SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Concurrent Data Structures

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Reading

- [AMP] Chapter 9

Slides based on book material

Data Structures

- Important building blocks
  - Often determine overall (asymptotic) performance
- Parallel program with sequential data structures
  - Will not work for shared data
  - (think two insertions in a shared linked list at same time)

SHARED DATA:

```
Queue q;
```

THREAD 1:

```
while(true) {
  d = read_data();
  q.insert(d);
}
```

THREAD 2:

```
while(true) {
  d = q.remove();
  process(d);
}
```

Data Structures

- Important building blocks
  - Often determine overall (asymptotic) performance
- Parallel program with sequential data structures
  - Will not work for shared data
  - (think two insertions in a shared linked list at same time)
- Simply attaching a lock around every data structure operation
  - Doesn't scale
  - Program will be bottlenecked by data structure access

SHARED DATA:

```
Queue q;
```

THREAD 1:

```
while(true) {
  d = read_data();
  lock();
  q.insert(d);
  unlock();
}
```

THREAD 2:

```
while(true) {
  lock();
  d = q.remove();
  unlock();
  process(d);
}
```

Concurrent Data Structures

- Fairly standard approaches to design “efficient” concurrent data structures
  - Bag of tricks

- Coarse-Grained Synchronization
- Fine-Grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- Non-Blocking Synchronization

Measuring Performance

- Goal: Design concurrent data structures
  - Enable concurrent execution of operations
- Operation times are small (often few milliseconds)
  - Workload dependent
  - Affected by runtime variations
- We'll measure throughput (operations per second)
  - How throughput scales as parallelism increases?
  - Expectation: throughput increases with parallelism

List-Based Sets

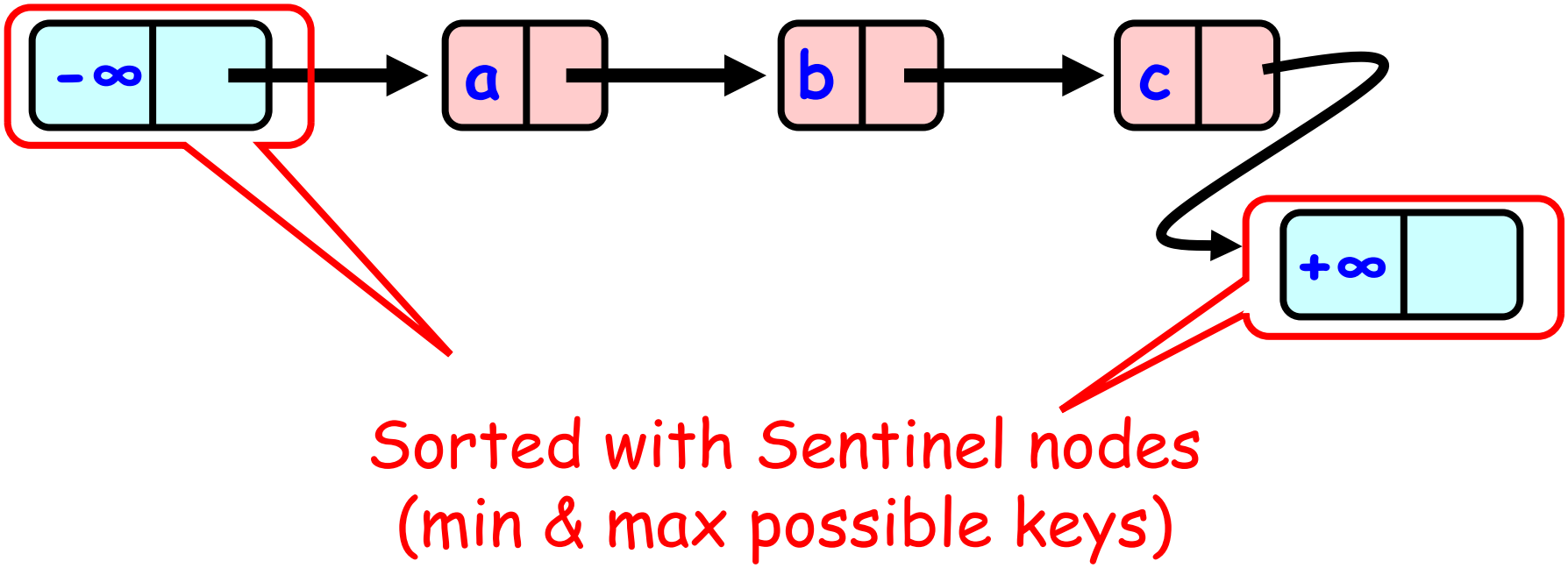
- Linked list that represents/models a set
- Set interface
  - Unordered collection of items
  - No duplicates

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

List-Based Sets

- Sorted linked list as our concrete representation

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



Invariants

- Invariants for set interface
  - Properties of set that are always maintained across all methods
- Representation invariant
  - Which concrete values meaningful?
  - Characterizes legal concrete representations
  - Maintained across all methods
- Representation invariant can be different for different concrete representation

Concrete Representation Choices

- Choice 1:
  - Sorted linked list
  - No duplicates allowed (add shouldn't leave behind > 1 copy)
- Choice 2:
  - Sorted linked list
  - Duplicates allowed (remove should remove all copies)
- Choice 3:
  - Unsorted linked list

  . . .

Representation Invariant

- Sentinel nodes are always present
  - Tail reachable from head
- Linked list should always remain sorted
- Linked list shouldn’t contain duplicates

- Implication on methods:
  - Add shouldn’t insert element if already present
  - Remove/Contains doesn’t need to traverse the entire list

Freedom from Interference

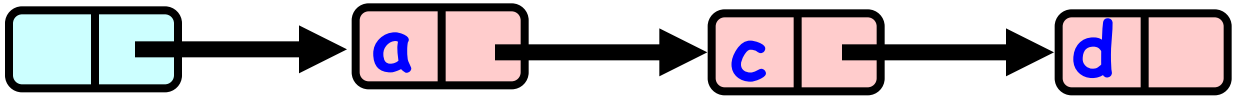
- Methods are the only modifiers of the data structure
- Language encapsulation
  - List nodes not visible outside class

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

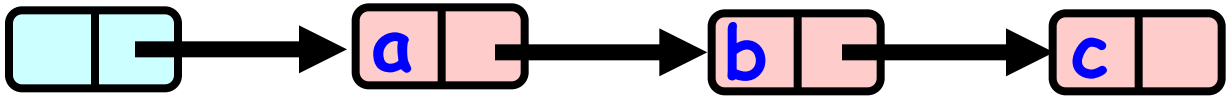
- Important for removed nodes
  - Some algorithms will traverse removed nodes
  - Careful with **malloc()** & **free()**
- Assume garbage-collection will be done

Sequential List Based Set

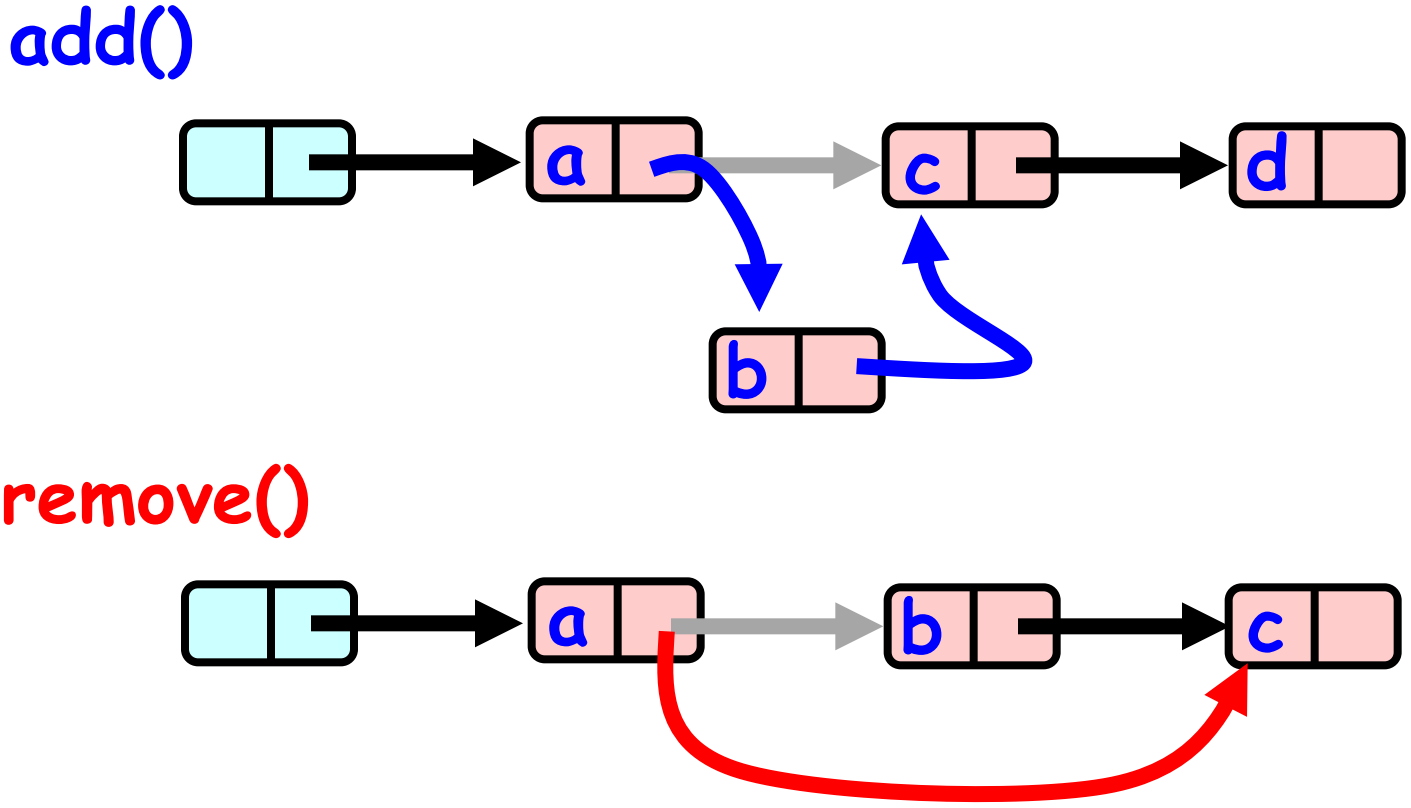
add()



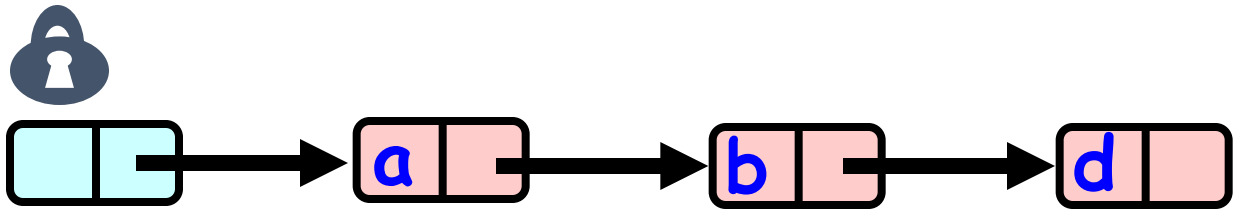
remove()



