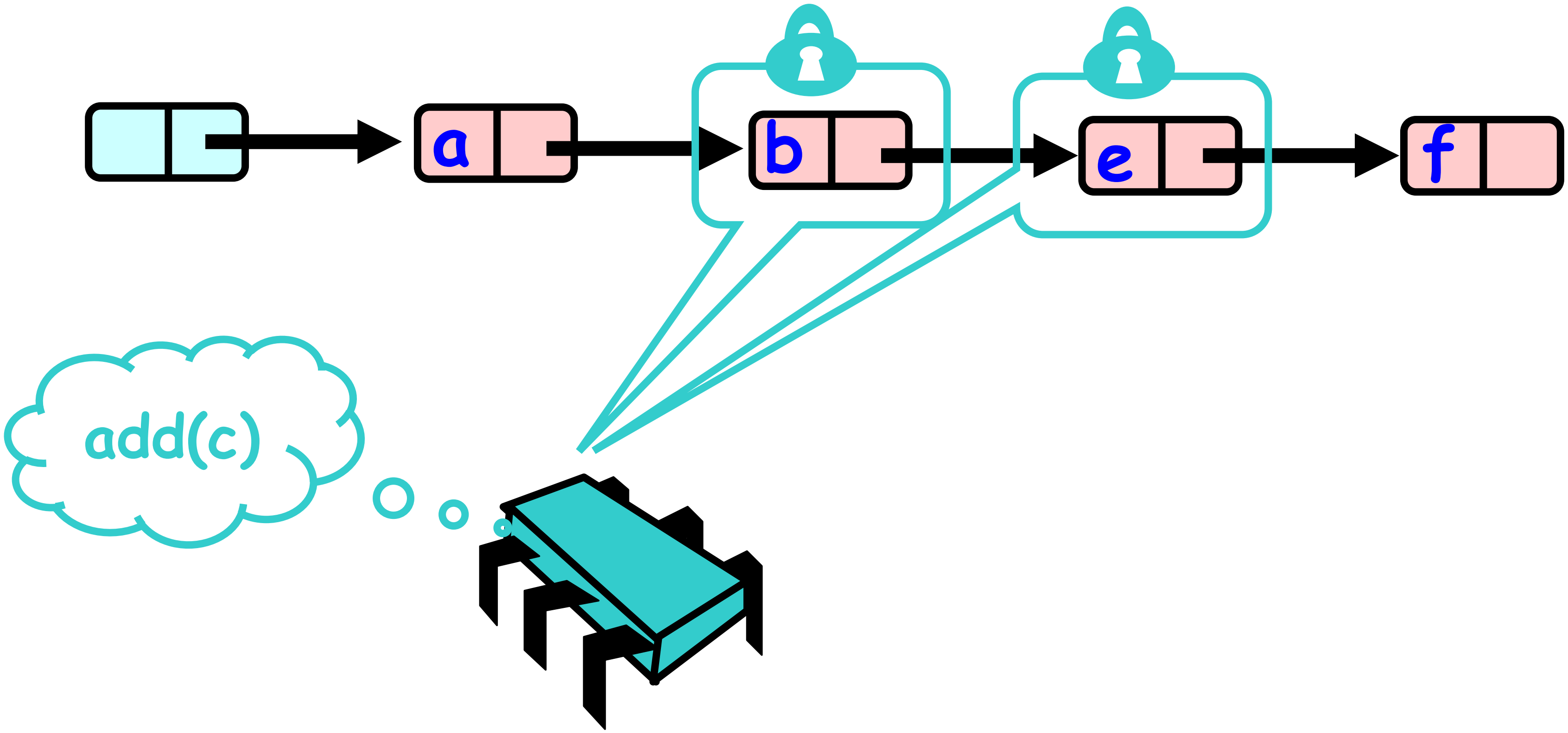
Optimistic synchronization

- Search without acquiring locks
- Lock the nodes found
- Confirm that the locked nodes are correct
- If a synchronization error caused the wrong nodes to be locked, start again