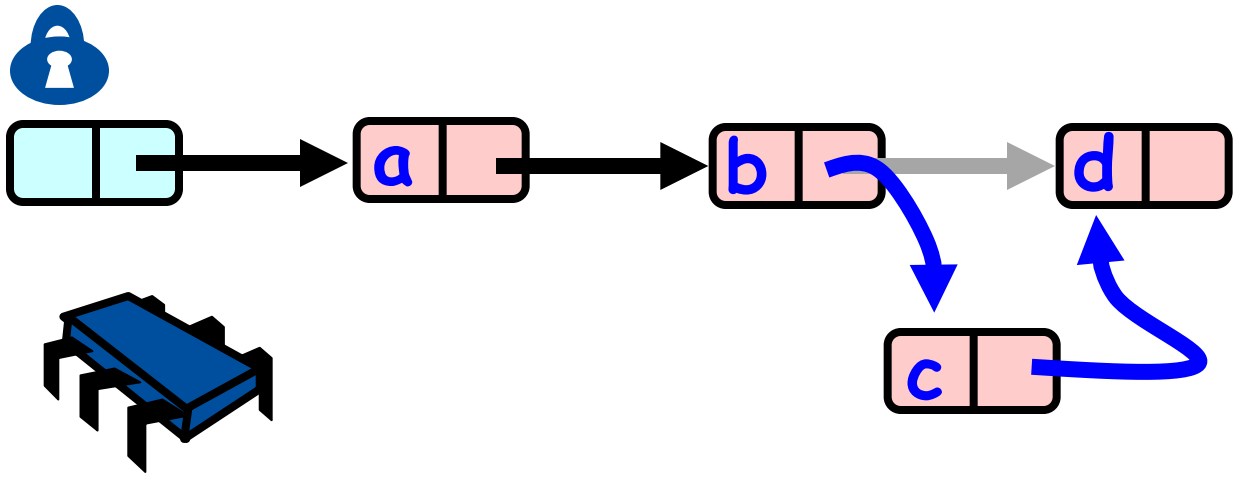
Sequential List Based Set



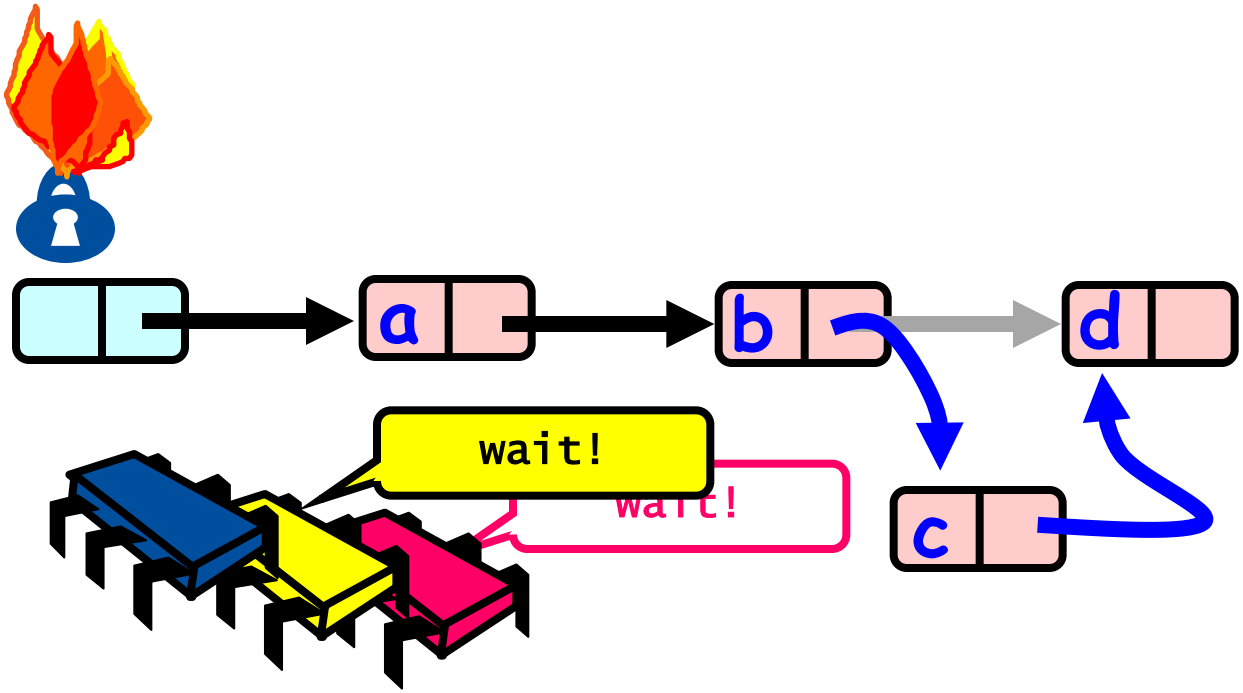
Coarse-Grained Synchronization



Coarse-Grained Synchronization



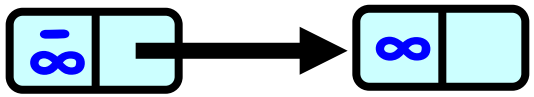
Coarse-Grained Synchronization



Simple but hotspot/bottleneck

Coarse-Grained Synchronization

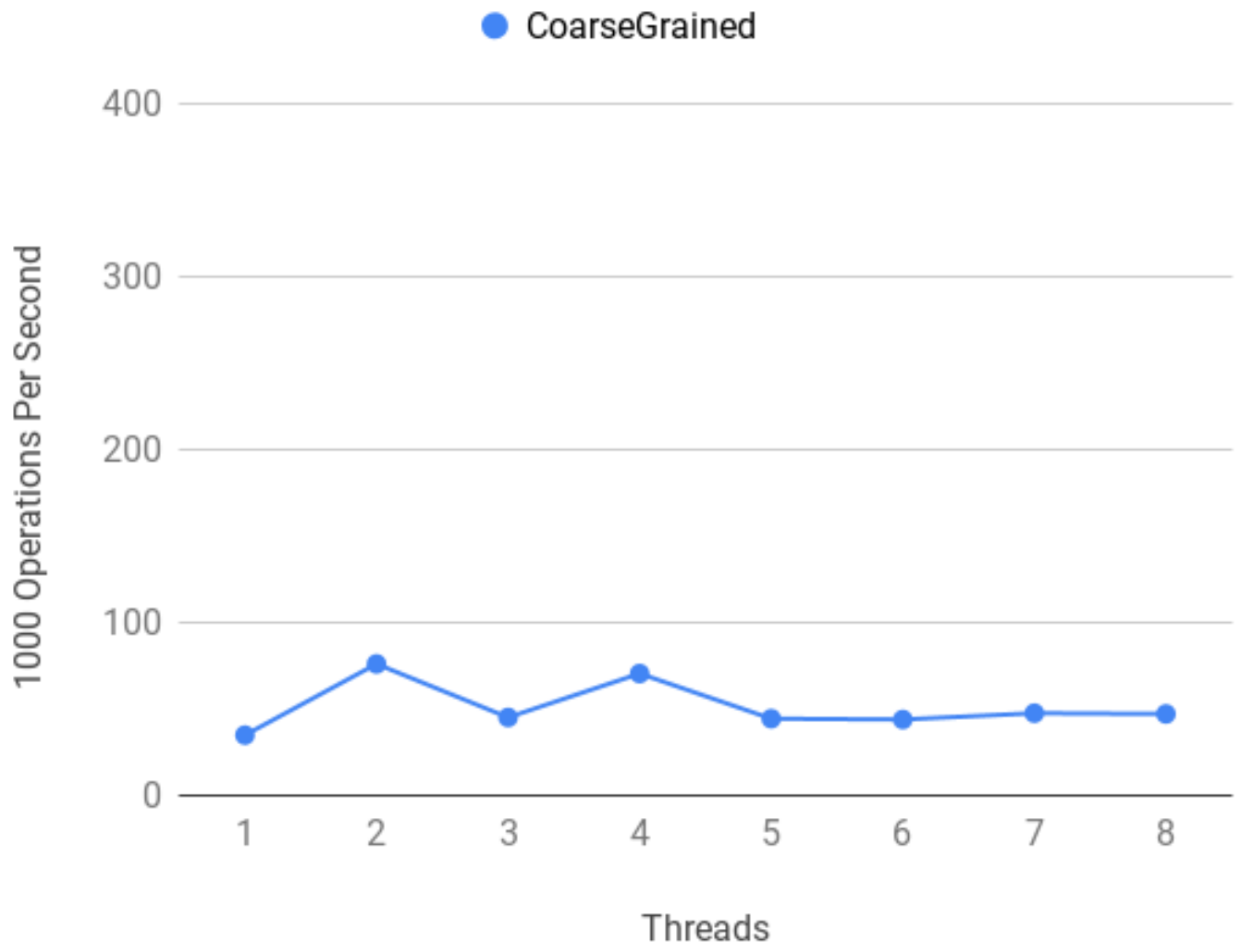
```
1  public class CoarseList<T> {
2    private Node head;
3    private Lock lock = new ReentrantLock();
4    public CoarseList() {
5      head  = new Node(Integer.MIN_VALUE);
6      head.next = new Node(Integer.MAX_VALUE);
7    }
```



```
 8    public boolean add(T item) {
 9      Node pred, curr;
10      int key = item.hashCode();
11      lock.lock();
12      try {
13        pred = head;
14        curr = pred.next;
15        while (curr.key < key) {
16          pred = curr;
17          curr = curr.next;
18        }
19        if (key == curr.key) {
20          return false;
21        } else {
22          Node node = new Node(item);
23          node.next = curr;
24          pred.next = node;
25          return true;
26        }
27      } finally {
28        lock.unlock();
29      }
30    }
```

```
31    public boolean remove(T item) {
32      Node pred, curr;
33      int key = item.hashCode();
34      lock.lock();
35      try {
36        pred = head;
37        curr = pred.next;
38        while (curr.key < key) {
39          pred = curr;
40          curr = curr.next;
41        }
42        if (key == curr.key) {
43          pred.next = curr.next;
44          return true;
45        } else {
46          return false;
47        }
48      } finally {
49        lock.unlock();
50      }
51    }
```

Coarse-Grained Synchronization

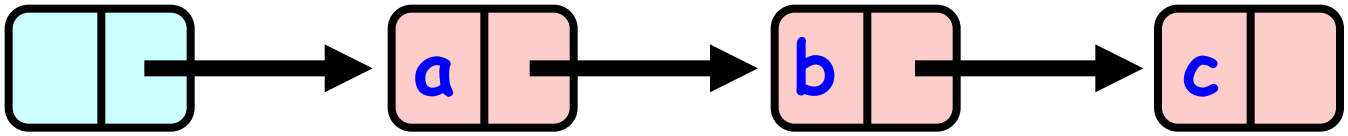


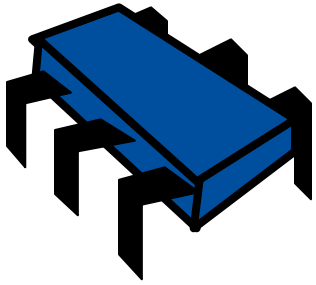
Fine-Grained Synchronization

- Split object into pieces
- Each piece has its own lock
- Methods that work on disjoint pieces do not need to contend with each other

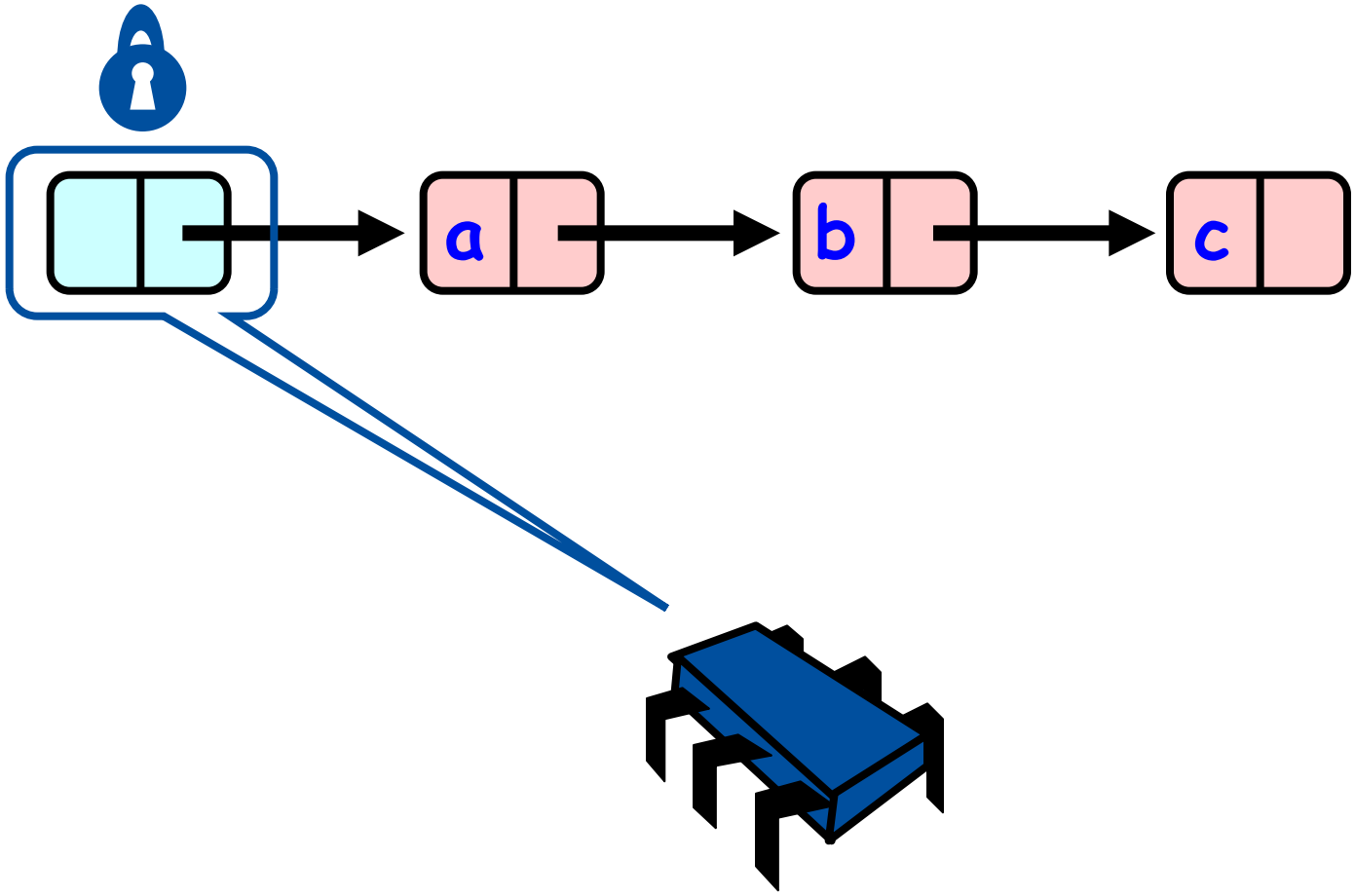
- Add a lock to each node
- As thread traverses, it locks the node before accessing it

Hand-Over-Hand Locking

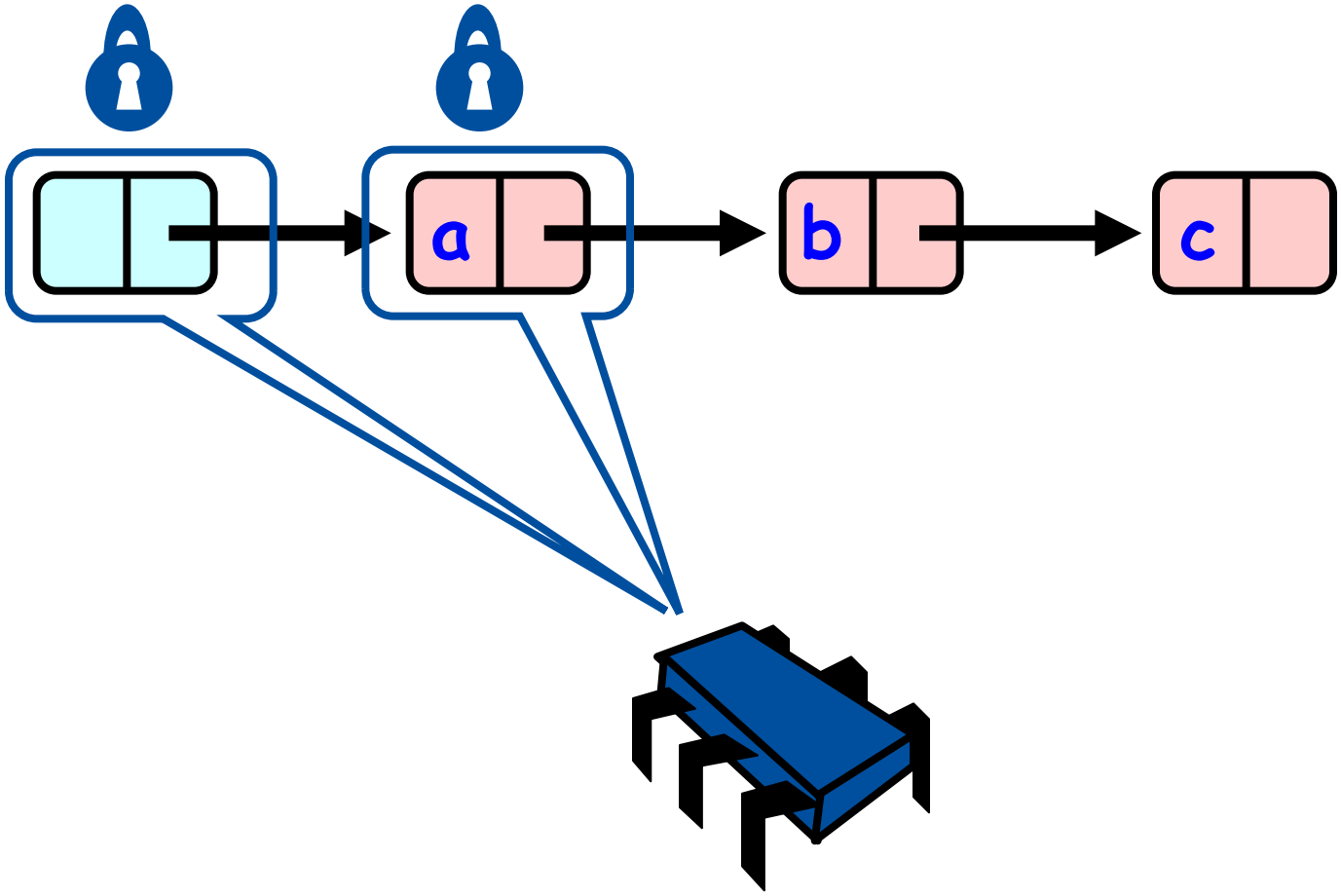




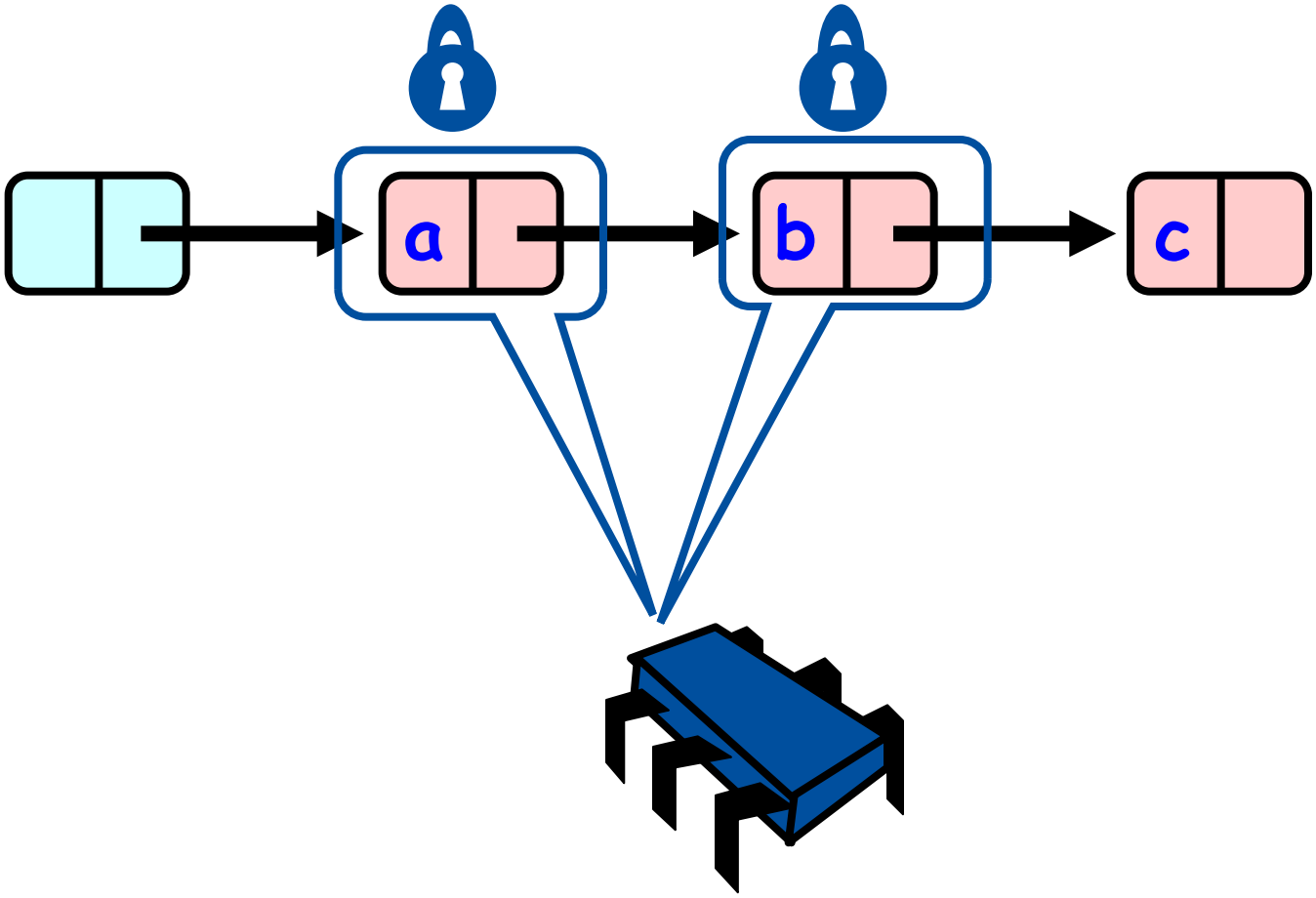
Hand-Over-Hand Locking



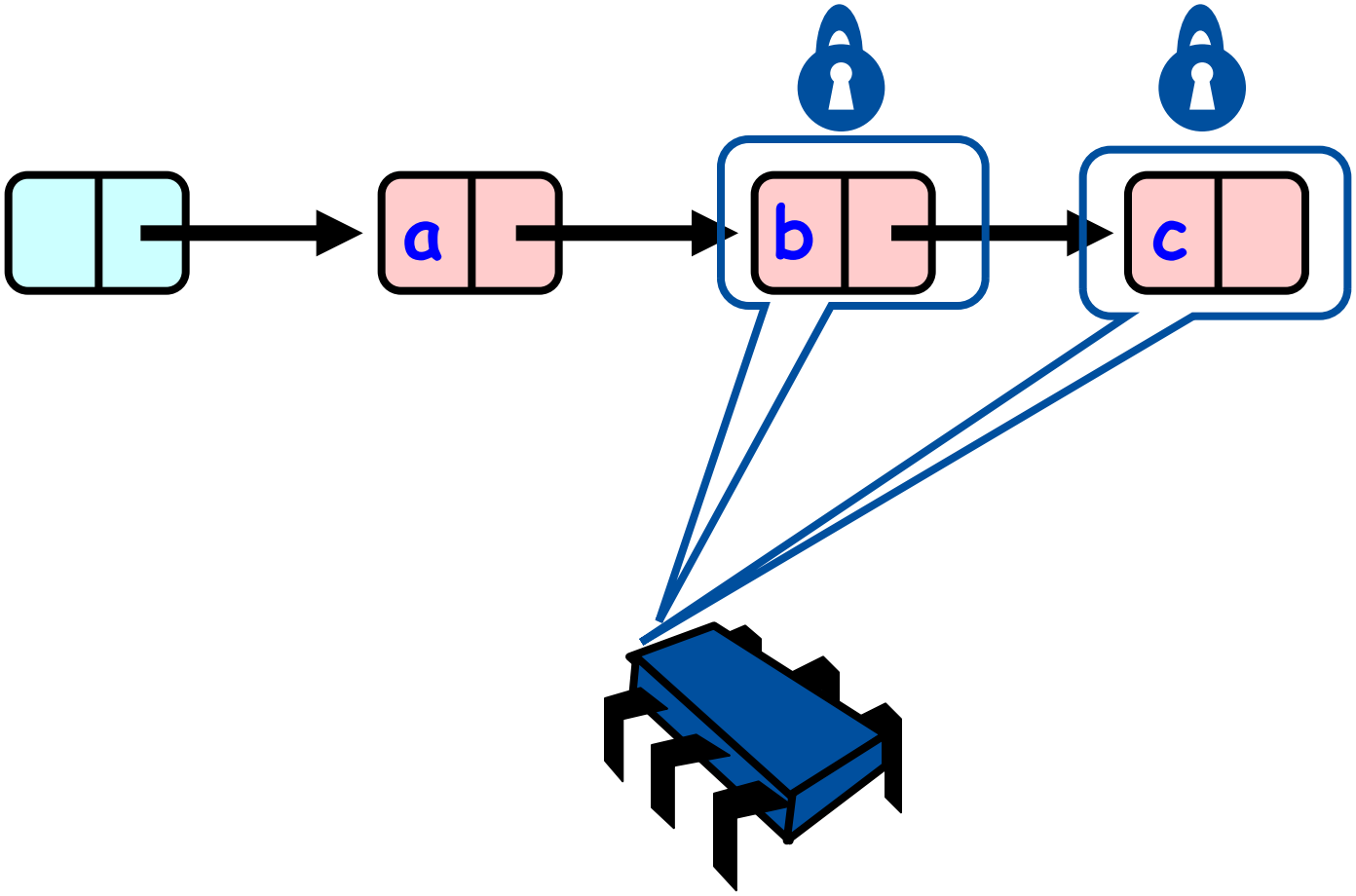
Hand-Over-Hand Locking



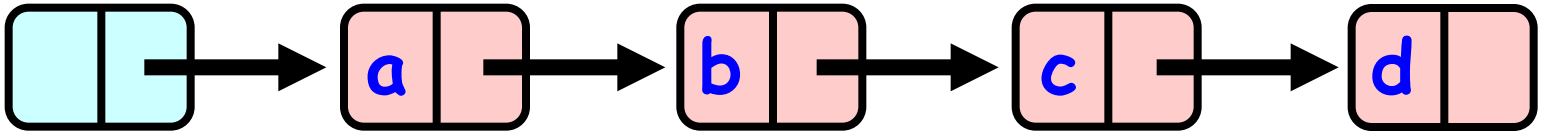
Hand-Over-Hand Locking