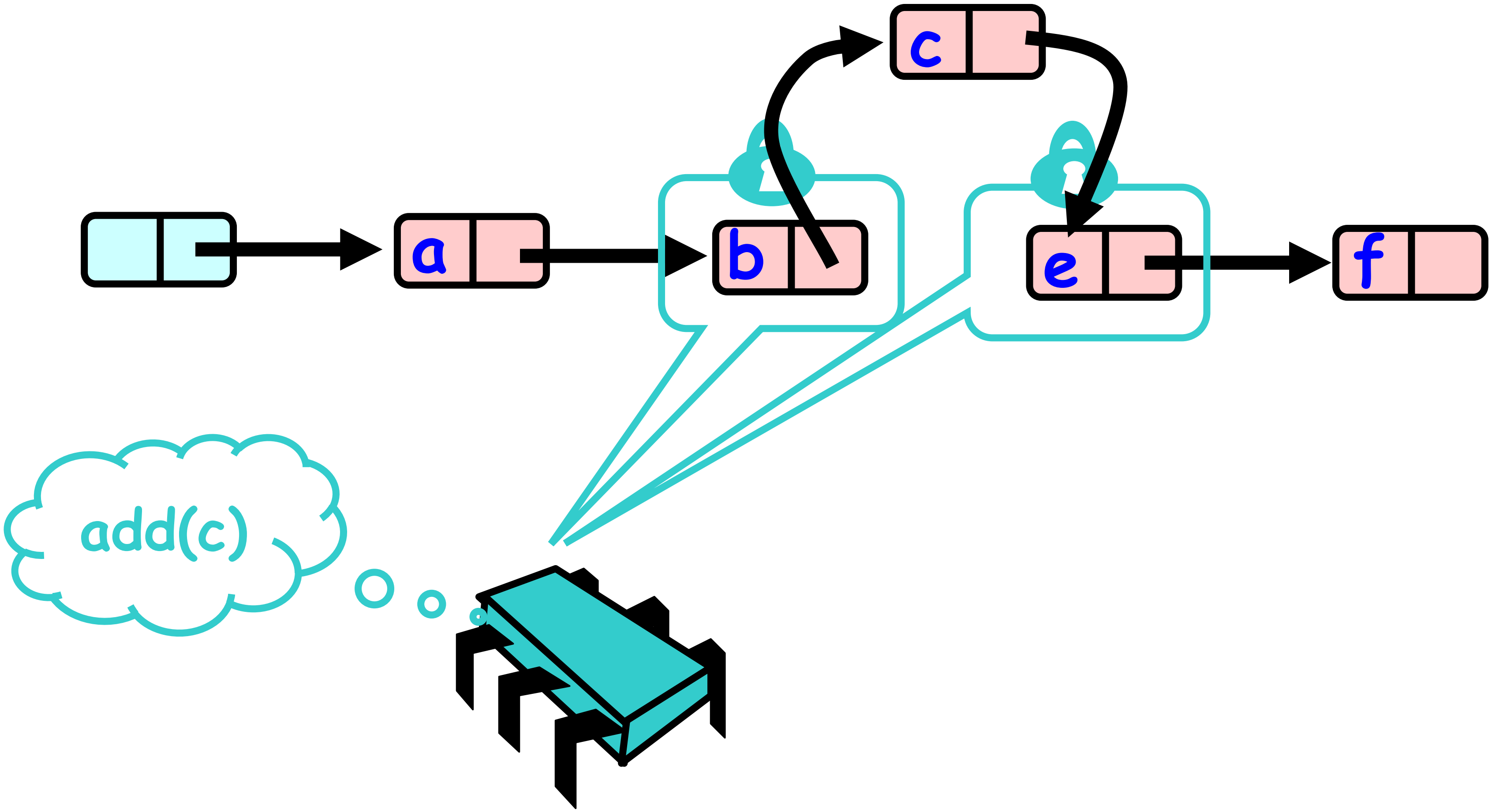
Optimistic: Traverse without Locking



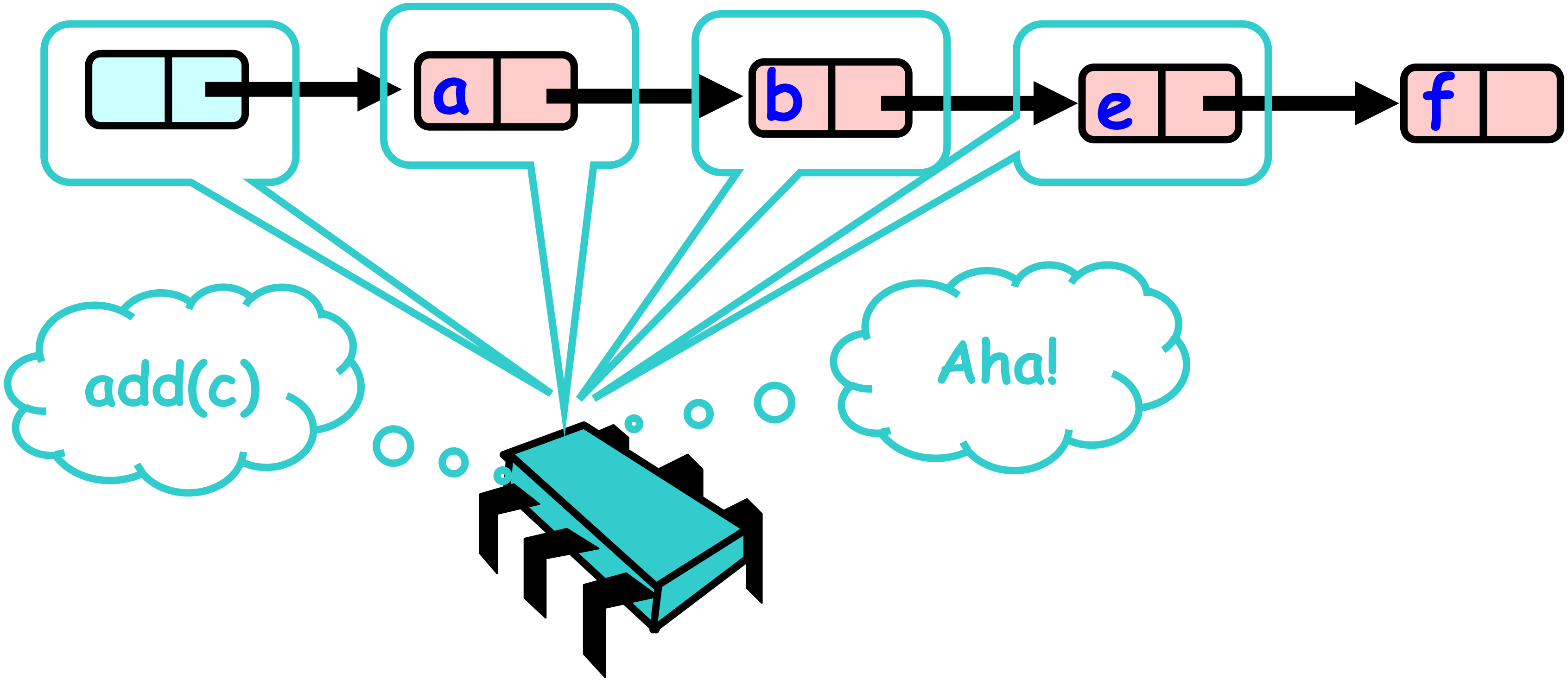
Optimistic: Lock and Load



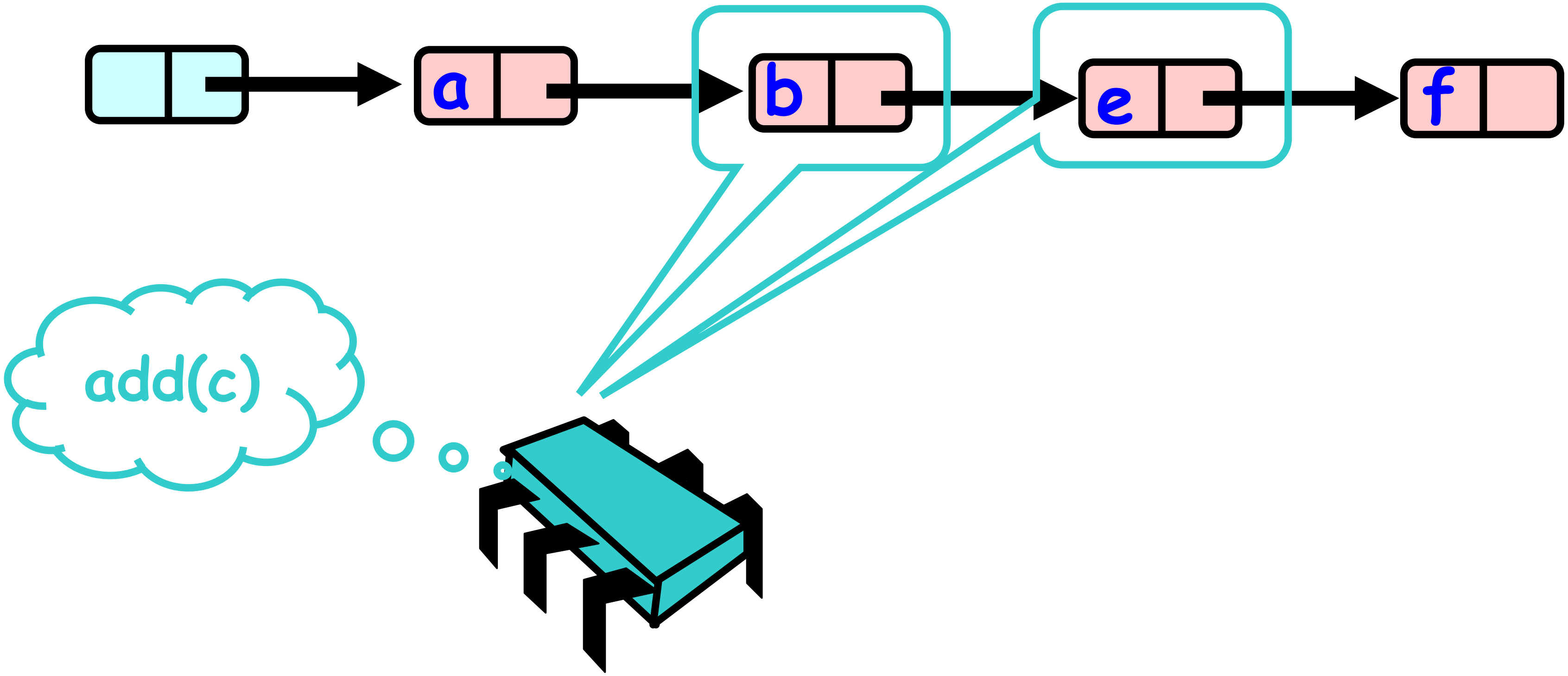
Optimistic: Lock and Load



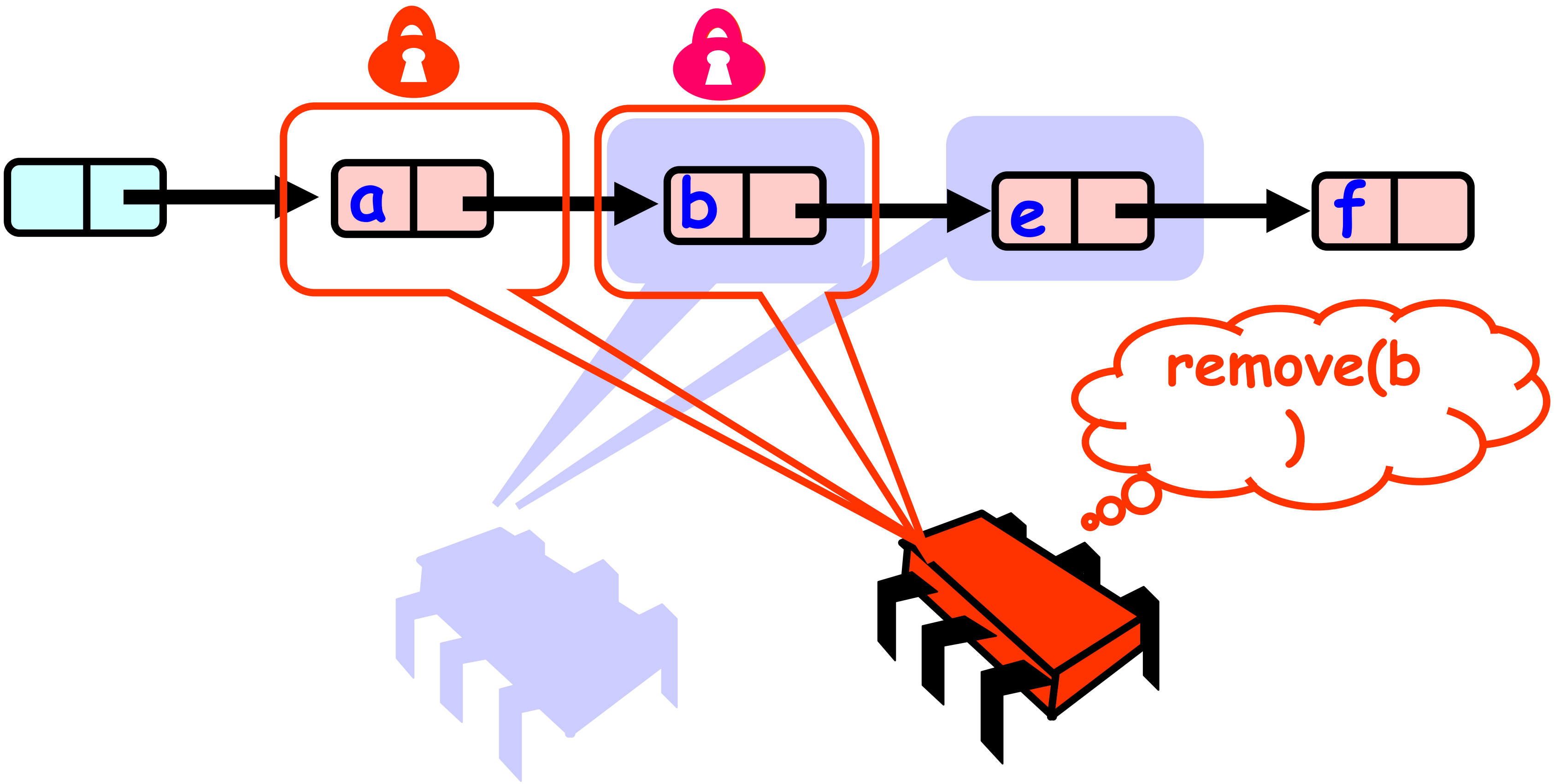
What could go wrong?



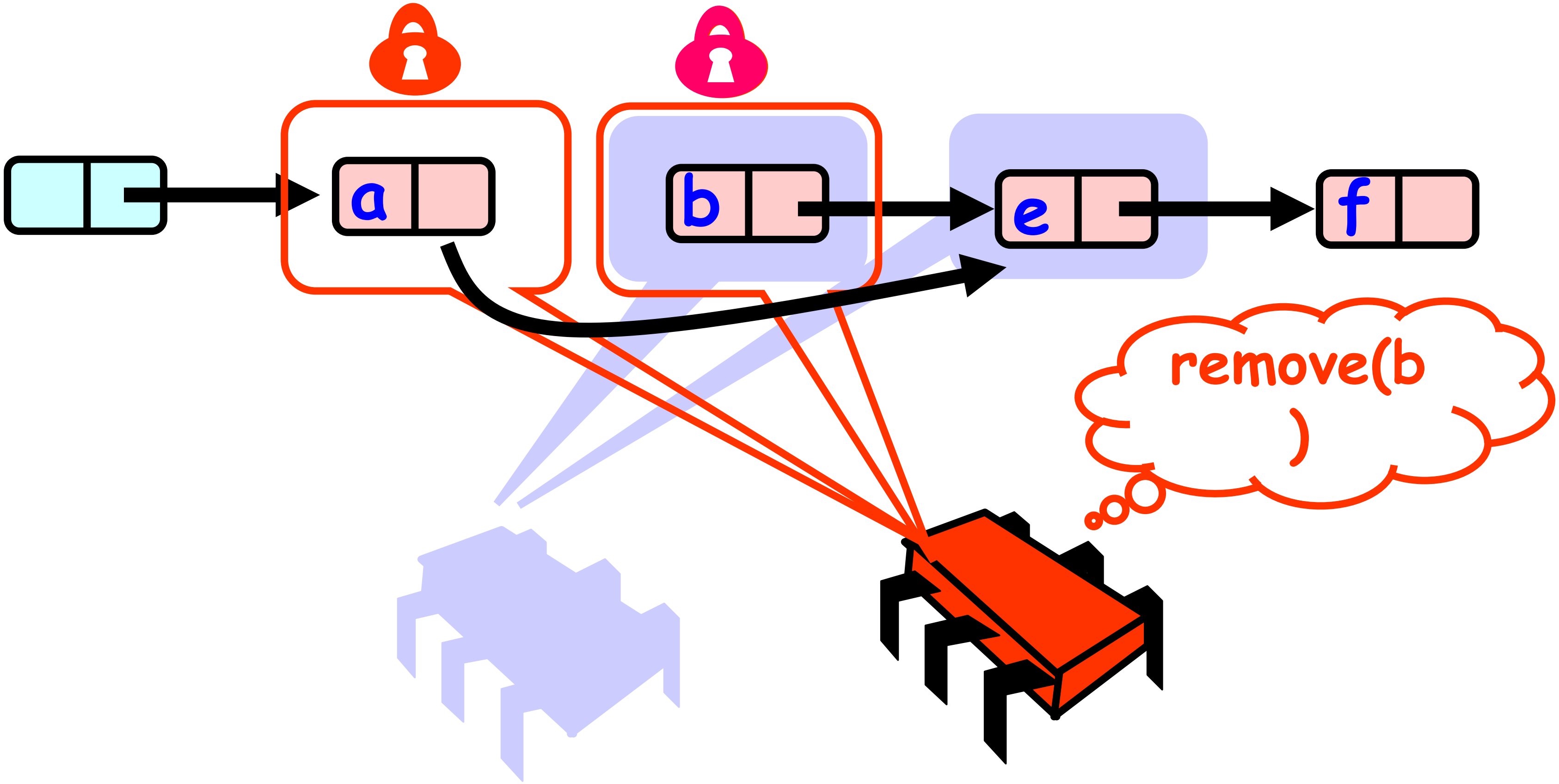
What could go wrong?



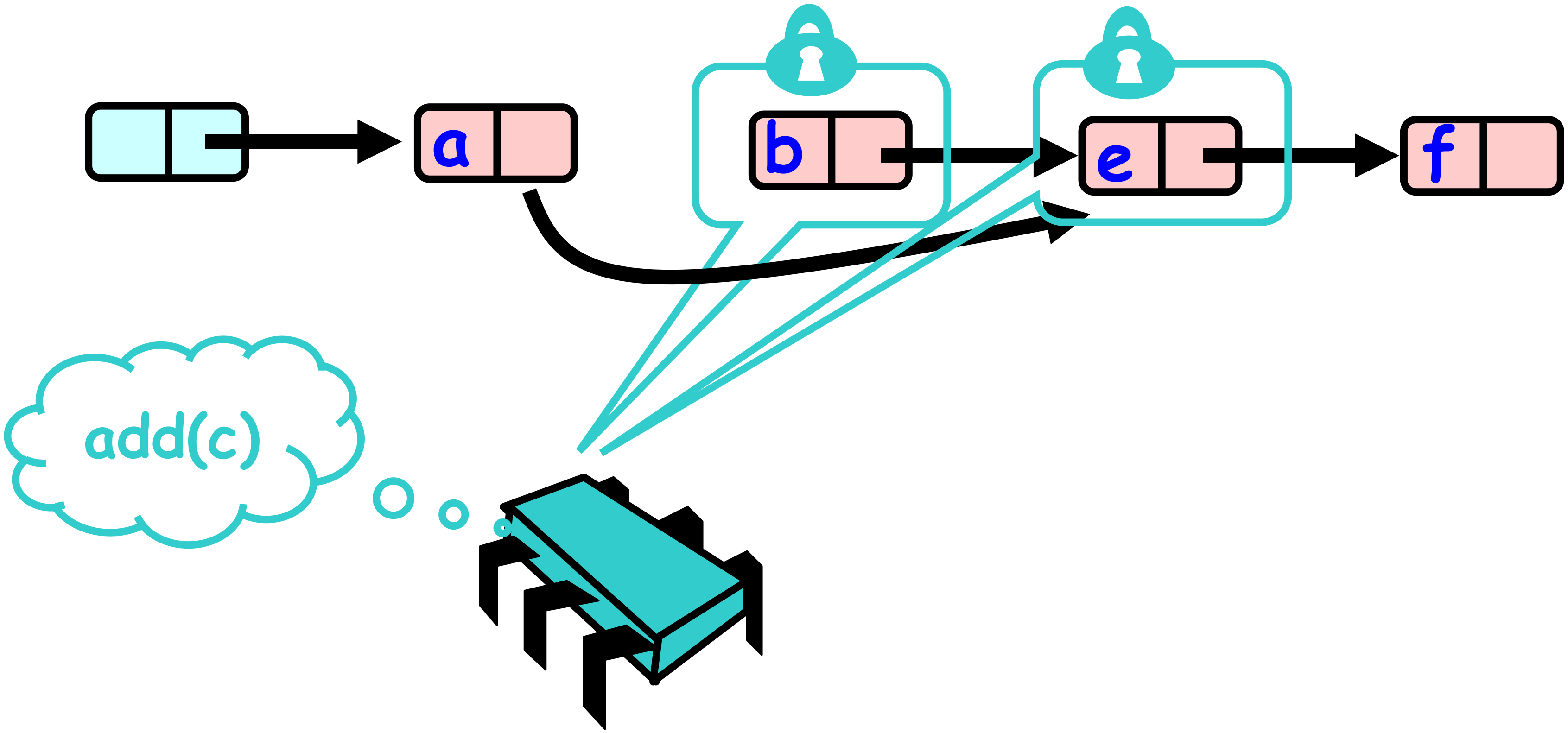
What could go wrong?



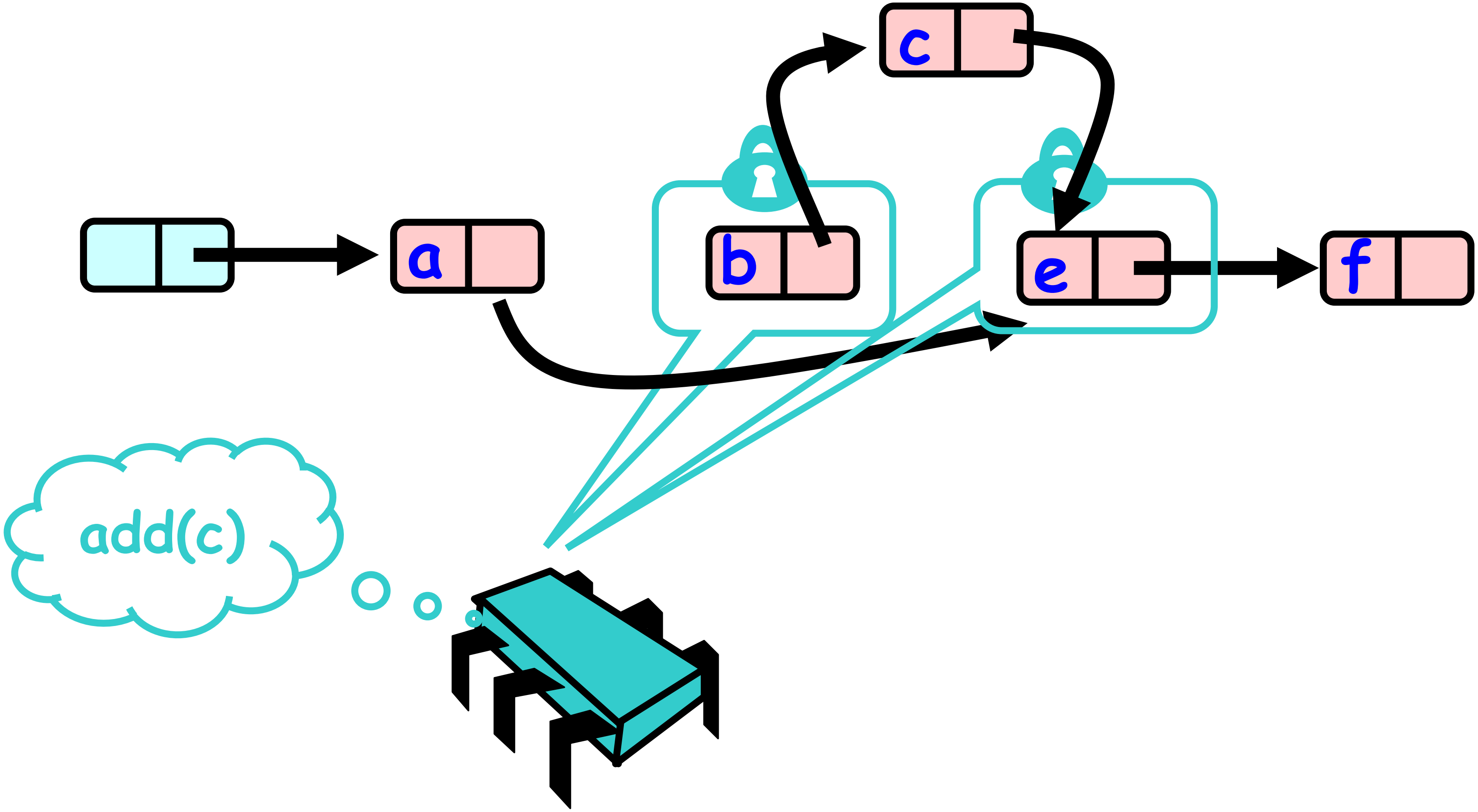
What could go wrong?