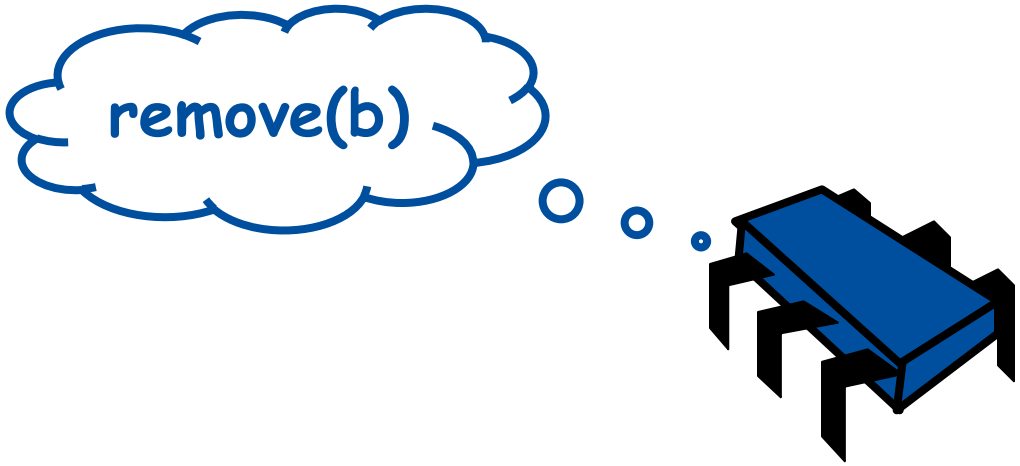


Hand-Over-Hand Locking

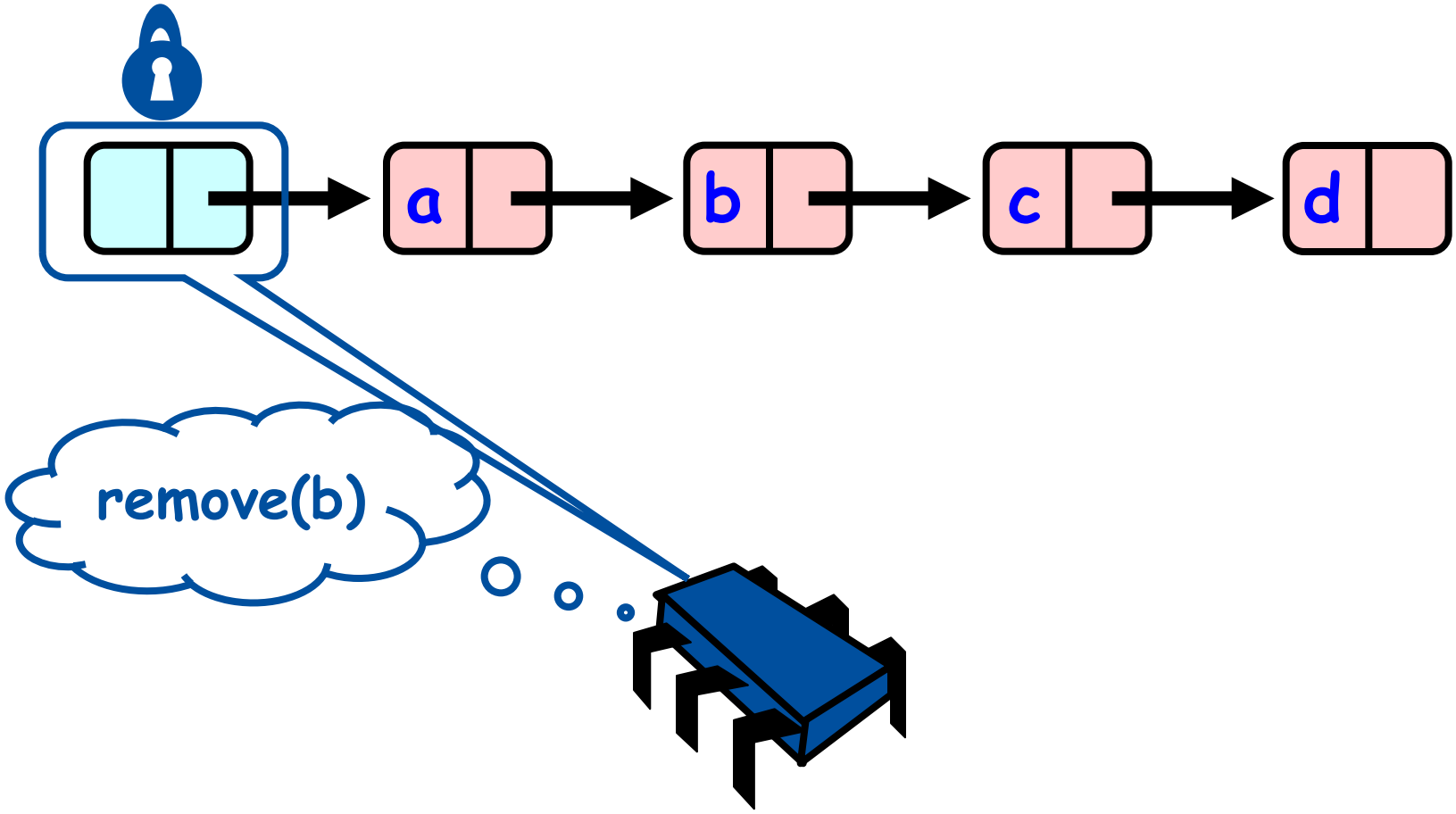


Removing a Node

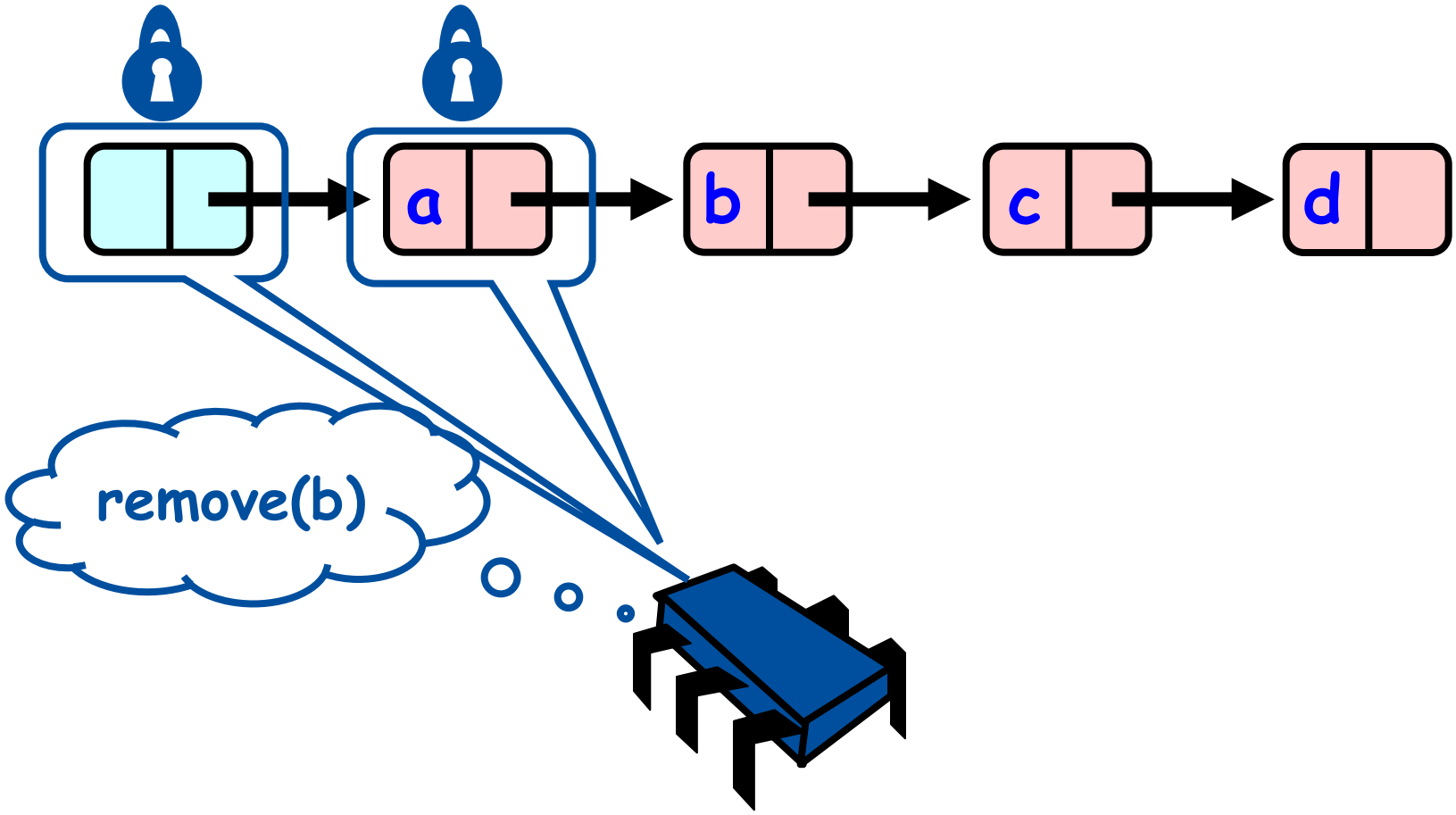




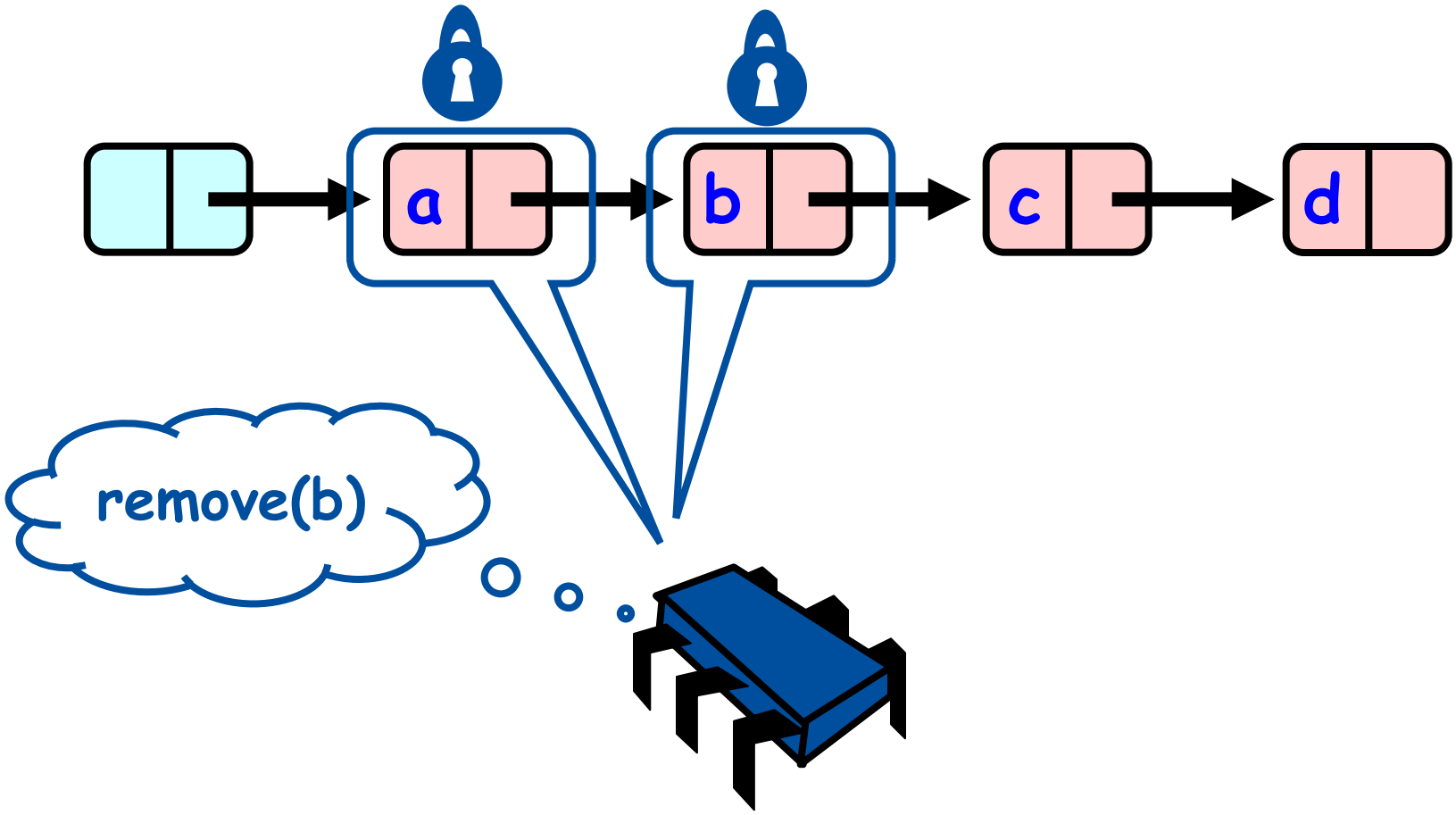
Removing a Node



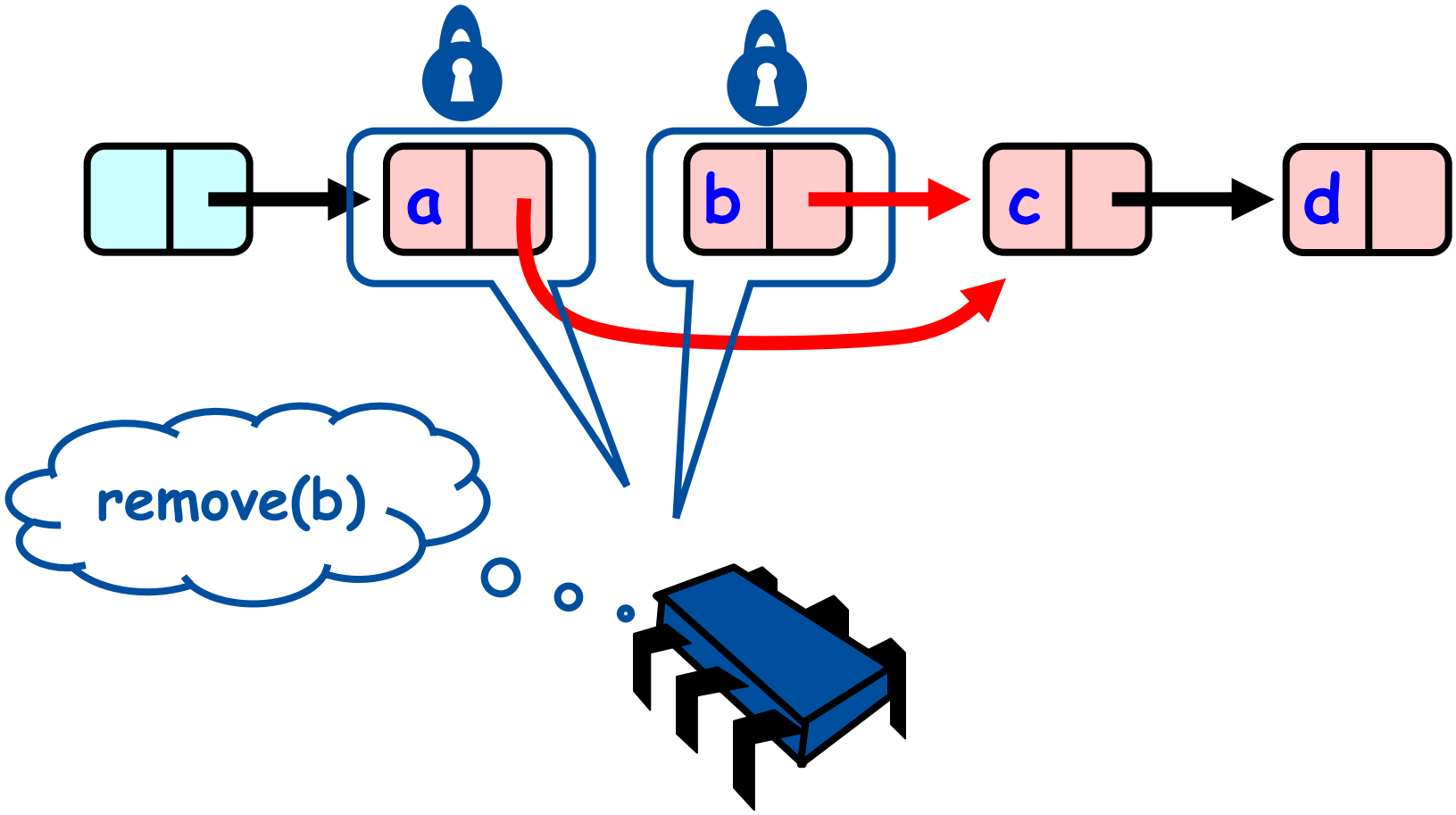