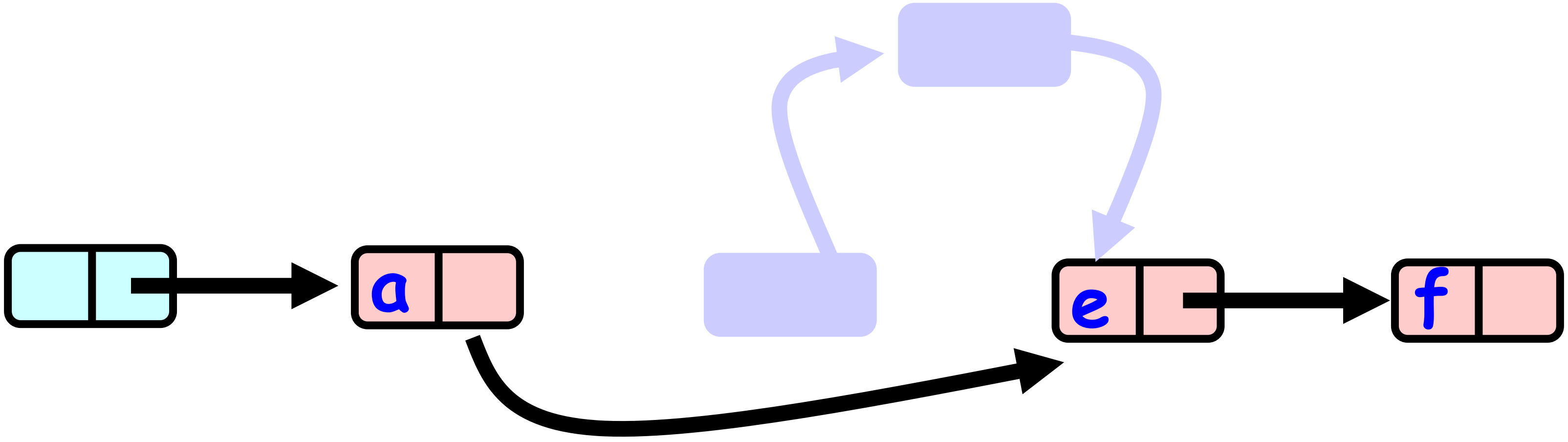
What could go wrong?



What could go wrong?

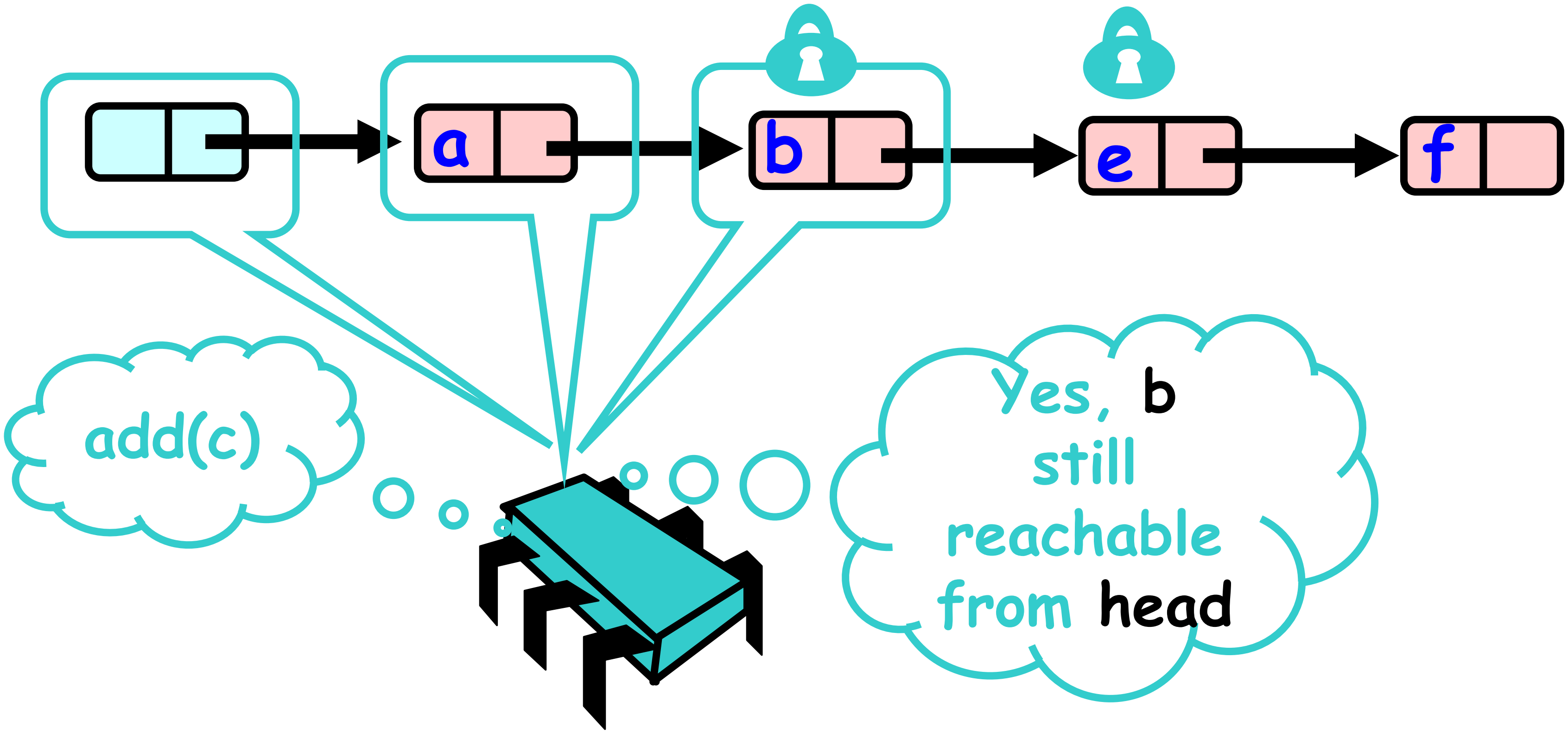


What could go wrong?

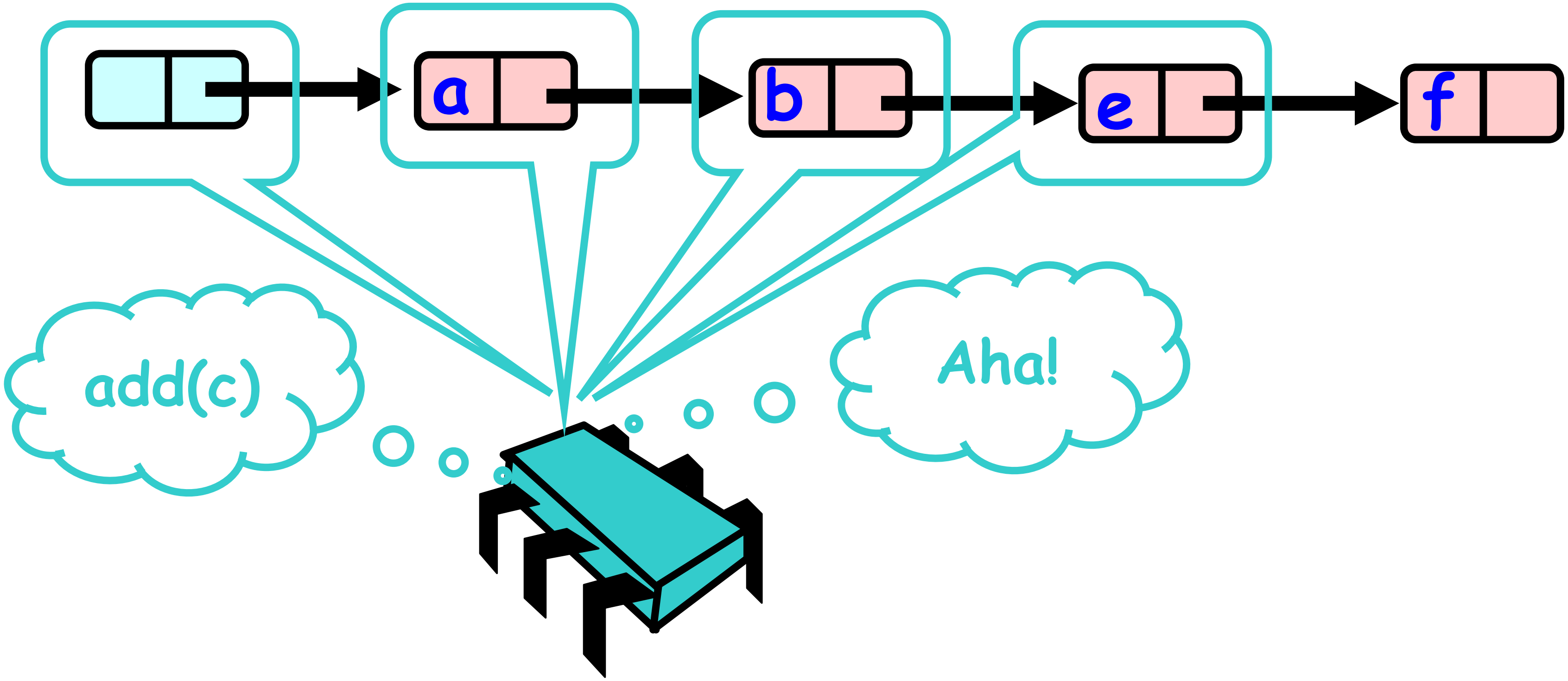


Uh-oh

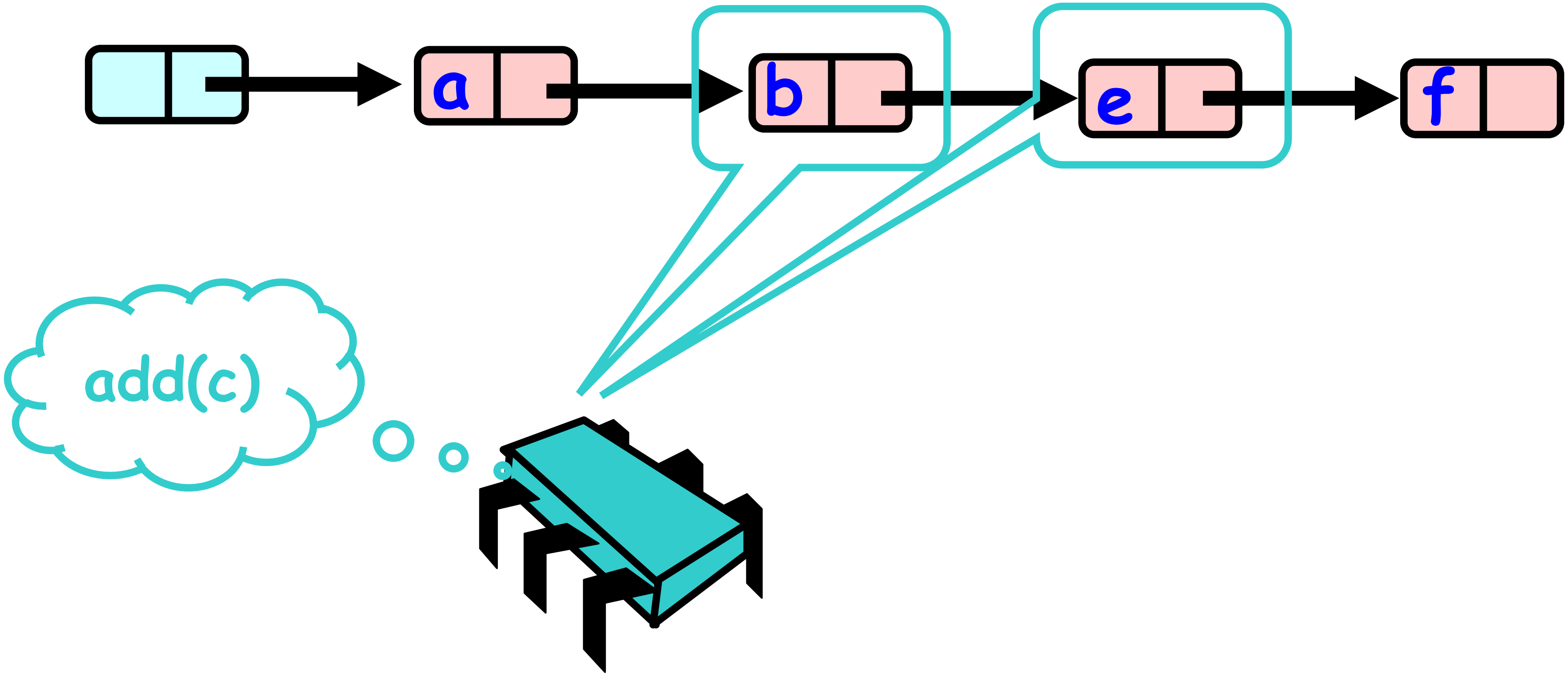
Validate – Part 1



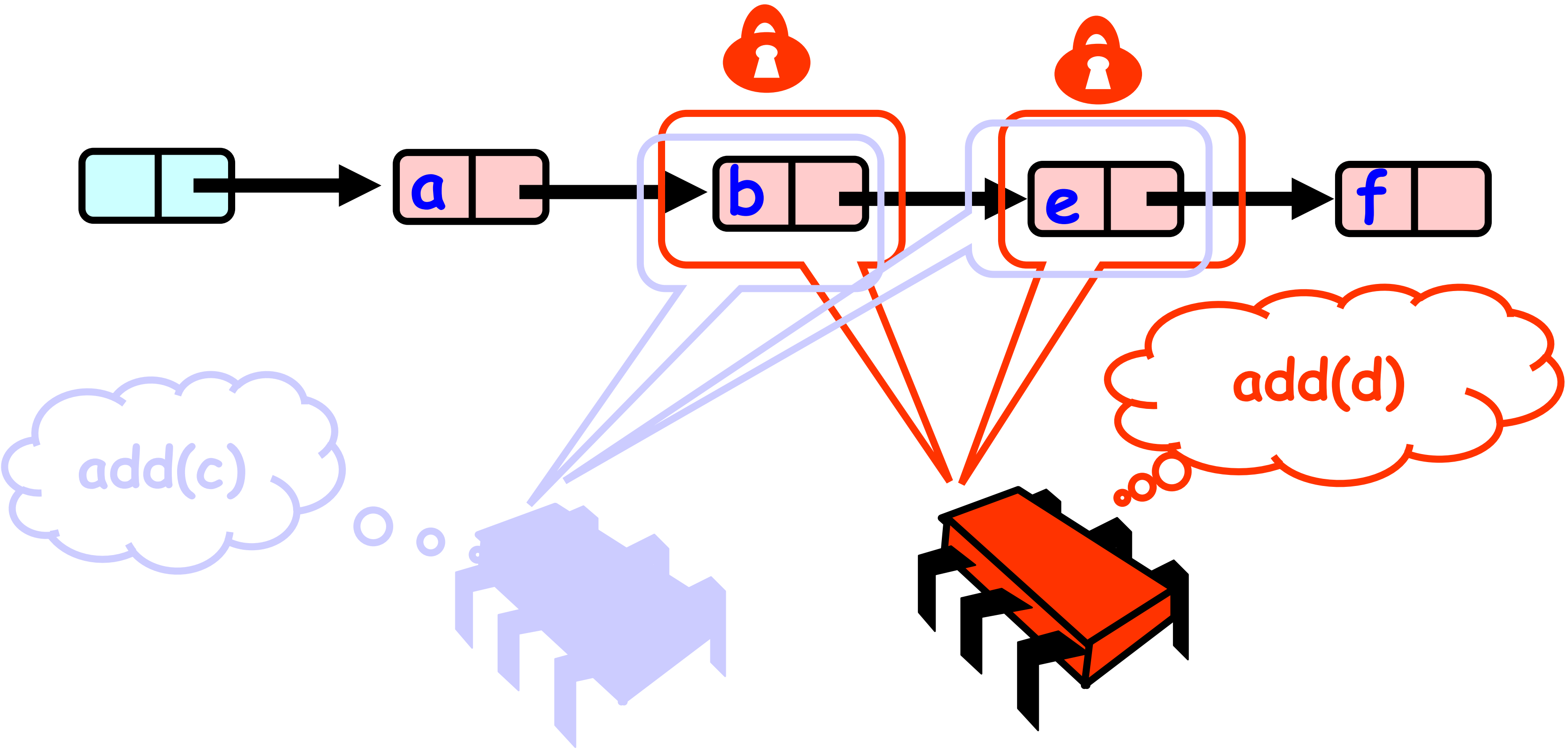
What else could go wrong?