Removing a Node



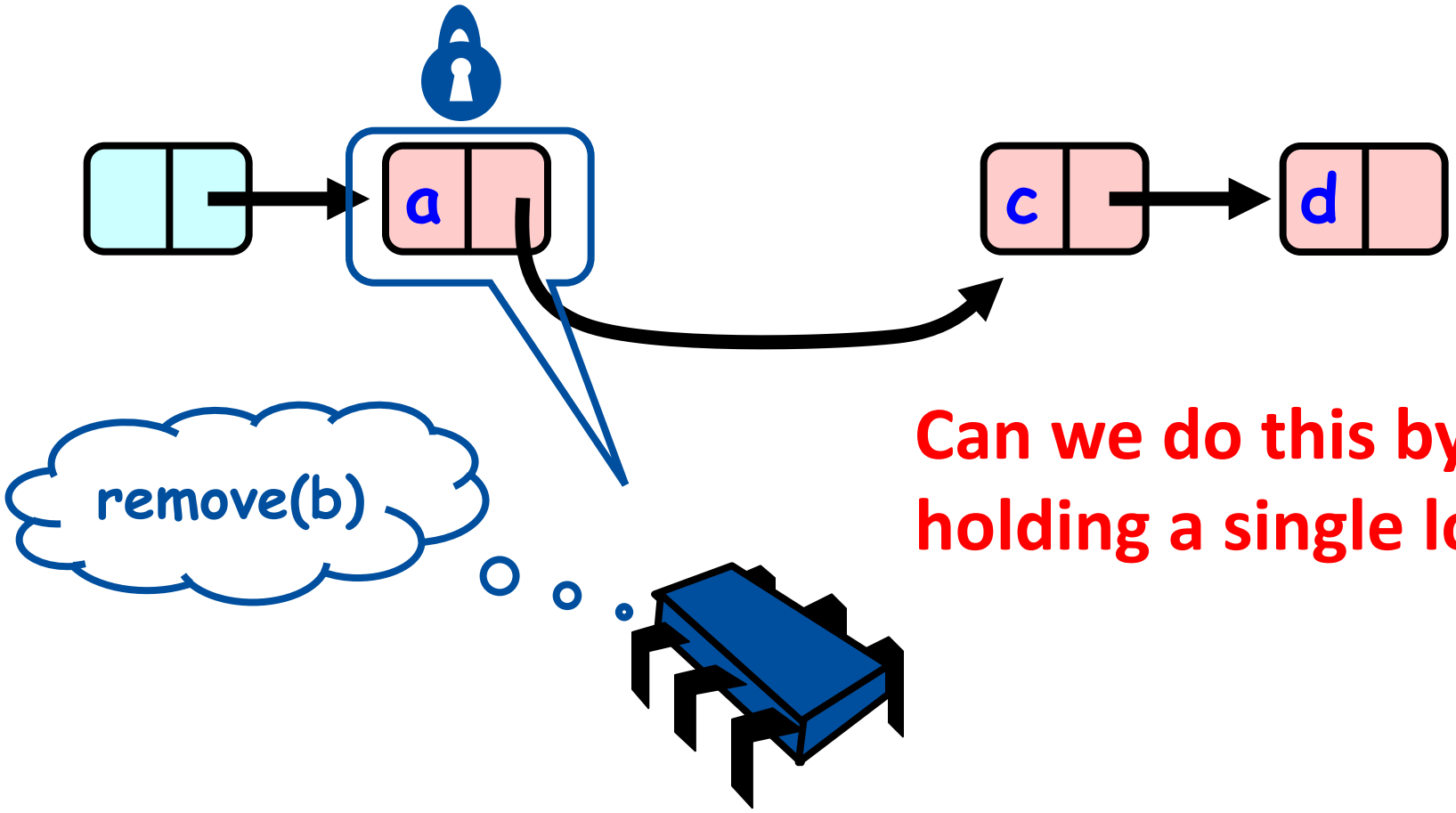
Removing a Node



Removing a Node

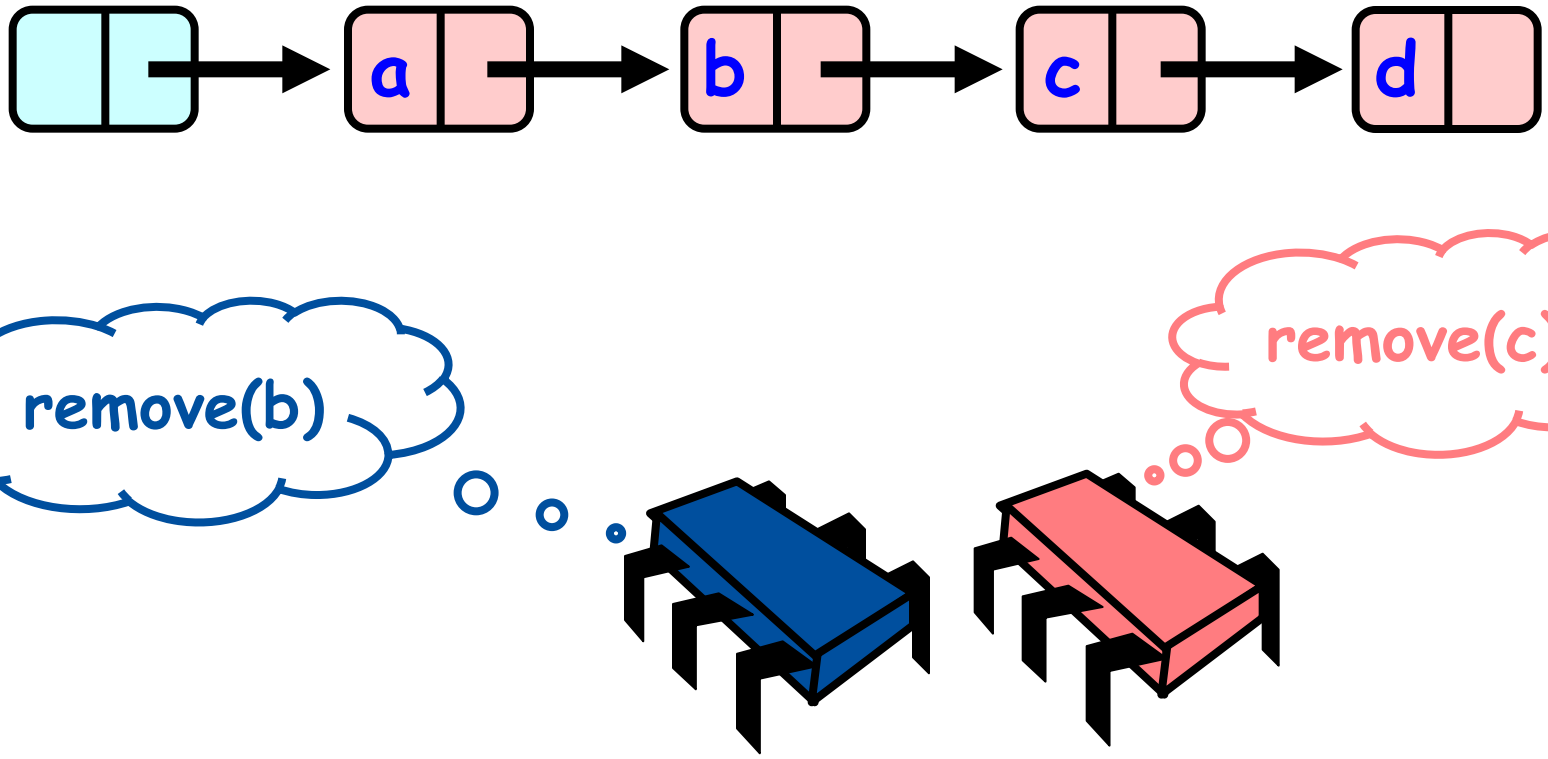


Removing a Node

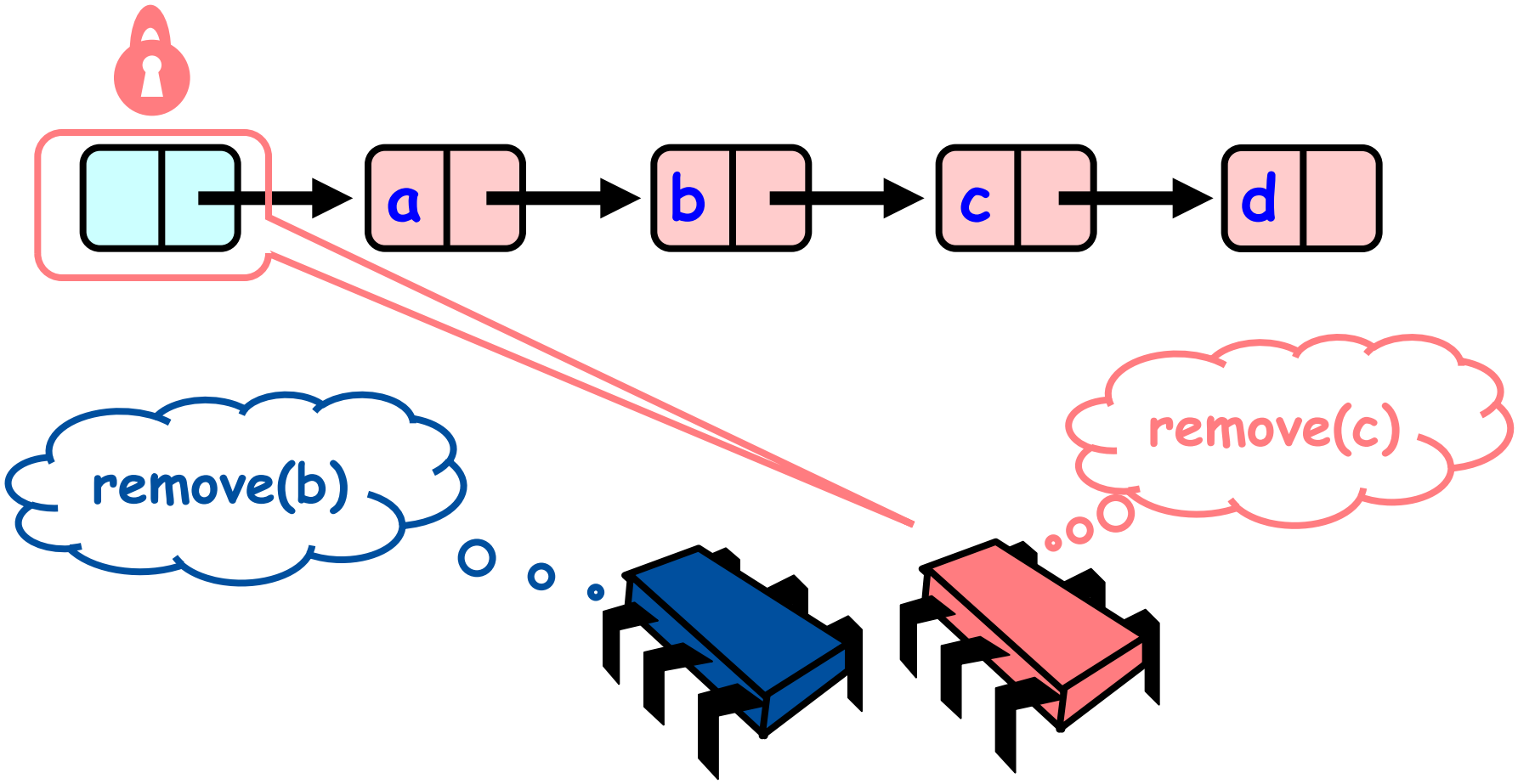


Can we do this by holding a single lock?