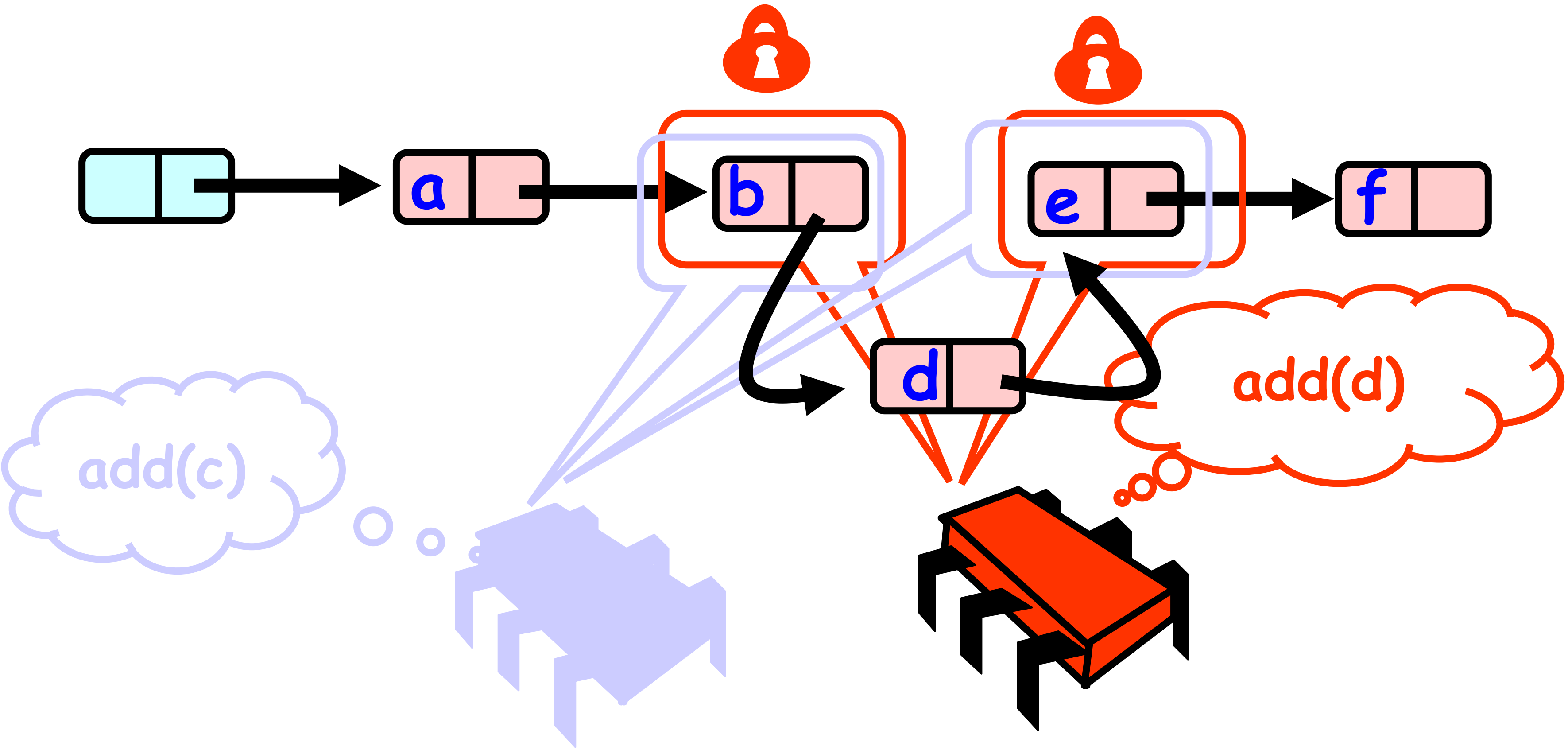
What else could go wrong?



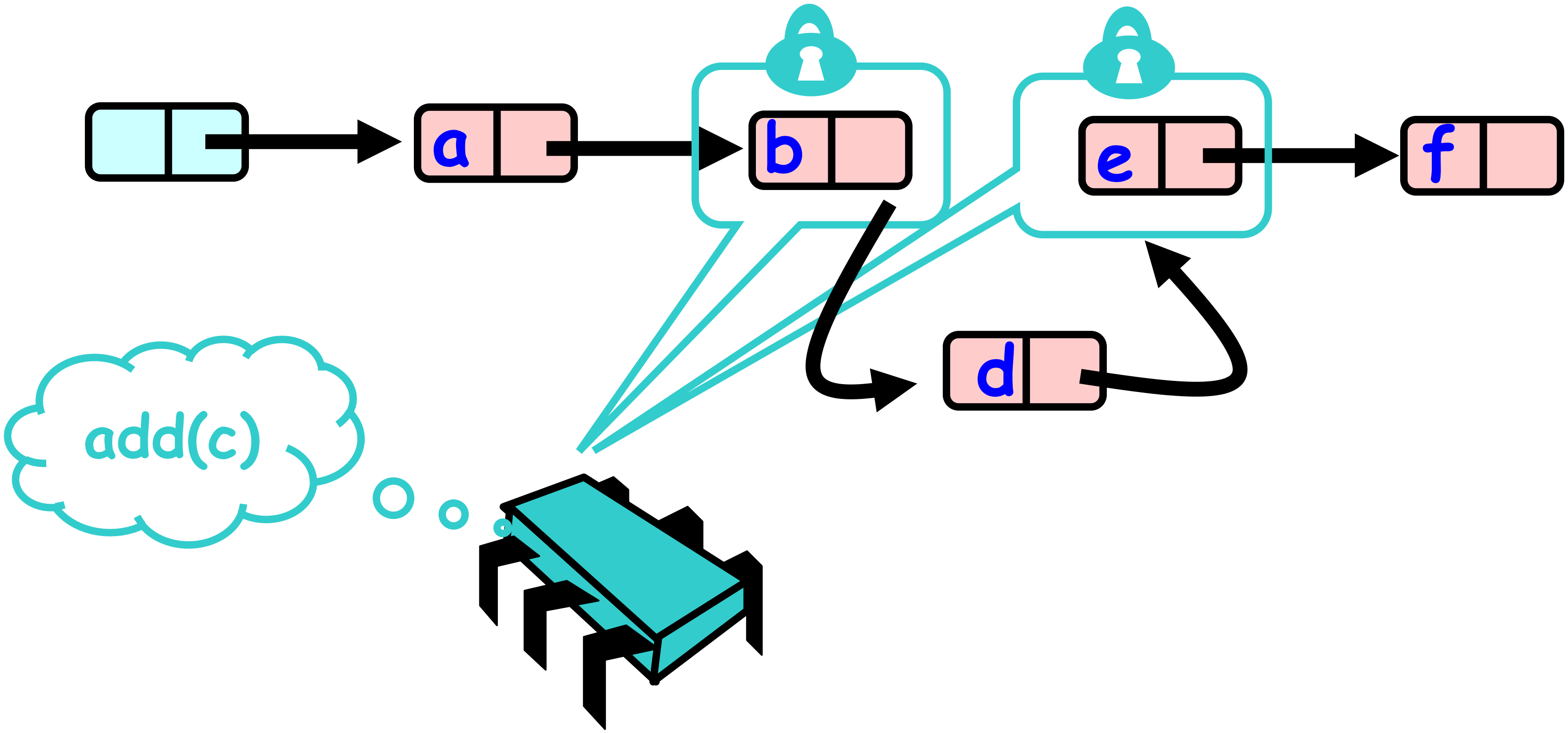
What else could go wrong?



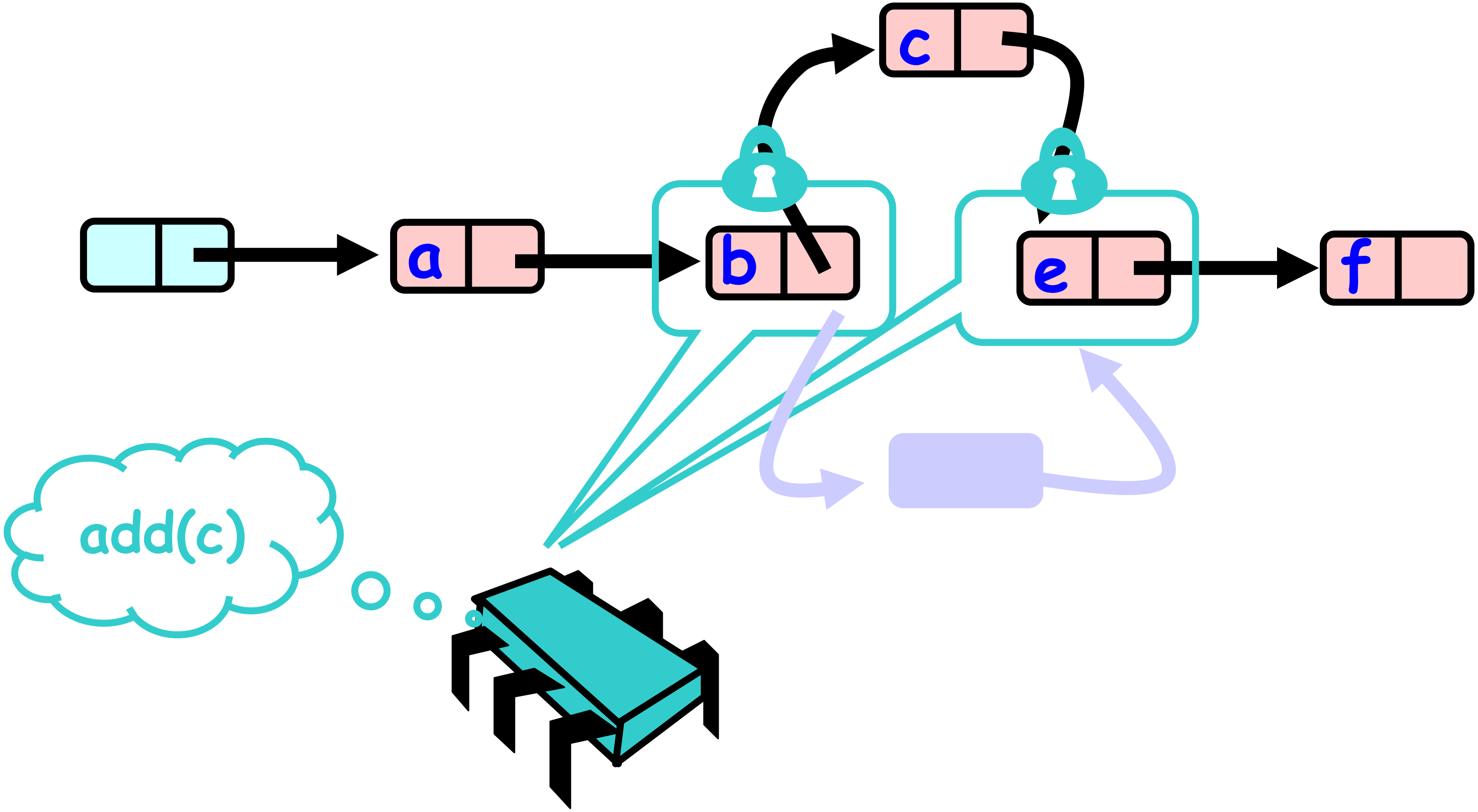
What else could go wrong?



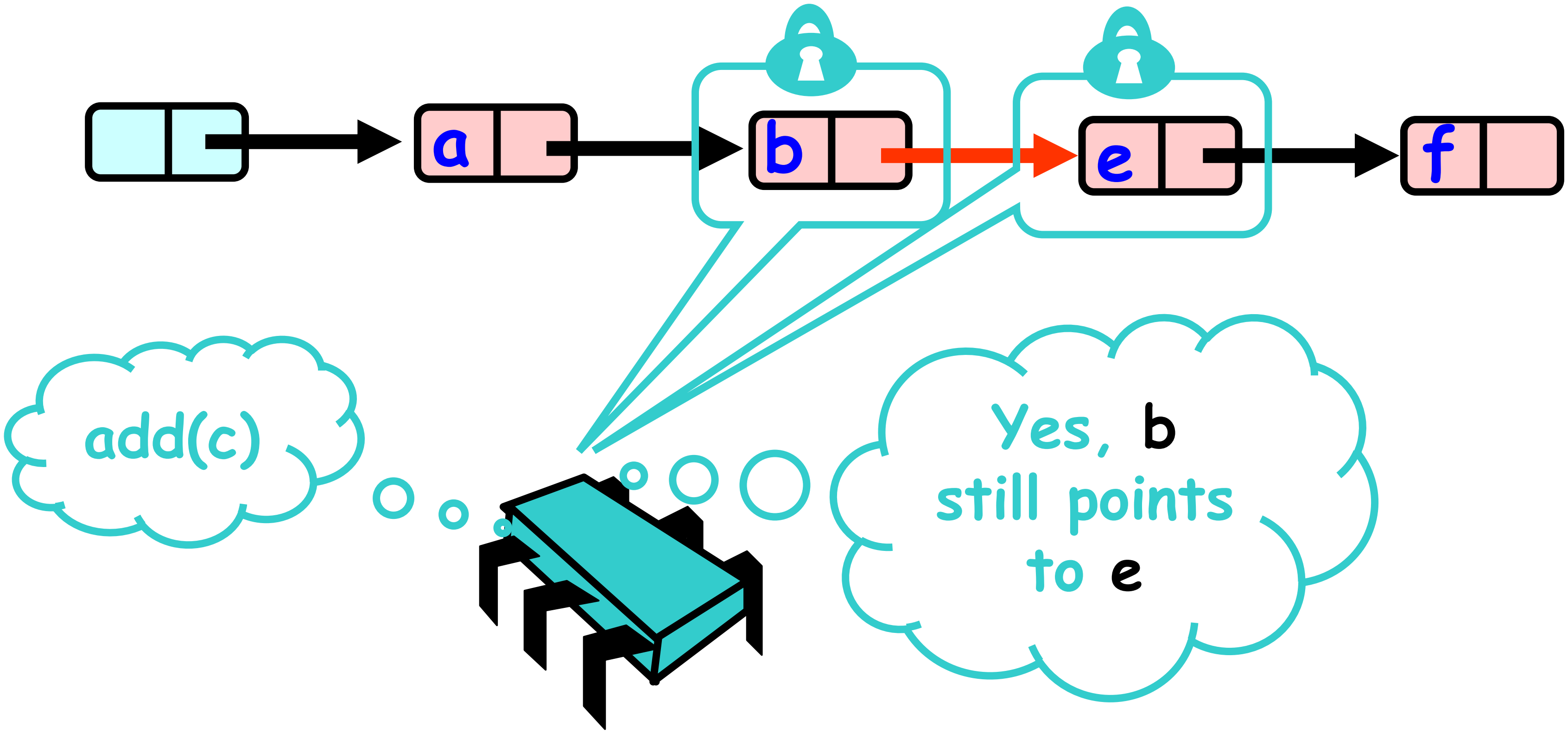
What else could go wrong?