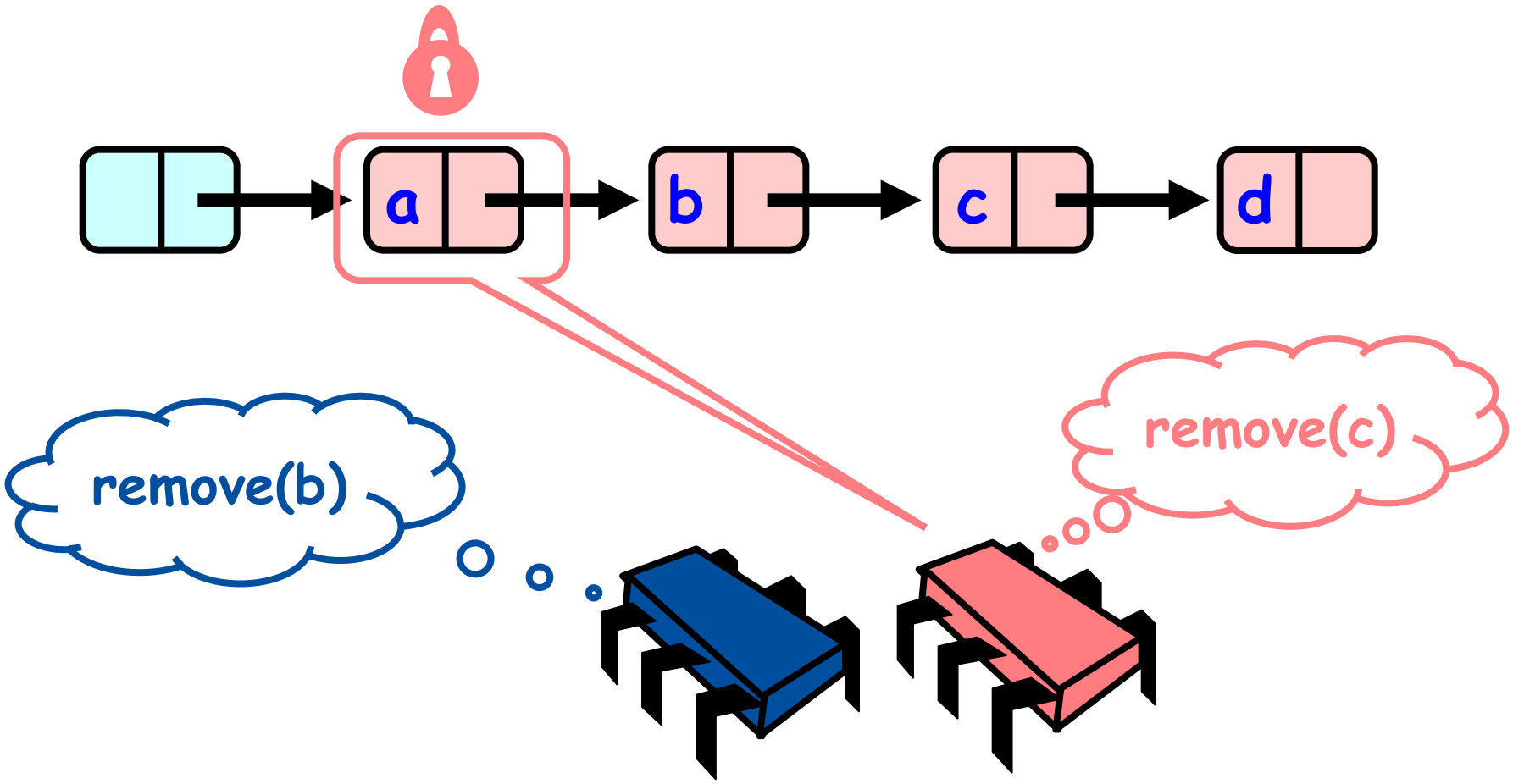
Concurrent Removes



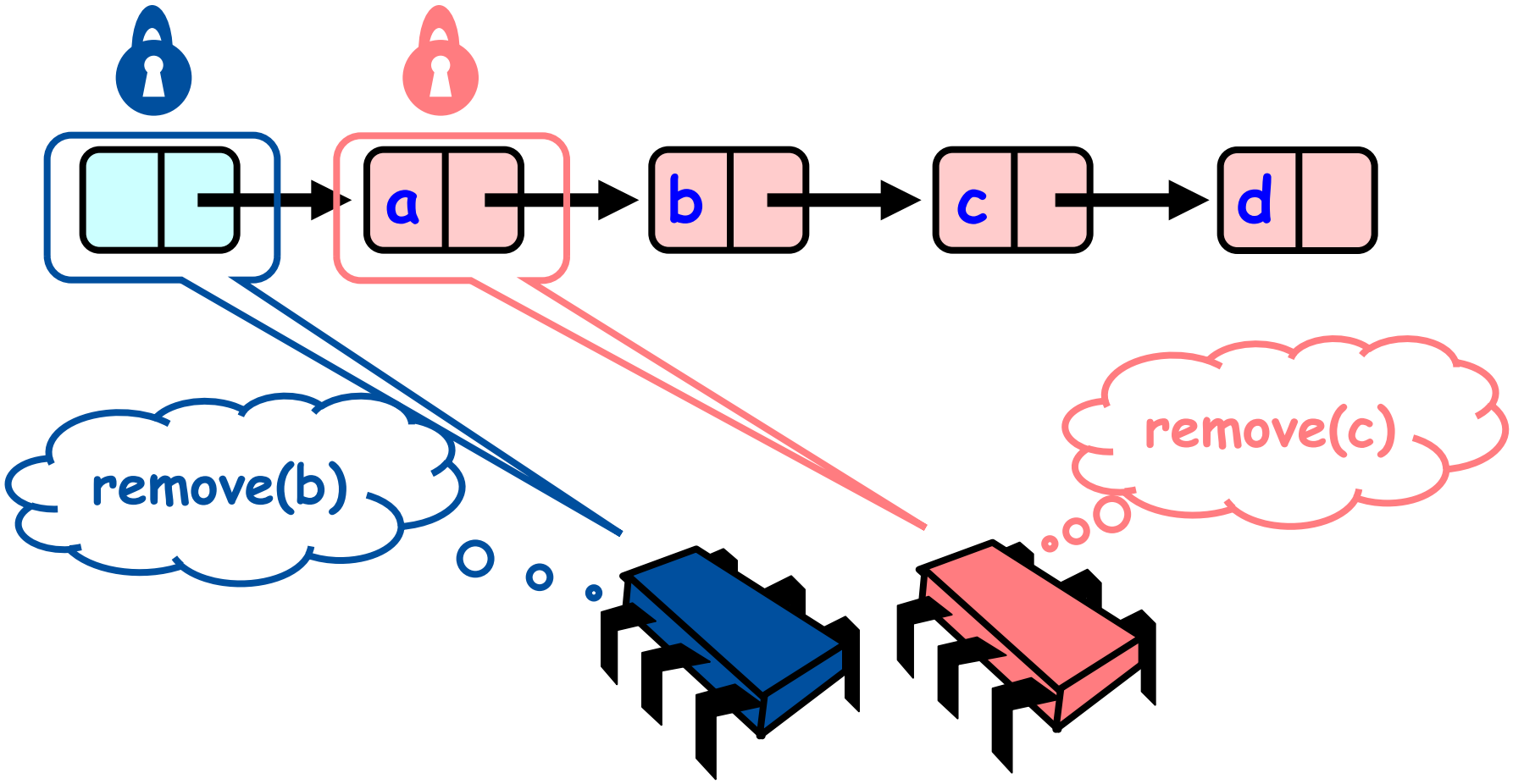
Concurrent Removes



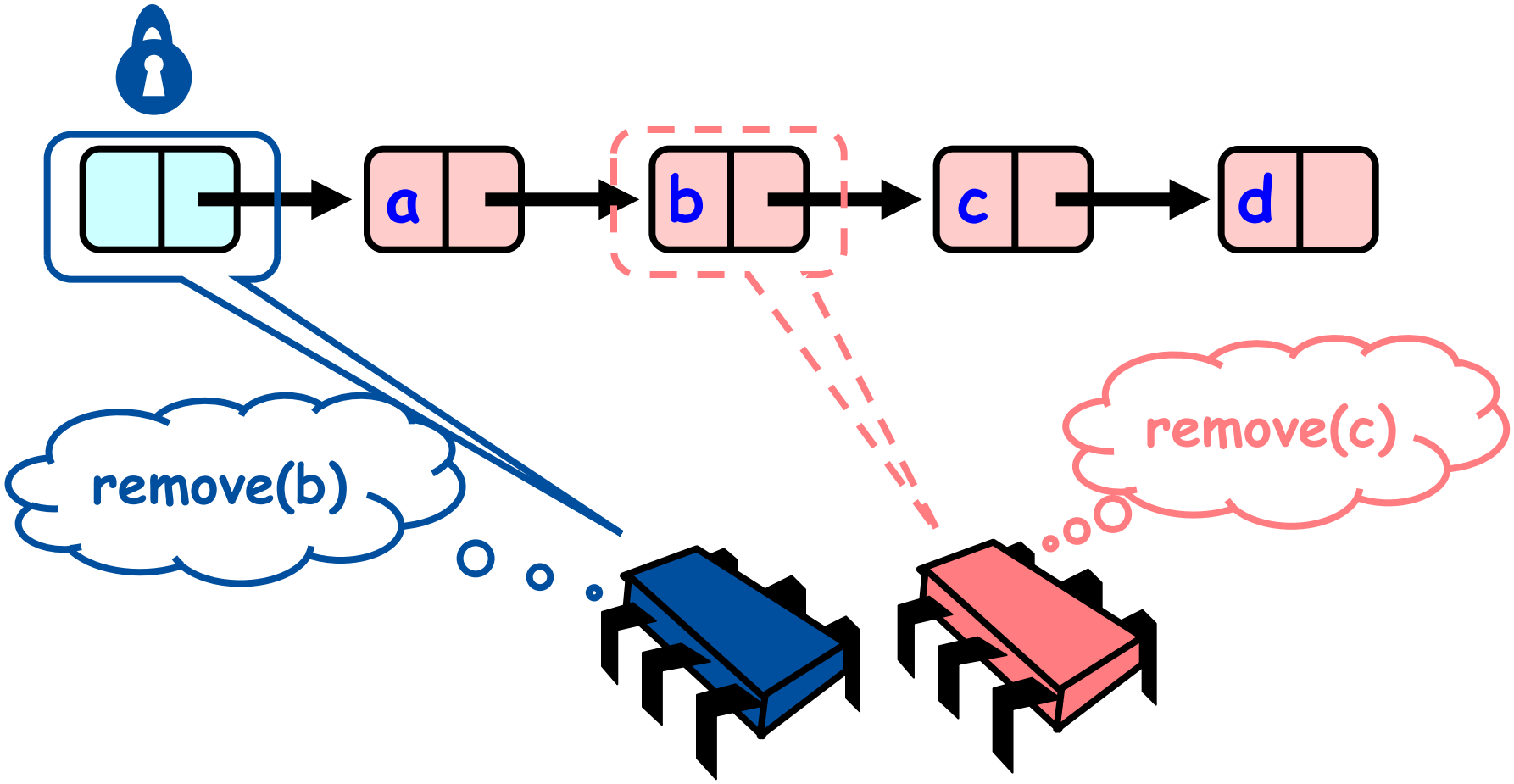
Concurrent Removes



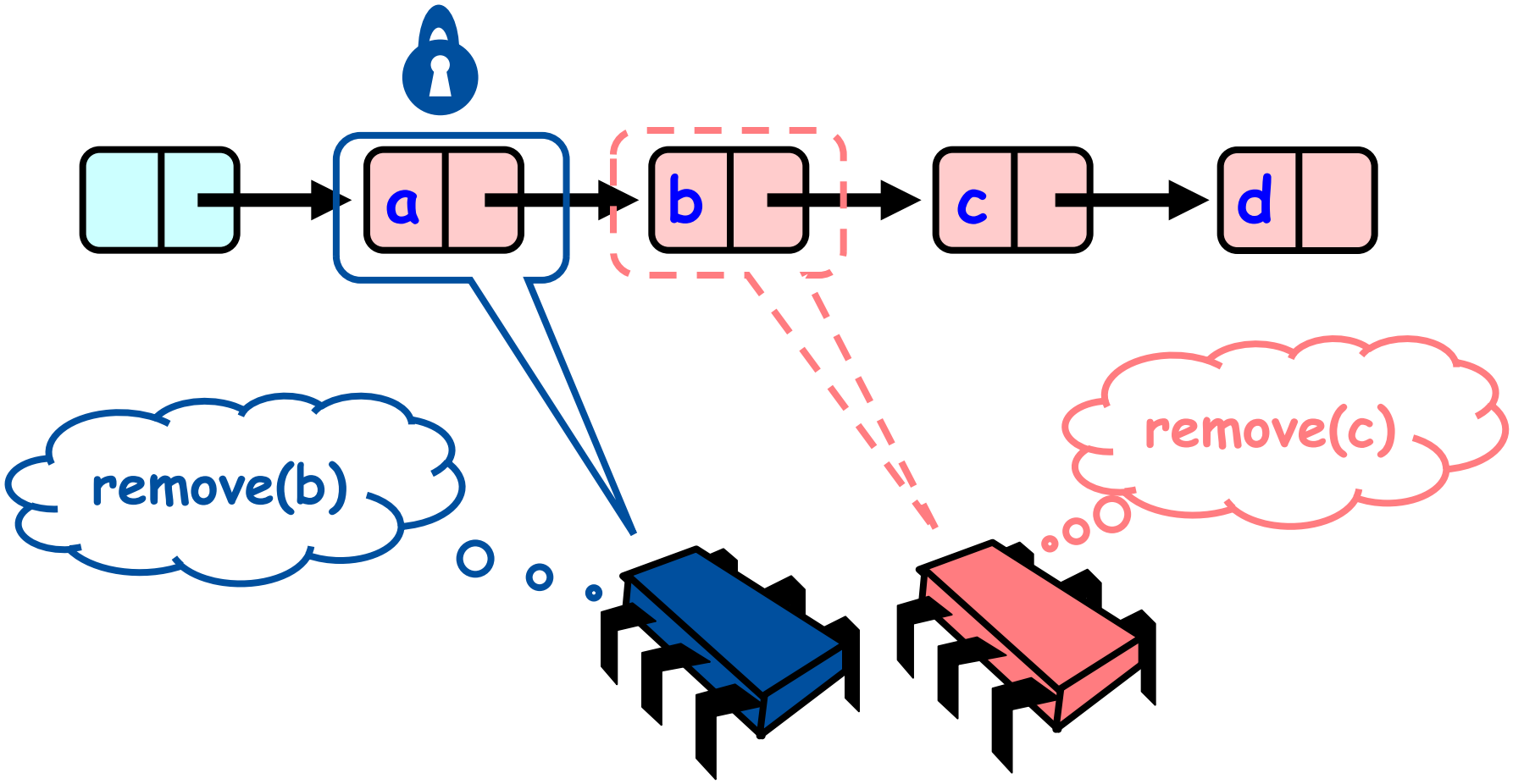
Concurrent Removes



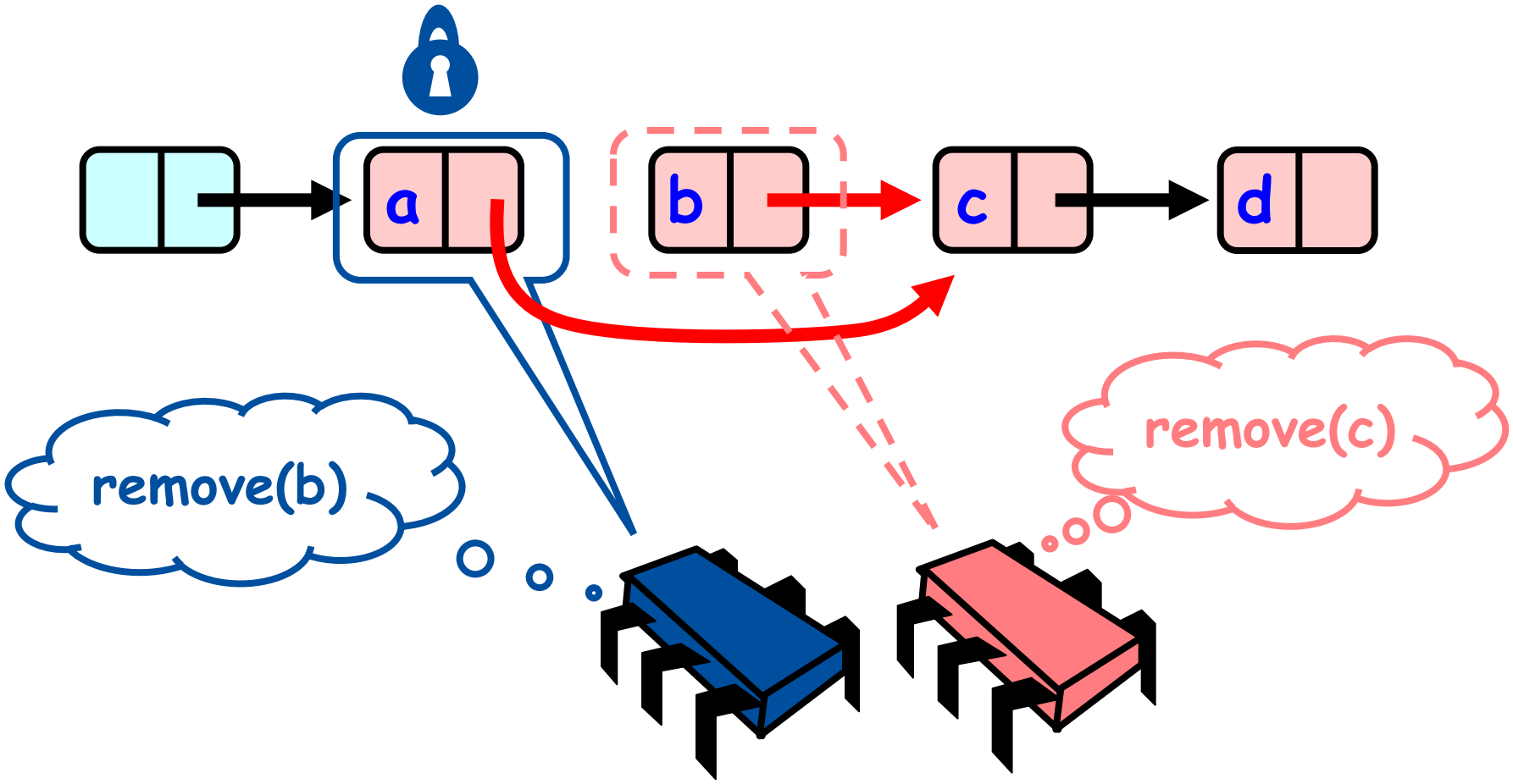
Concurrent Removes



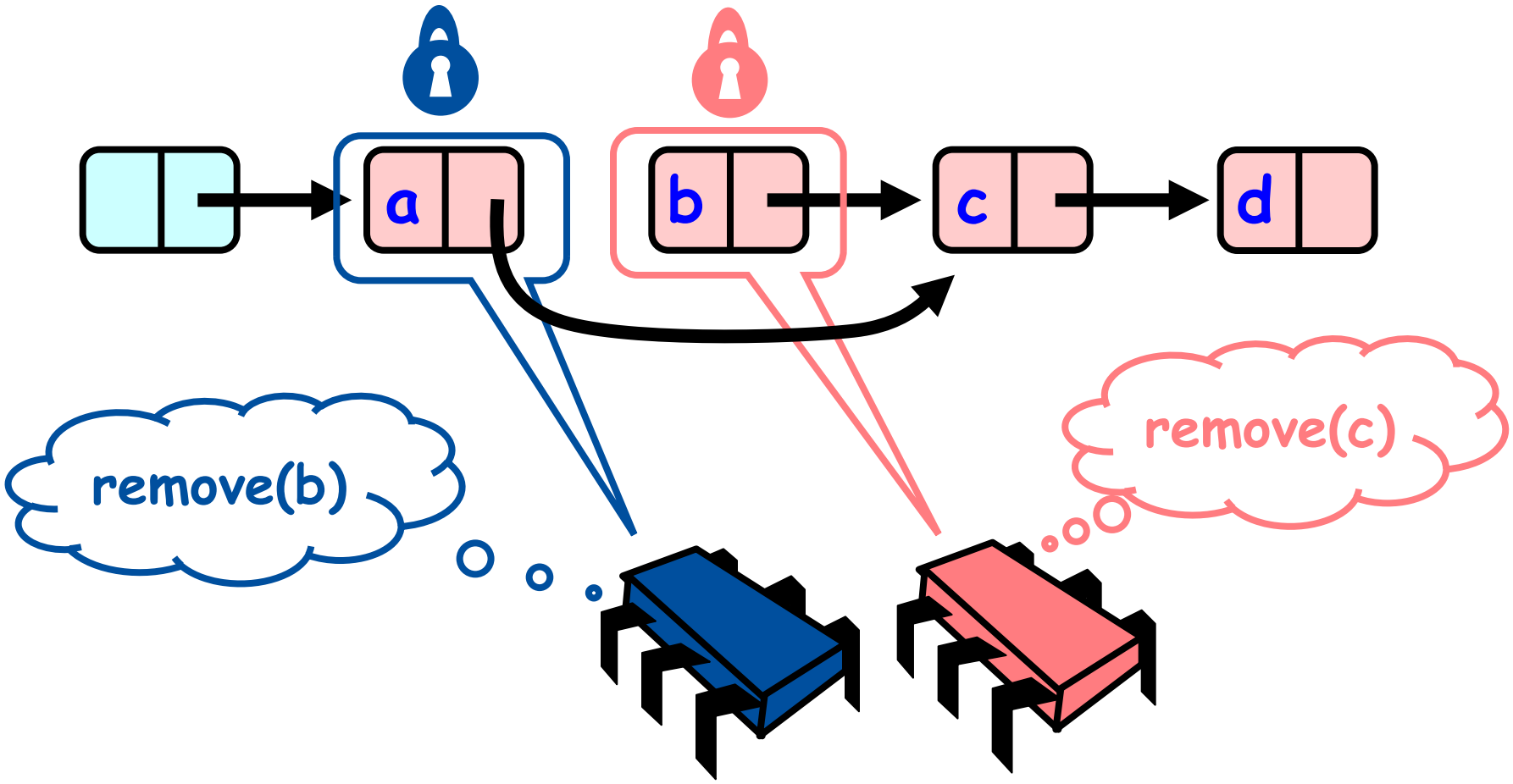
Concurrent Removes



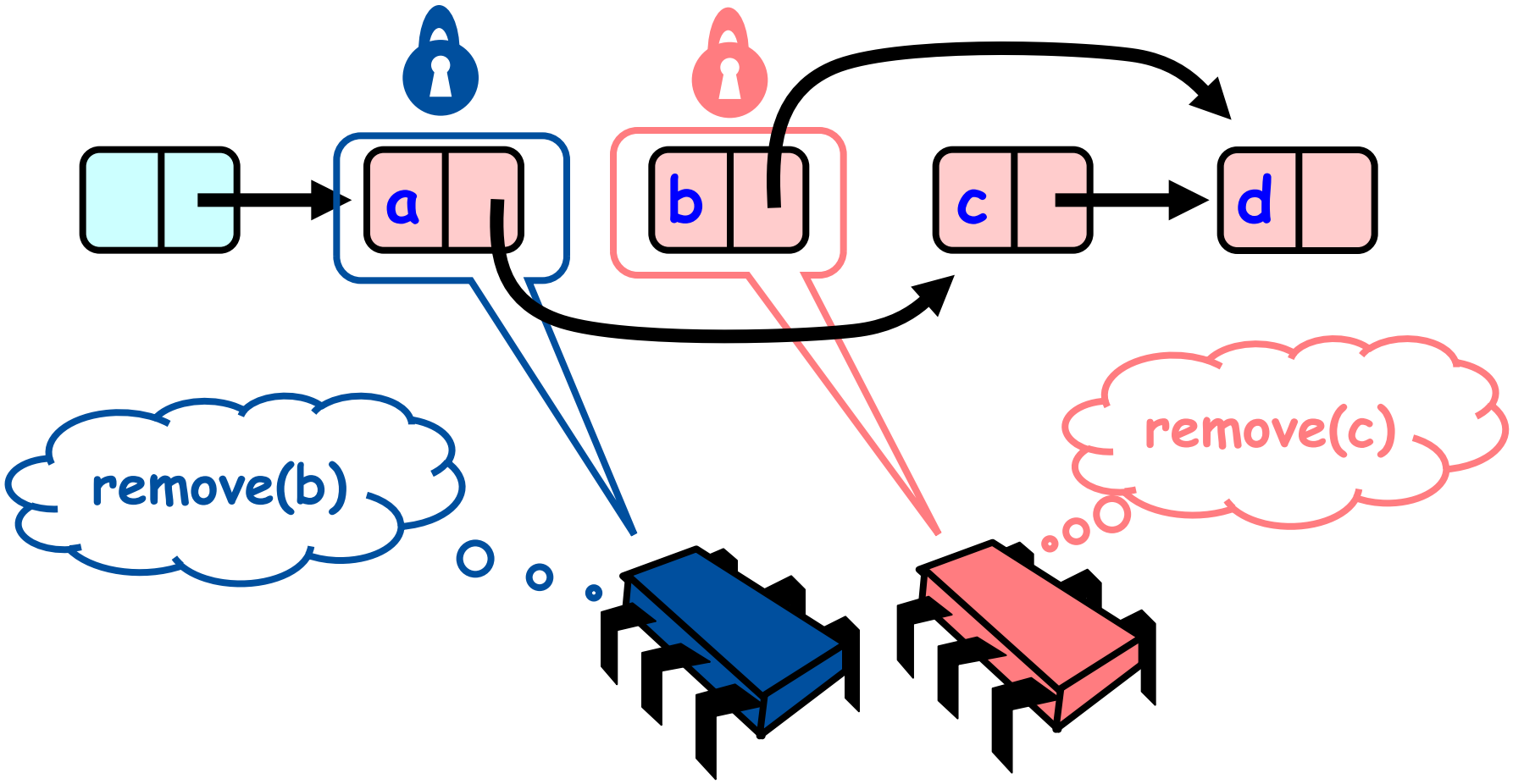