What else could go wrong?



Validate Part 2
(while holding locks)



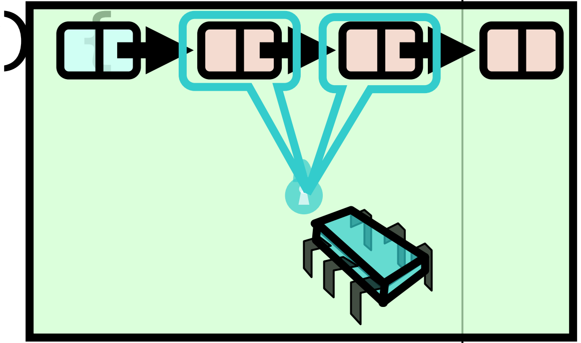
Validation

```
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;
}
```

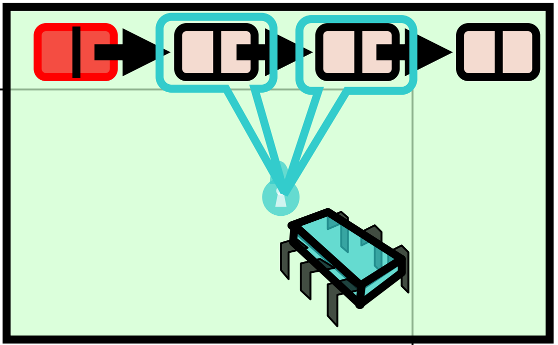
Validation

```
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



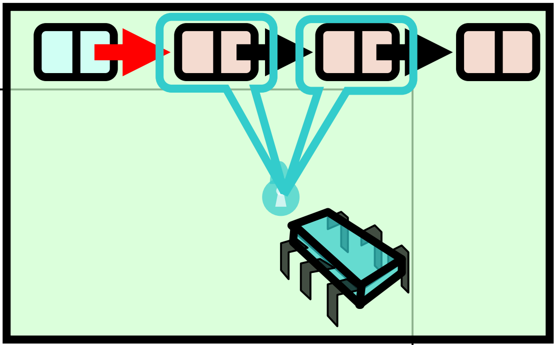
Validation



```
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

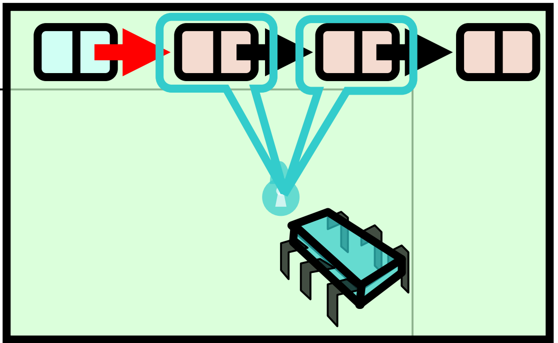


```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Search range of keys

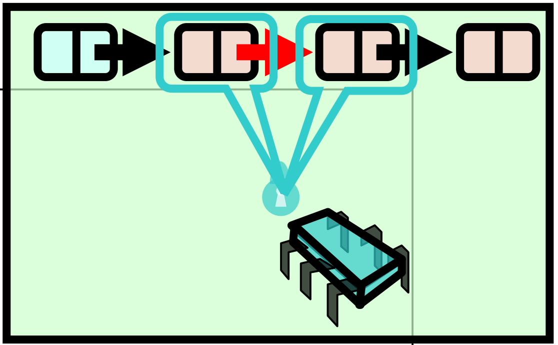
Validation

```
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



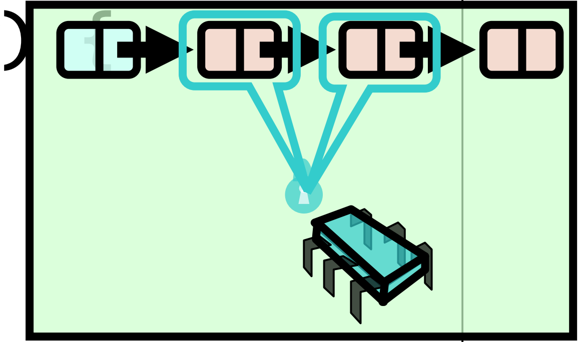
```
private boolean
 validate(Node pred,
          Node curry) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

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;
}
```

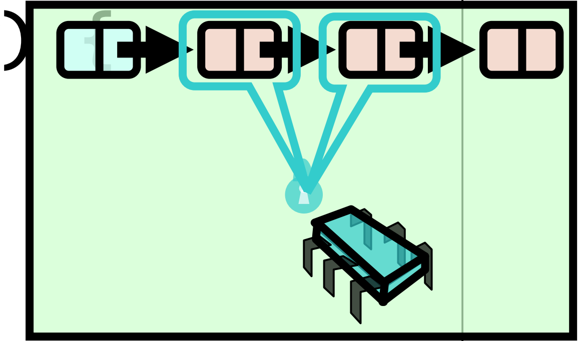
Otherwise move on



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



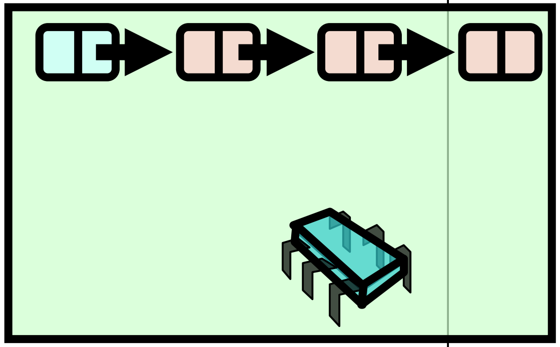
Remove: searching

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred = head;
 Node curr = pred.next;
 while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   } …
```

Remove: searching

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred = head;
 Node curr = pred.next;
 while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   } …
```

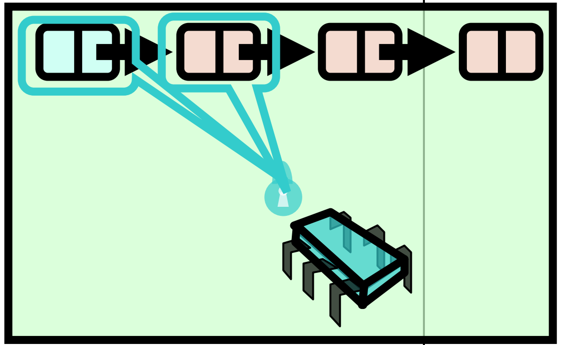
Search key



Remove: searching

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

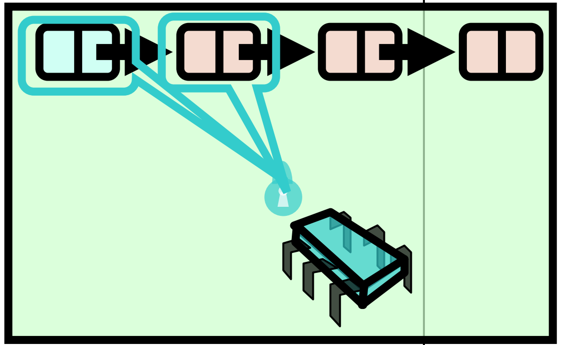
Predecessor and current nodes



Remove: searching

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred = head;
 Node curr = pred.next;
 while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```

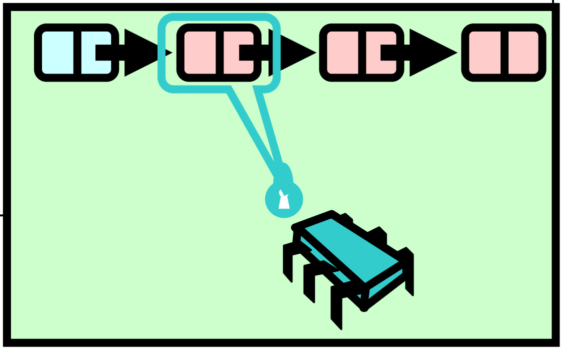
Search by key



Remove: searching

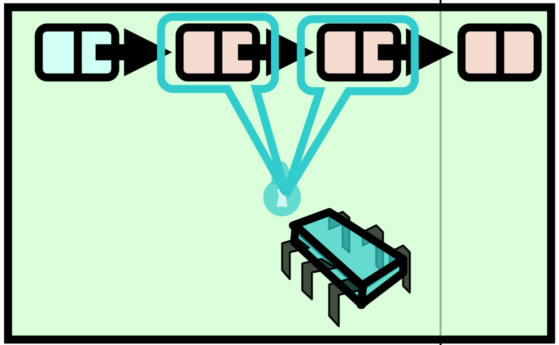
```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred = head;
 Node curr = pred.next;
 while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```

Stop if we find item



Remove: searching

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred = this.head;
 Node curr = pred.next;
 while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```



Move along

On Exit from Loop

- If item is present
  - curr holds item
  - pred just before curr
- If item is absent
  - curr has first higher key
  - pred just before curr
- Assuming no synchronization problems

Remove Method

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

Remove Method

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

Always unlock

Remove Method

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

Lock both nodes

Remove Method

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

Check for synchronization conflicts

Remove Method

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

target found,
remove node

Remove Method

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

target not found

Optimistic List

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

What about contains()?

- Contains() imply that the item is in the list, if and only if it is reachable

- Coarse-grained synchronization?
- Fine-grained synchronization?
- Optimistic synchronization?

Coarse-grained synchronization

- Works much the same as add() and remove()
- Thread acquires lock, searches through the list, returns true/false, releases the lock

Fine-grained synchronization

- Also works much the same as add() and remove()
- Threads that search through the list, acquire and release the pred and curr lock until the item is found or it reaches the end of the list

Optimistic synchronization

- Thread traverses through the list without locking until items are found or the end of the list is reached
- Does this mean that the item is reachable however?
- Nodes are then locked and determined if they are reachable

Optimistic synchronization

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node pred = 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();
  }
```

Optimistic synchronization

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node pred = 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();
  }
```

Search for item

Optimistic synchronization

```
public boolean contains(T item) {
   int key = item.hashCode();
   Node pred = 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();
   }
```

Acquire lock

Optimistic synchronization

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node pred = 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();
  }
```

Is item reachable?

Optimistic synchronization

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

Lazy synchronization

- contains() calls are likely to be made more often than calls to other methods
- Idea of lazy synchronization is to refine optimistic synchronization so that contains() calls are wait-free and add() and remove() calls traverse the list only once (in the absence of contention)

Lazy synchronization

- Each node has an additional boolean field called **marked**
- **marked** indicates whether the node is in the list (reachable)
- Now there is no need to validate if the node is reachable – every unmarked node is reachable

Lazy synchronization

- If a thread does not find a node, or finds it marked, that item is not in the list
- As a result contains() needs only one wait-free traversal

Lazy synchronization

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

Lazy synchronization

- All methods traverse the list ignoring locks
- add() and remove() lock pred and curr as with Optimistic
- Validation however does not require a traversal through the list to determine if a node is reachable

Validation

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

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() method

- Thread traverse through list
- Instead of locking pred and curr, the **marked** field of the target is checked
- Contains() is wait-free

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?

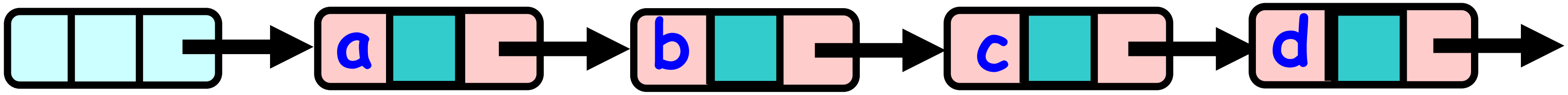
Add() method

- Same as Optimistic synchronization
- Validate() method differs

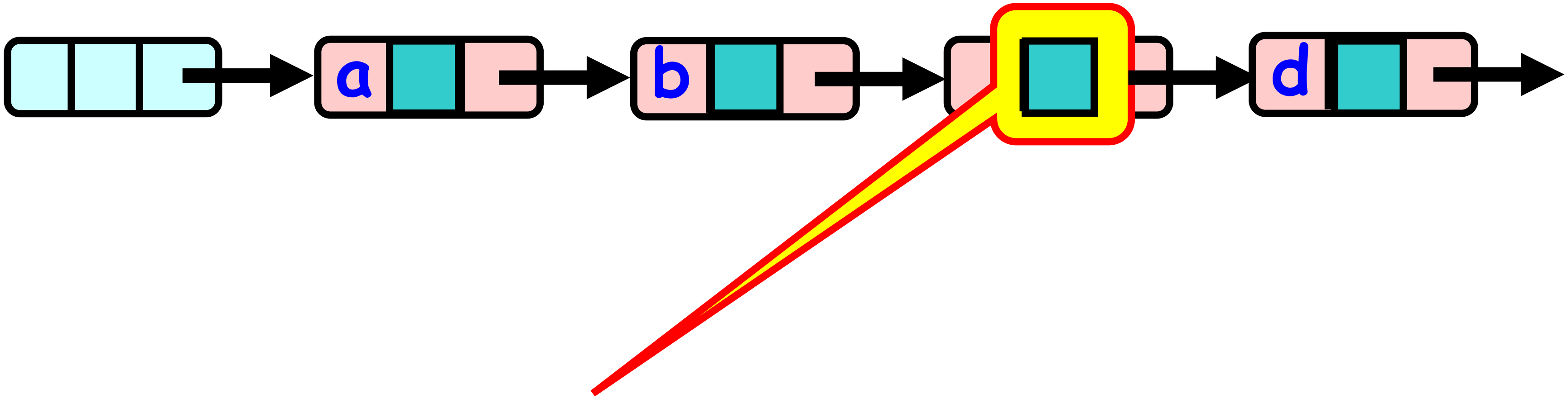
Remove() method

- Divided into:
  - Logical removal – set the node's **marked** field
  - Physical removal – change the links to remove node from linked list
- Thread traverses through list without locks
- When item is found, acquire locks, validate and remove