Concurrent Removes



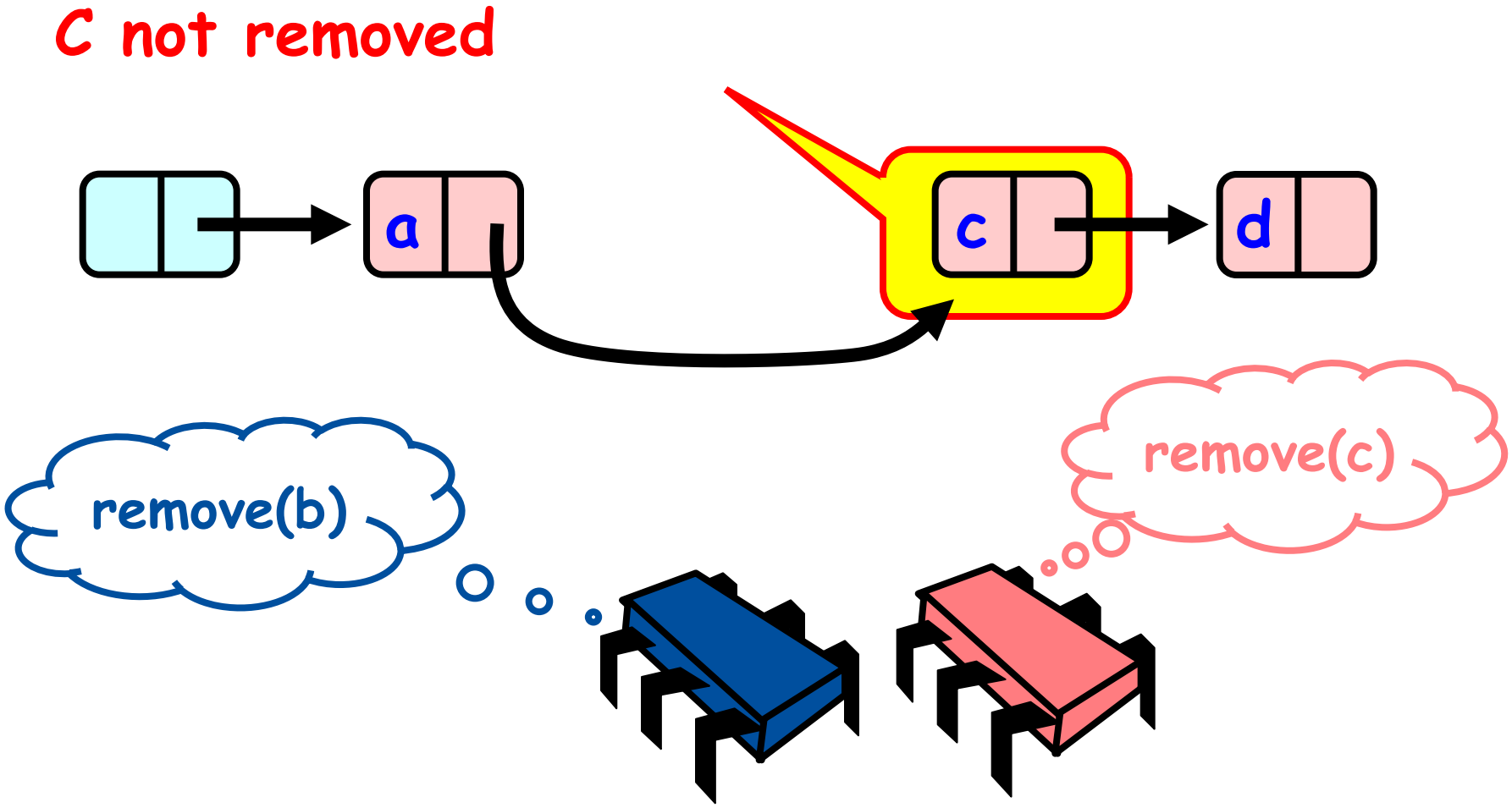
Concurrent Removes



Concurrent Removes



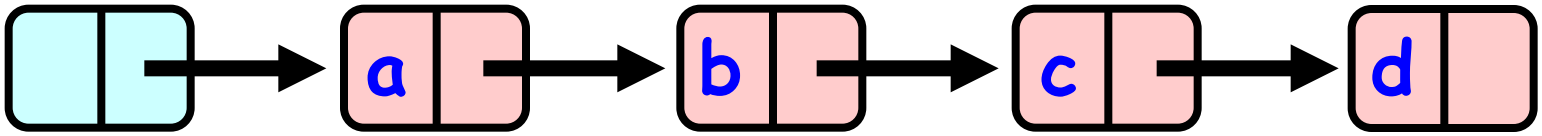
Concurrent Removes

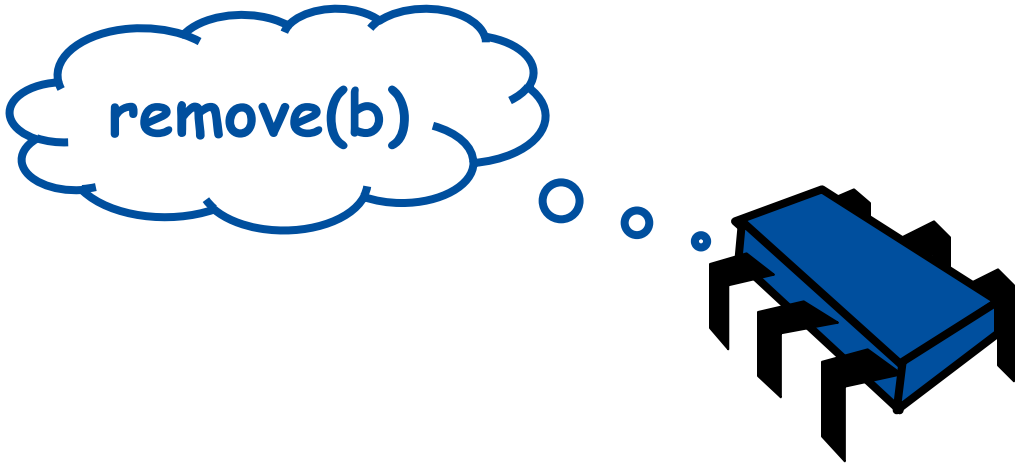


Insight

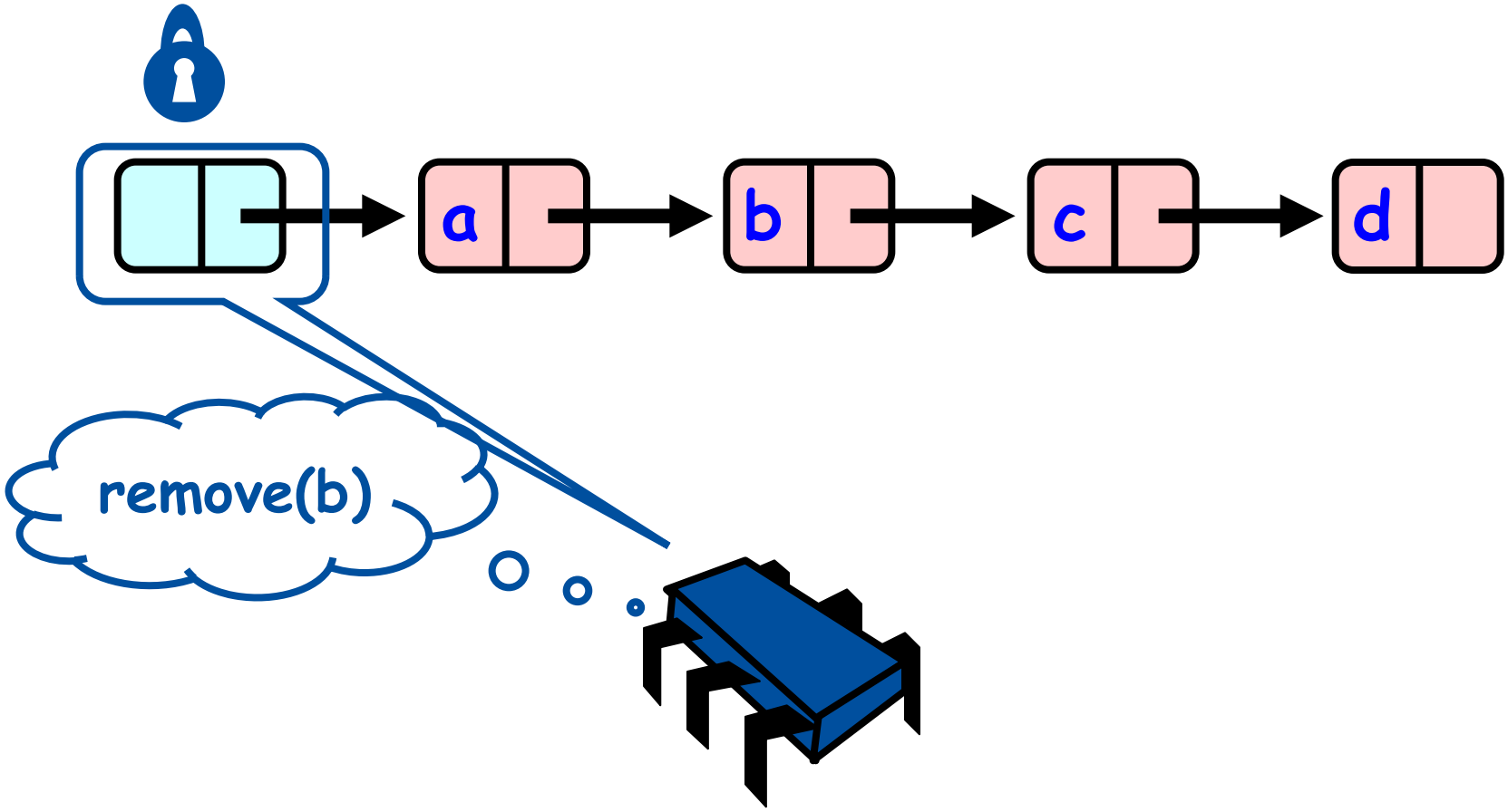
- If a node is locked, no one can remove its *successor*
- To ensure correct removal, we must lock:
  - Node to be deleted
  - And its predecessor