Lazy Removal

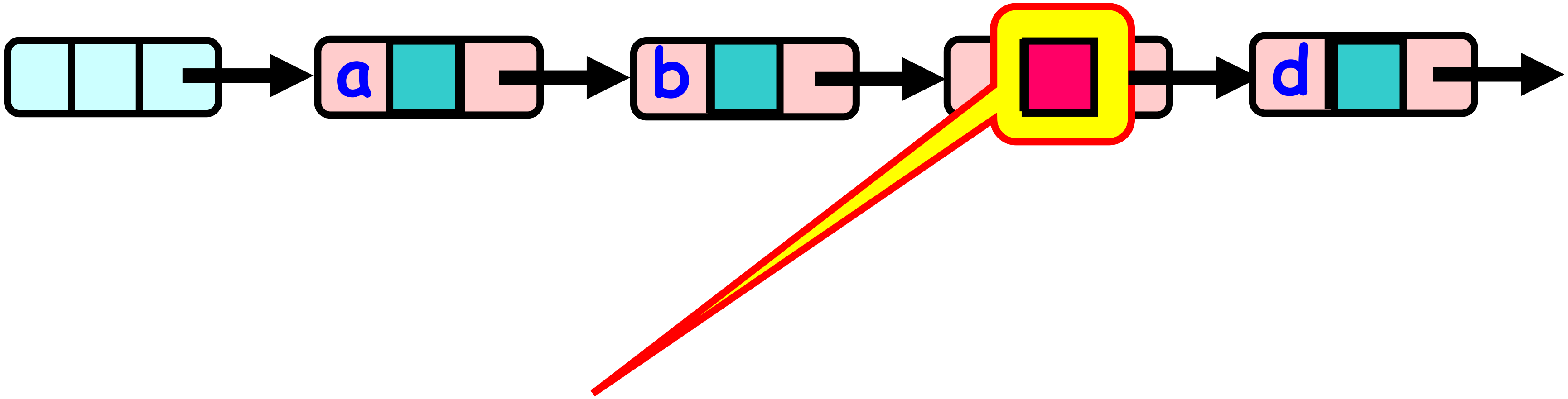


Lazy Removal



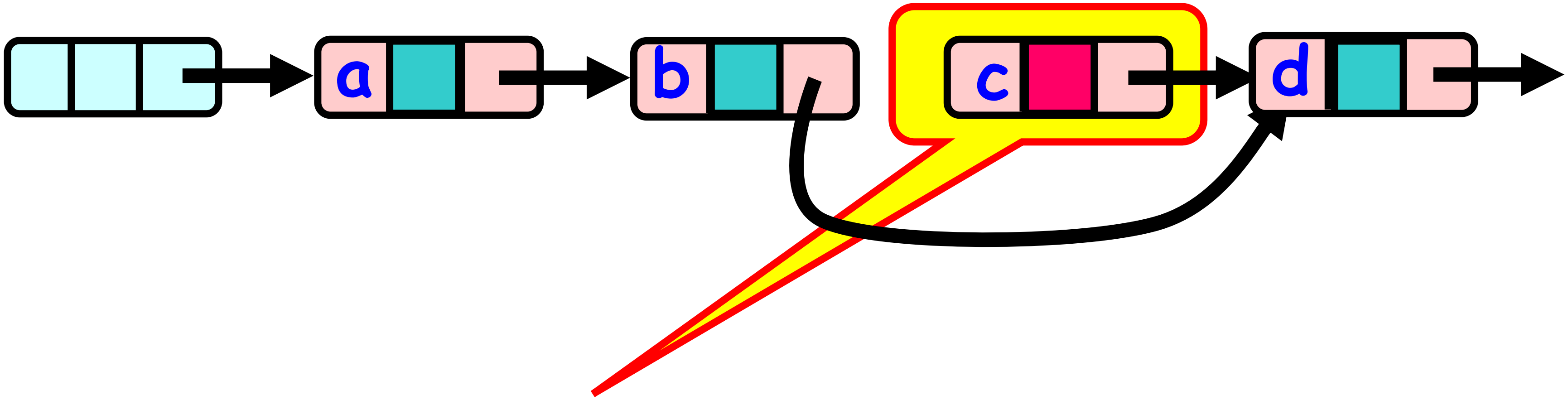
Present in list

Lazy Removal



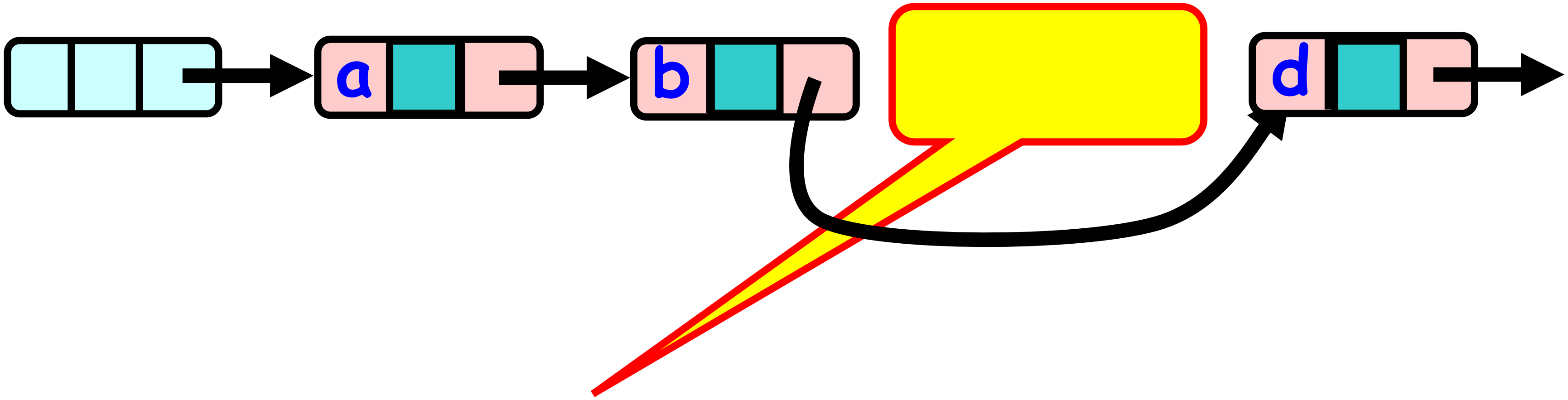
Logically deleted

Lazy Removal



Physically deleted

Lazy Removal



Physically deleted

Remove

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

Remove

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

Remove

```
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 as before

Remove

```
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

Remove

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

Logical remove

Remove

```
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 remove

Evaluation of Lazy Synchronization

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

Difference between Optimistic and Lazy Synchronization

	Optimistic	**Lazy**
contains()	Ignores locks, then locks pred and curr and validates before returning true/false	Ignores locks and returns true/false based on **marked** field
validate(pred, curr)	Traverses through list and validates if node is reachable from head and if curr follows on pred	Does not traverse but validates on **marked** fields of pred and curr and if curr follows on pred
add()	Ignores locks, then locks pred and curr and validates before adding	Ignores locks, then locks pred and curr and validates before adding
remove()	Ignores locks, then locks pred and curr and validates before removing	Ignores locks, then locks pred and curr, validates, changes **marked** field and then removes

Traffic Jam

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

Reminder: Lock-Free Data Structures

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

Lock-free Synchronization

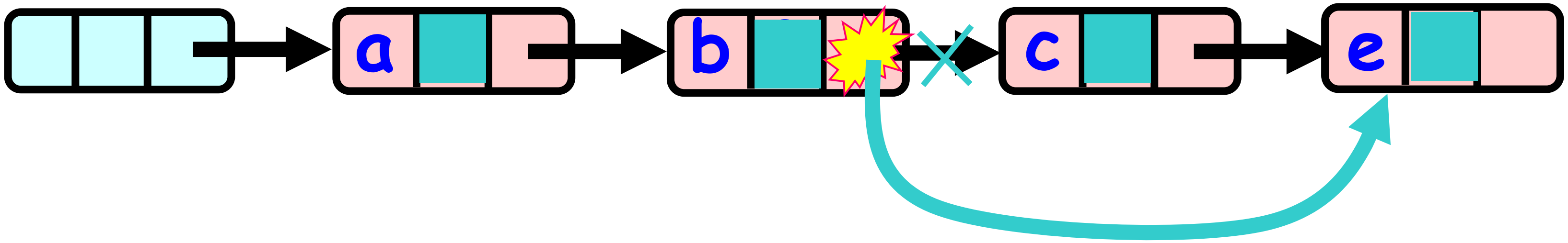
- We already have wait-free contains()
- Next logical step
  - lock-free add() and remove()

- Solution:
  - Use compareAndSet()

Lock-Free synchronization

- Make use of compareAndSet() to change the next links when items are added or removed
- Since compareAndSet() is atomic, mutual exclusion is enforced
- What could go wrong?

Remove Using CAS

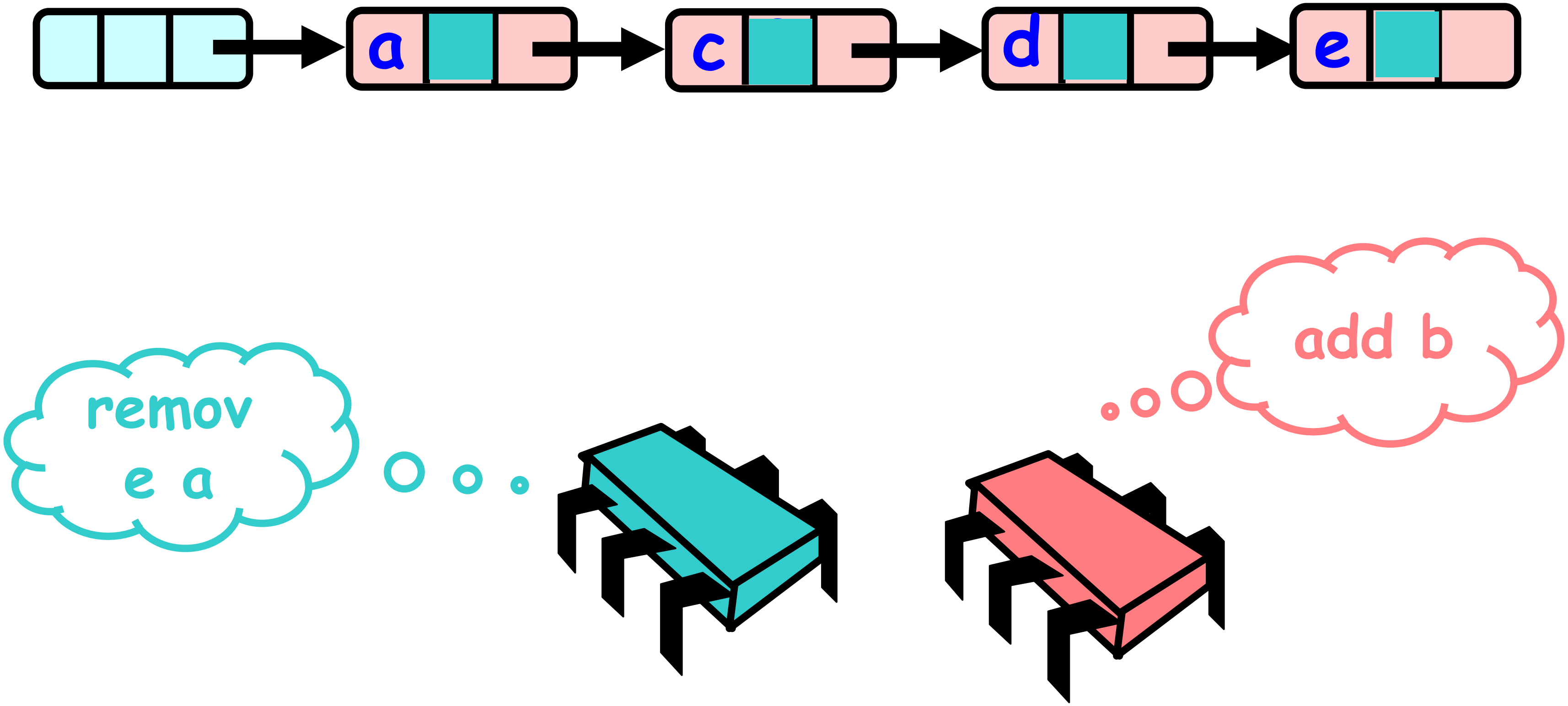


- remove(c)
- Use compareAndSet() to set b’s next field to point to e

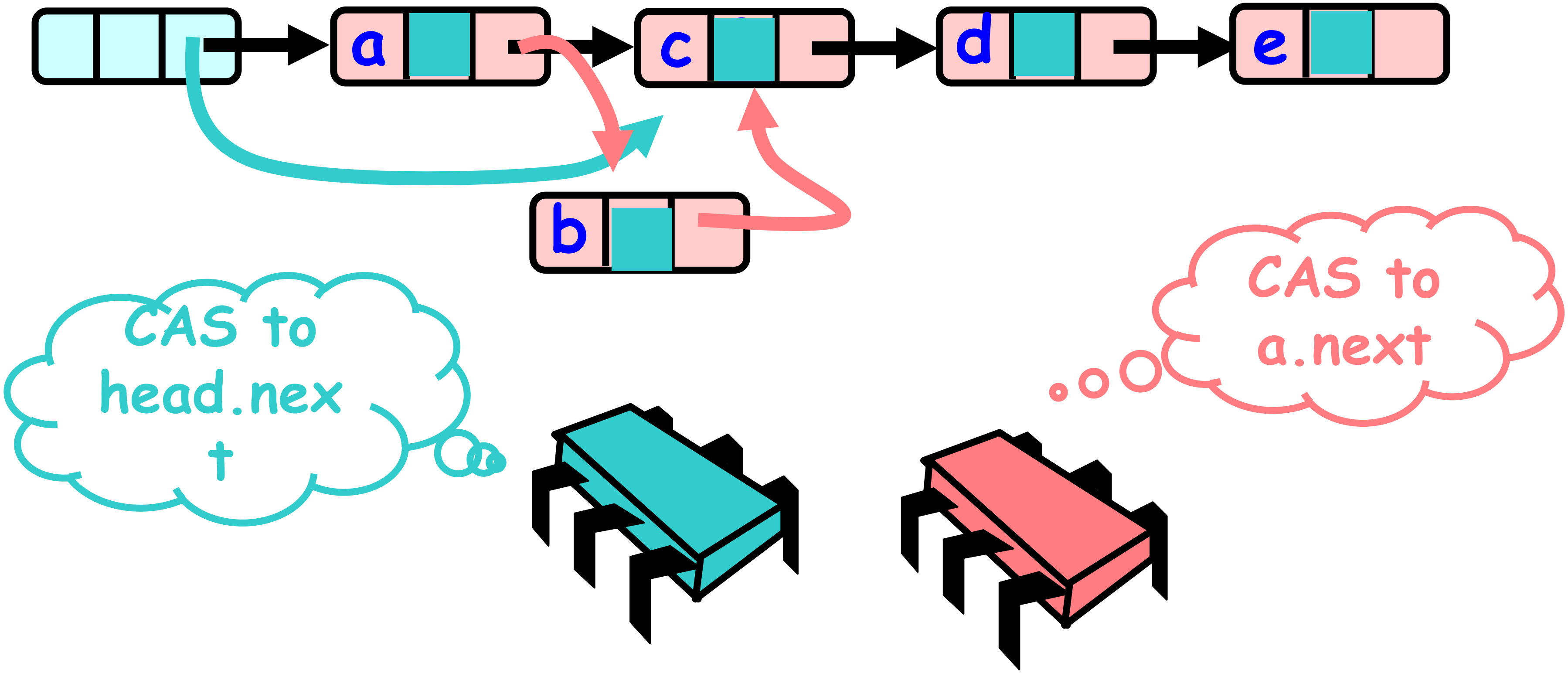
Remove using CAS

- Unfortunately this idea does not work

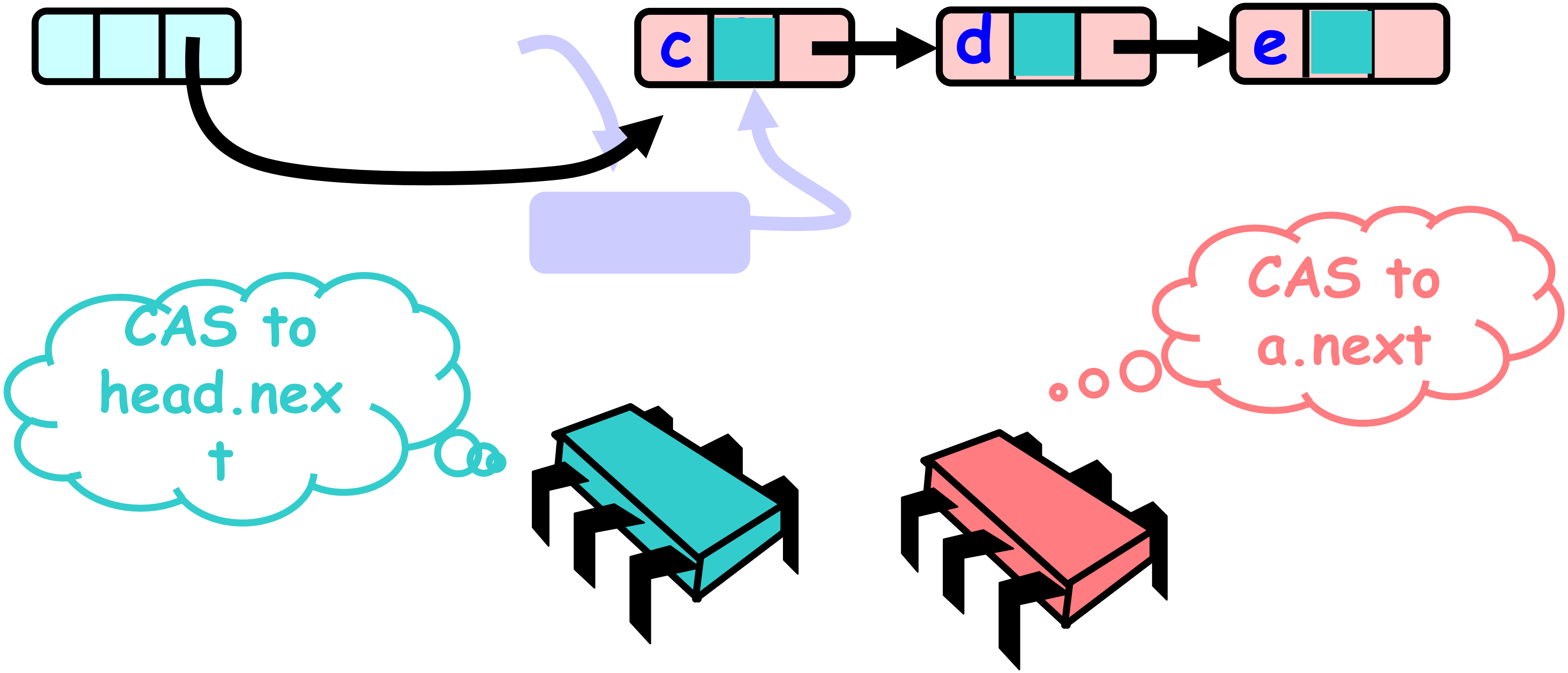
Remove using CAS



Remove using CAS



Remove using CAS



Non-blocking synchronization

- We need a way to ensure that a node’s fields cannot be updated after that node has been logically/physically deleted from the list
- Use a **marked** field
- Any attempt to update the next field when the **marked** field is true will fail

Non-blocking synchronization

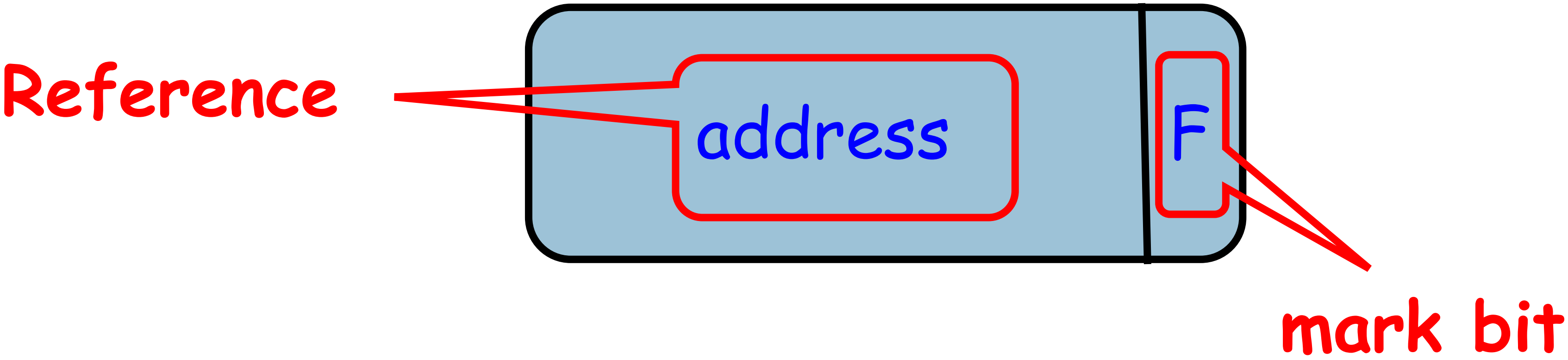
- Our approach:
  - To treat the next and marked fields as a single unit

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



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 …

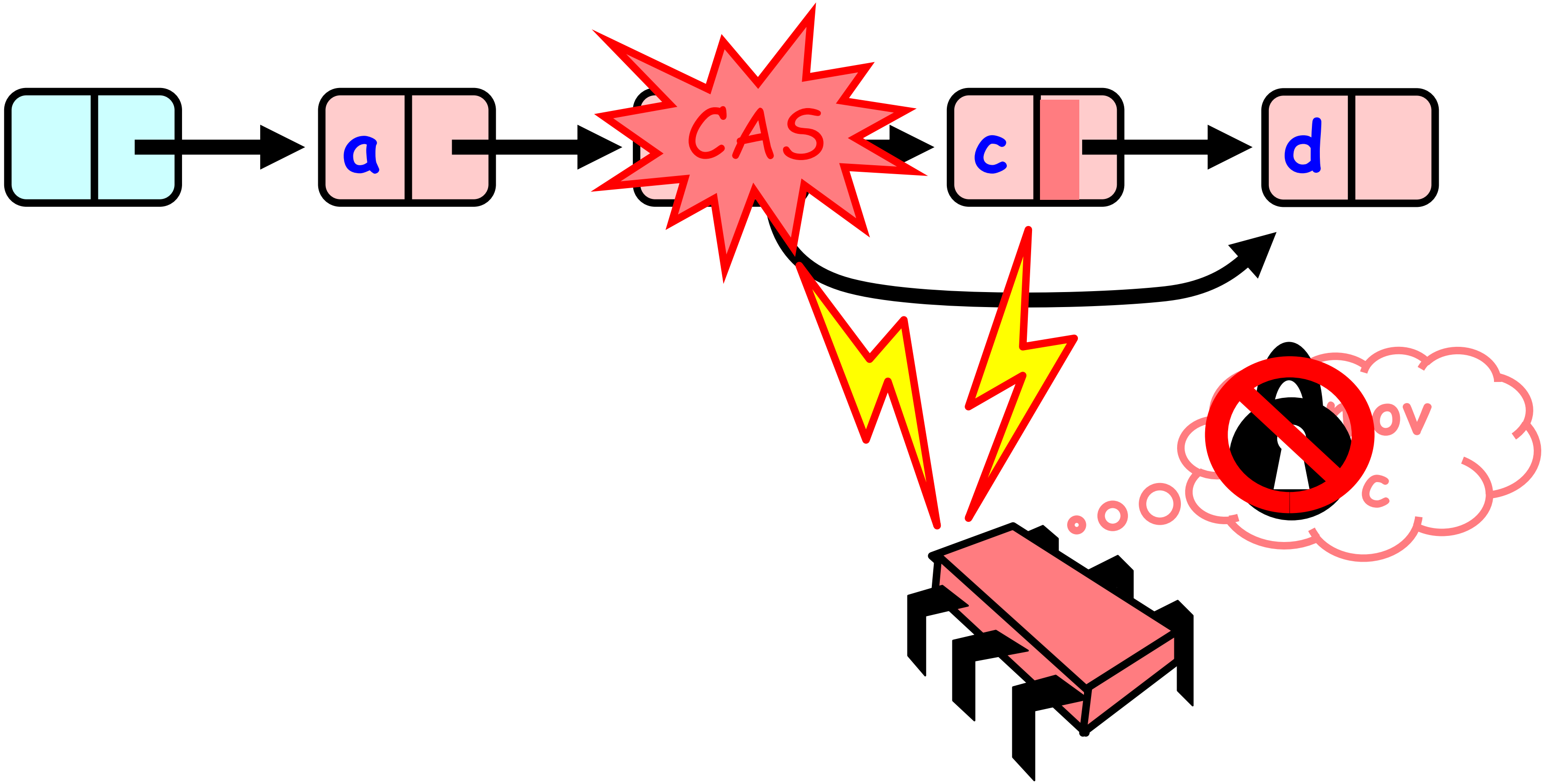
Changing State

…then change to this new reference …

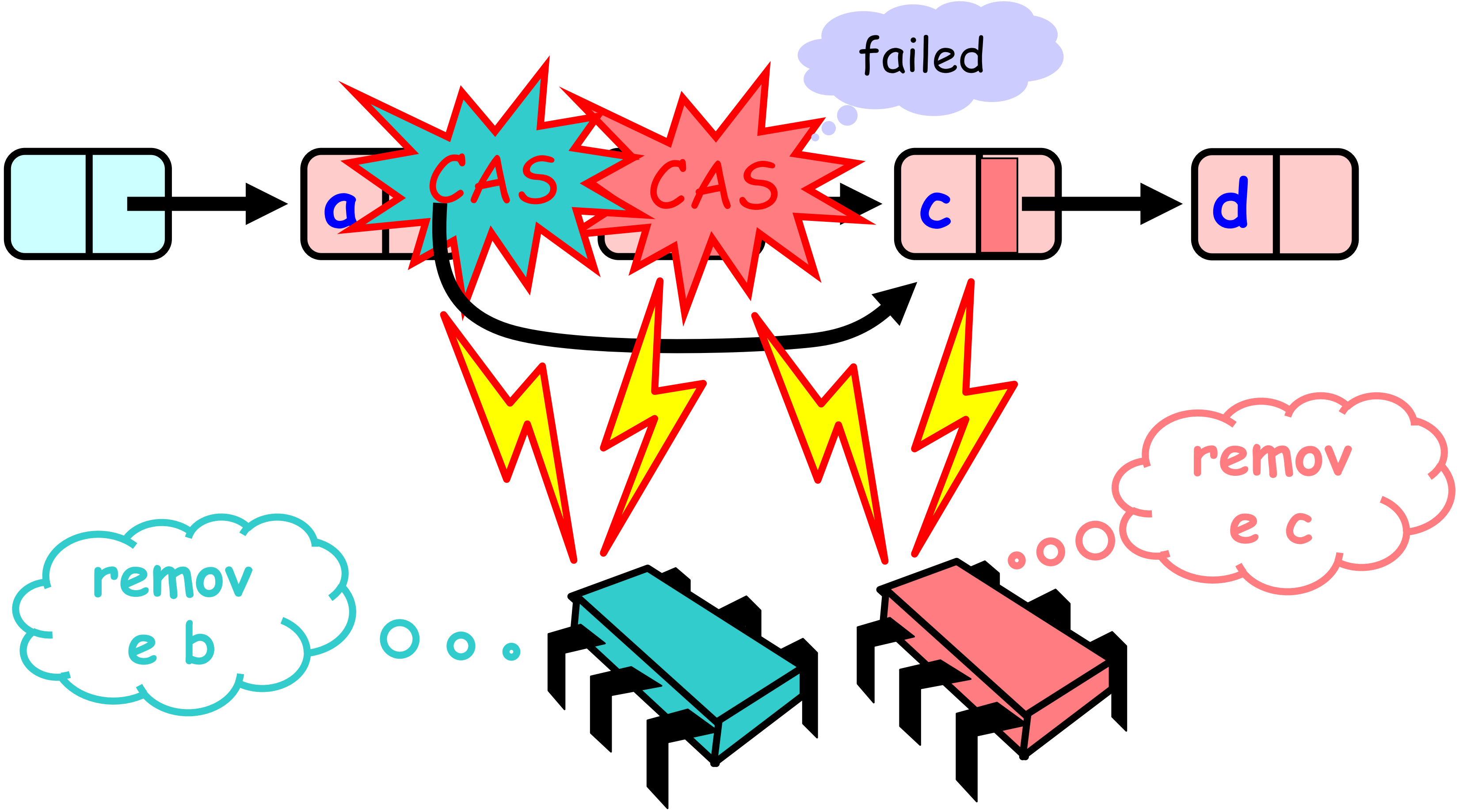
```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

… and this new mark

Removing a Node

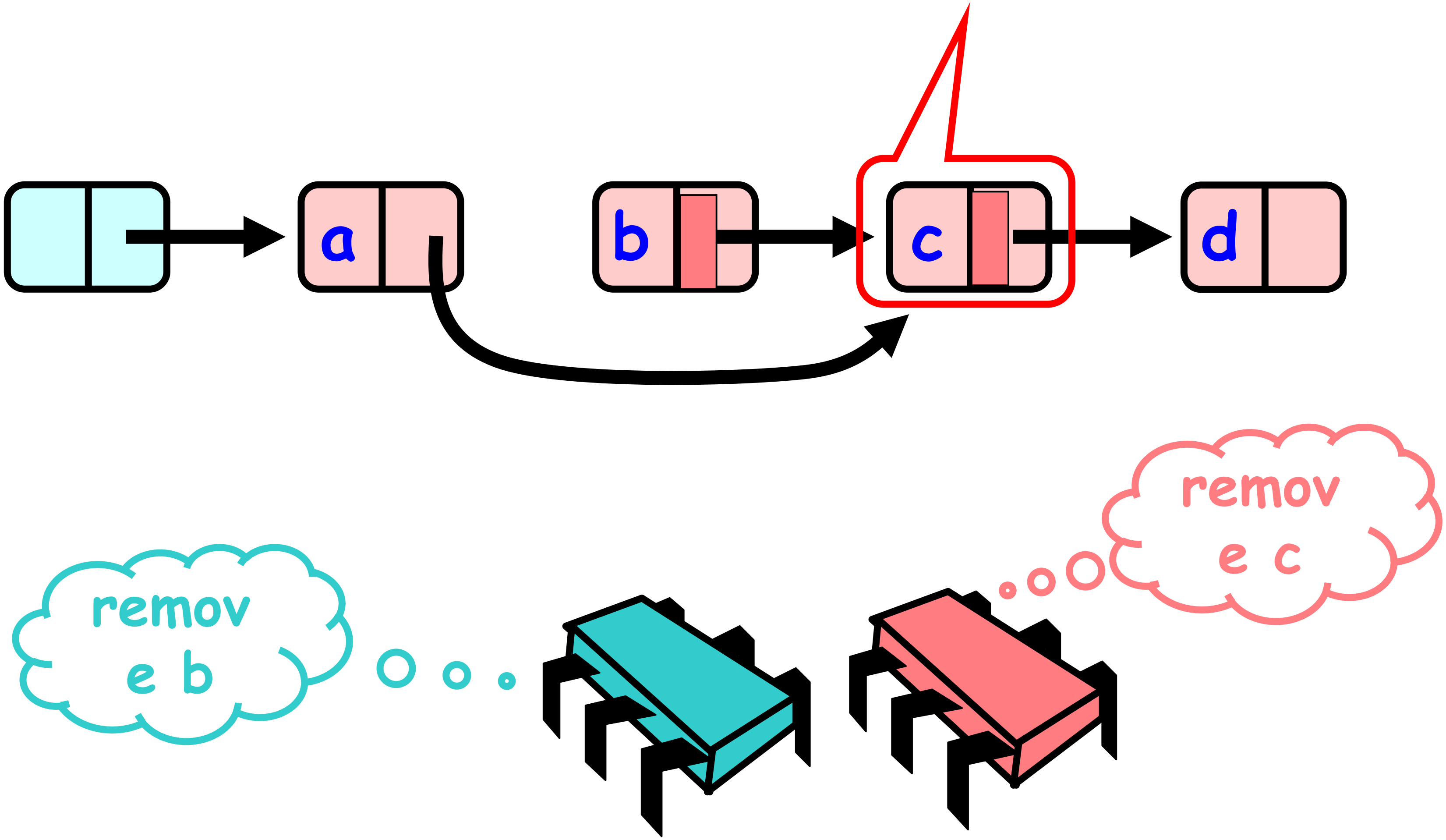


Removing a Node



Removing a Node

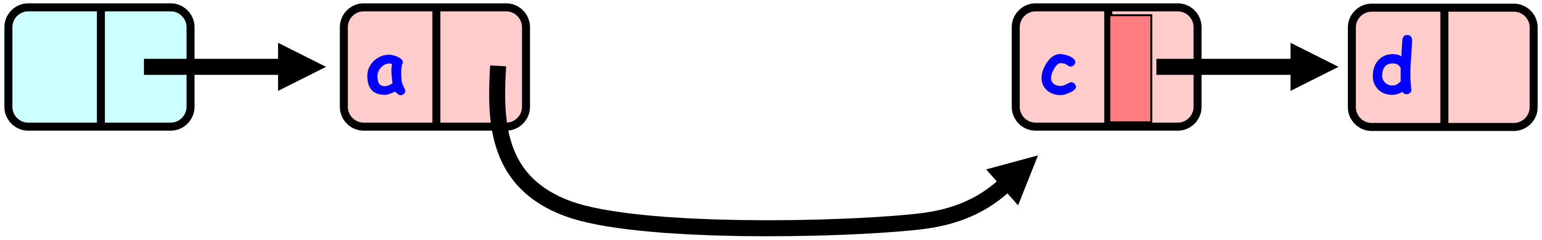
But c is still in the list…
Is it?

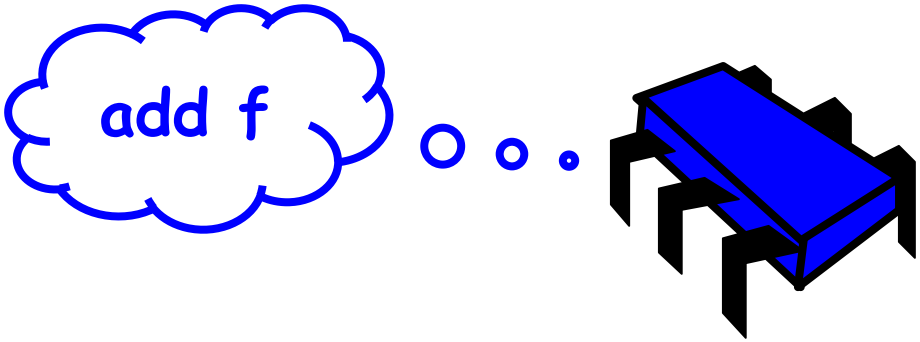


Non-blocking synchronization

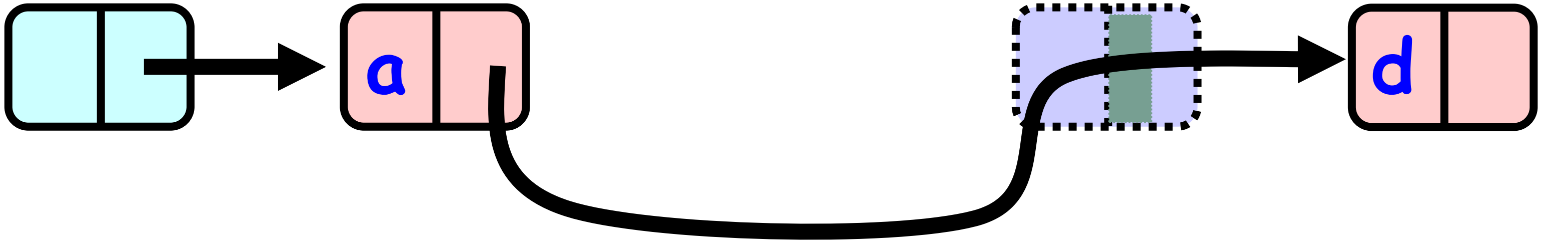
- The physical removal is shared by all threads calling add() or remove()
  - As each thread traverses the list, it cleans up the list by physically removing any marked nodes it encounters
- Contains does not remove any nodes but traverses all nodes whether they are marked or not

Removing a Node

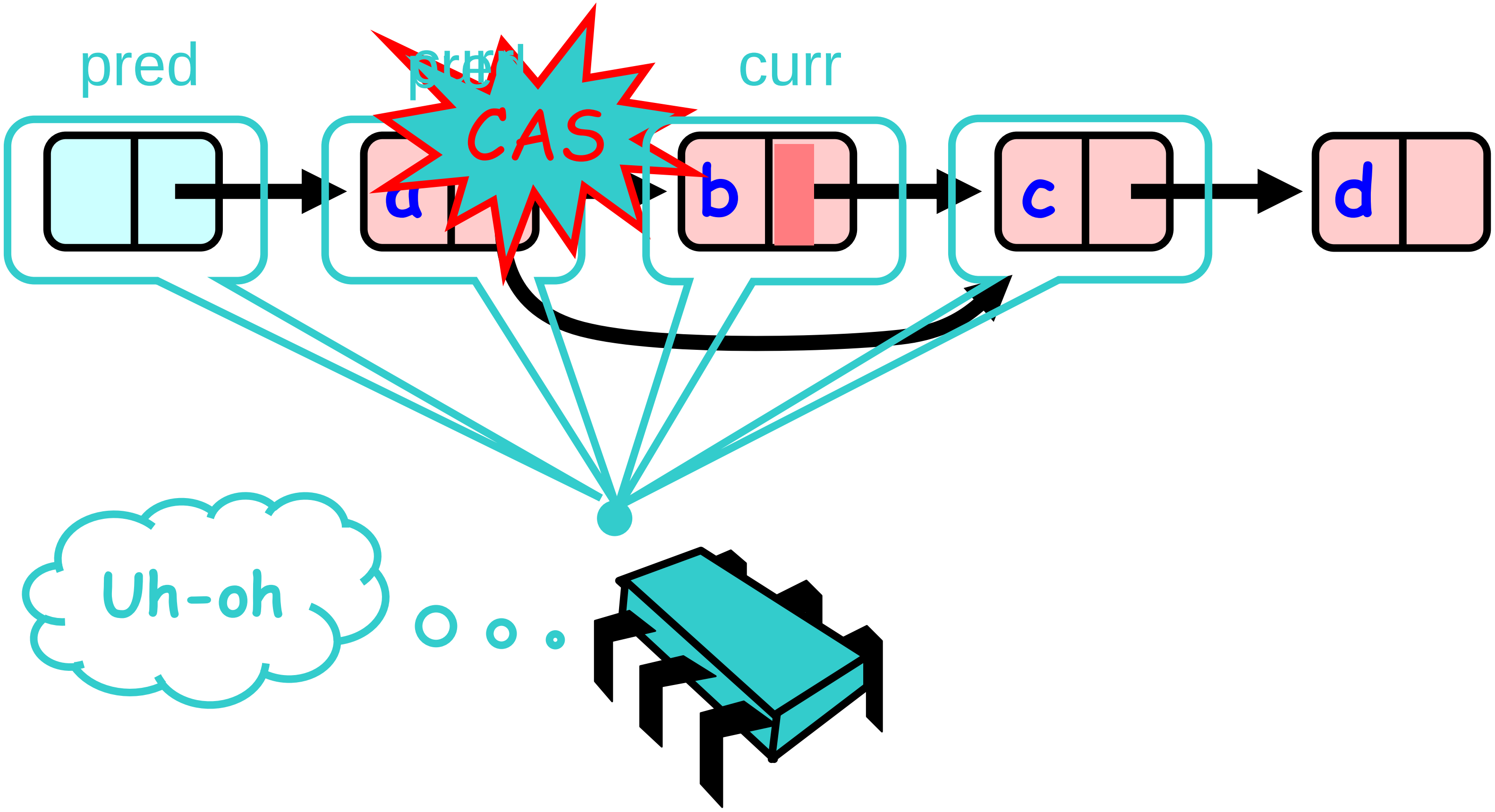




Removing a Node



Lock-Free Traversal (only Add and Remove)



Non-blocking synchronization

- Why can other threads not simply traverse the list without removing marked nodes?
- Why are nodes not removed directly after marking them?

The Window find() method

```
public Window find(Node head, int key)
{
  boolean[] marked = {false};
  boolean snip;

  retry: while (true) {
     pred = head;
     curr = pred.next.getReference();
     while (true) {
          succ = curr.next.get(marked);
          while (marked[0]) {
               ...
```

The Window find() method

```
               ...
     snip = pred.next.CAS(curr, succ,
     false, false);
     if (!snip)
          continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
if (curr.key >= key)
     return new Window(pred, curr);
pred = curr;
curr = succ;
}
```

add() method

```
public boolean add(T item) {
  int key = item.hashCode();

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

add() method

```
public boolean add(T item) {
  int key = item.hashCode();

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

Traverses list and physical remove items till position found

add() method

```
  if (curr.key == key)
     return false;
 else {
     Node node = new Node(item);
     node.next = new
AtomicMarkableReference(curr, false);

     if (pred.next.compareAndSet(curr,
node, false, false))
          return true;
} }
```

add() method

```
  if (curr.key == key)
     return false;
 else {
     Node node = new Node(item);
     node.next = new
AtomicMarkableReference(curr, false);

     if (pred.next.compareAndSet(curr,
node, false, false))
          return true;
} }
```

Item is already in list

add() method

```
   if (curr.key == key)
      return false;
  else {
      Node node = new Node(item);
      node.next = new
AtomicMarkableReference(curr, false);

      if (pred.next.compareAndSet(curr,
node, false, false))
           return true;
} }
```

Create new node

add() method

```
  if (curr.key == key)
     return false;
 else {
     Node node = new Node(item);
     node.next = new
AtomicMarkableReference(curr, false);

     if (pred.next.compareAndSet(curr,
node, false, false))
          return true;
} }
```

Add item using CAS

remove() method

```
public boolean remove(T item) {
   int key = item.hashCode();
   boolean snip;

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

remove() method

```
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;

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

Traverses list and physical remove items till position found

remove() method

```
  if (curr.key != key)
     return false;
 else {
     Node succ =
     curr.next.getReference();
     snip = curr.next.attemptMark(succ,
     true);

     if (!snip) continue;
     pred.next.compareAndSet(curr,
succ, false, false);
     return true;
}
```

remove() method

```
  if (curr.key != key)
     return false;
 else {
     Node succ =
     curr.next.getReference();
     snip = curr.next.attemptMark(succ,
     true);

     if (!snip) continue;
     pred.next.compareAndSet(curr,
succ, false, false);
     return true;
}
```

Item is not in list

remove() method

```
  if (curr.key != key)
     return false;
 else {
     Node succ =
     curr.next.getReference();
     snip = curr.next.attemptMark(succ,
     true);

     if (!snip) continue;
     pred.next.compareAndSet(curr,
succ, false, false);
     return true;
}
```

Mark item as removed with CAS

remove() method

```
   if (curr.key != key)
      return false;
  else {
      Node succ =
      curr.next.getReference();
      snip = curr.next.attemptMark(succ,
      true);

      if (!snip) continue;
      pred.next.compareAndSet(curr,
succ, false, false);
      return true;
}
```

Try again if CAS failed

remove() method

```
  if (curr.key != key)
     return false;
 else {
     Node succ =
     curr.next.getReference();
     snip = curr.next.attemptMark(succ,
     true);

     if (!snip) continue;
     pred.next.compareAndSet(curr,
succ, false, false);
     return true;
}
```

Attempt physical removal

Performance

- Non-blocking synchronization guarantees progress in the face of arbitrary delays
- At what cost?
  - Support of atomic modification of a reference and a boolean mark has added performance cost
  - As add() and remove() traverse the list they have to do additional cleanup