Hand-Over-Hand Again

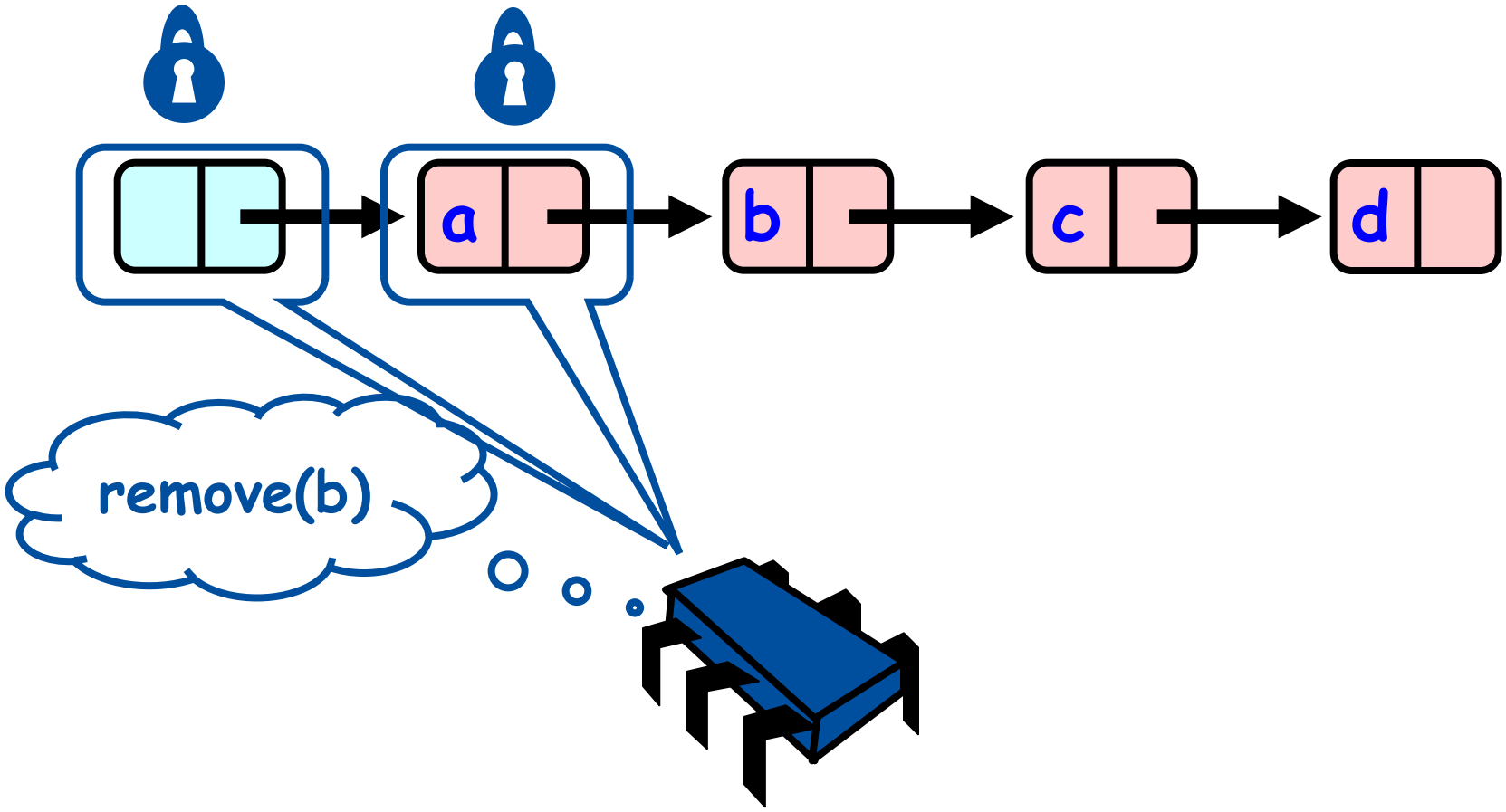




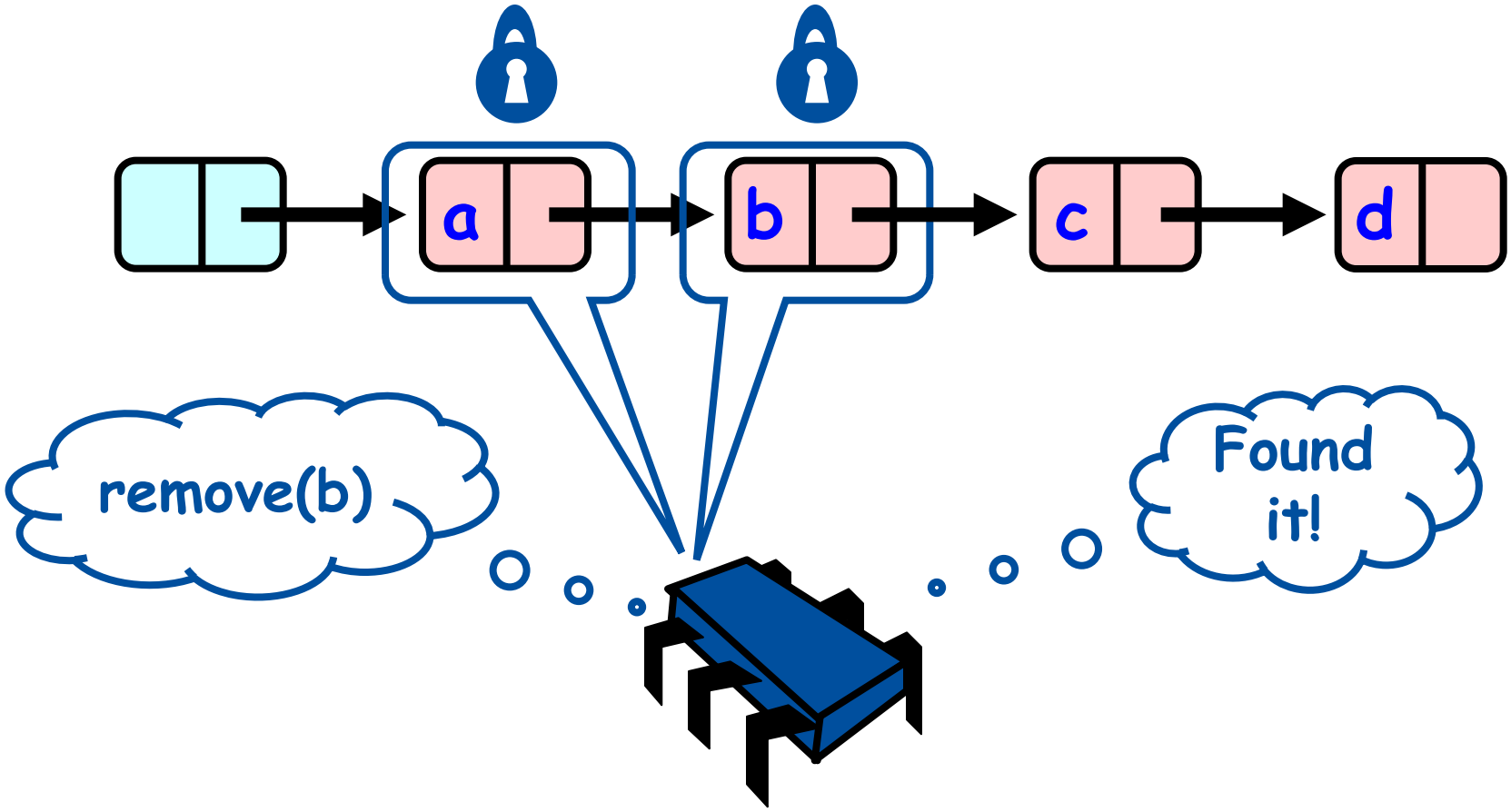
Hand-Over-Hand Again



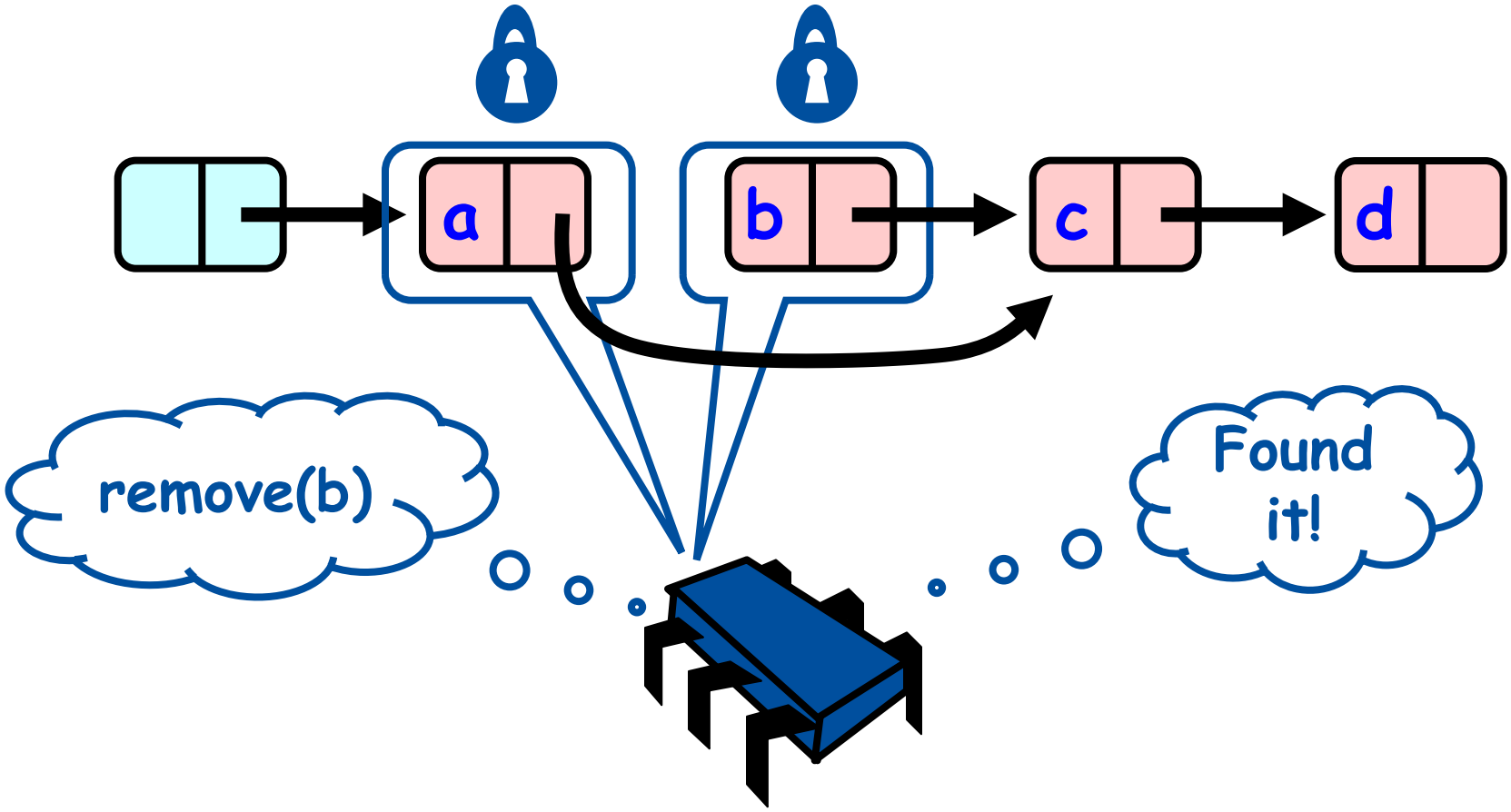
Hand-Over-Hand Again



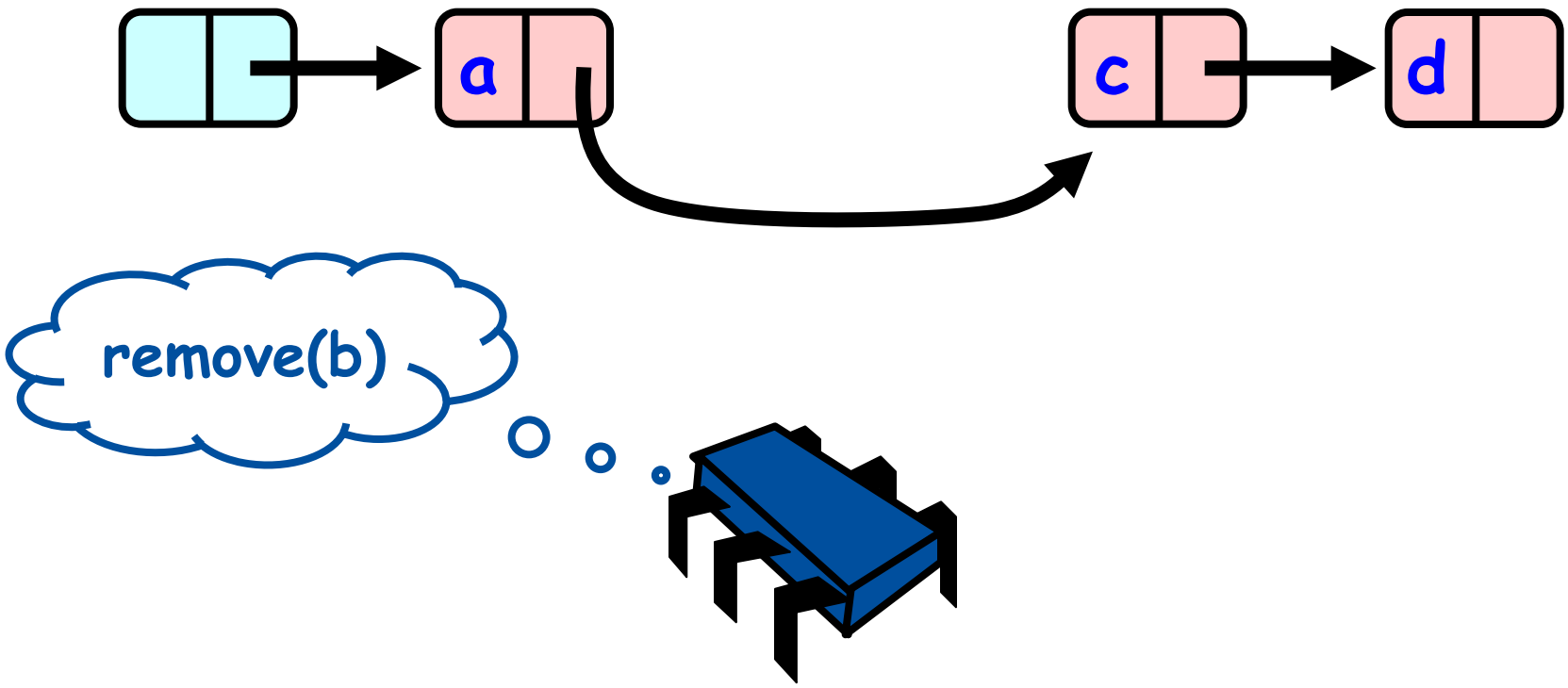
Hand-Over-Hand Again



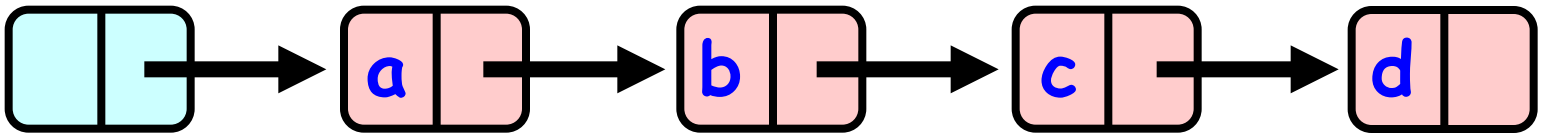
Hand-Over-Hand Again

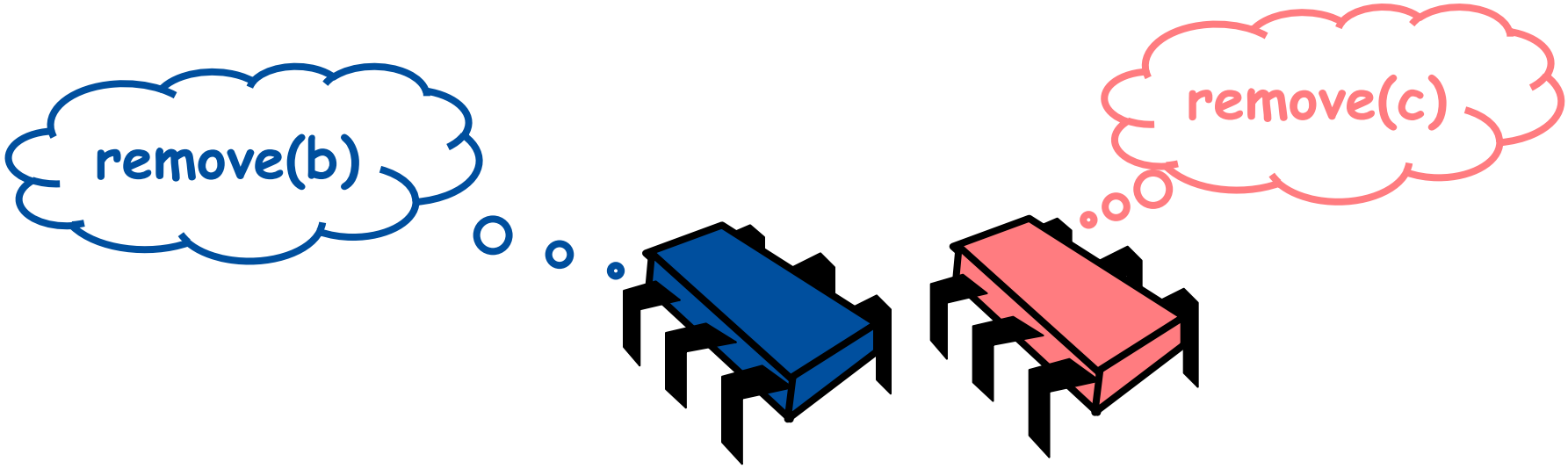


Hand-Over-Hand Again

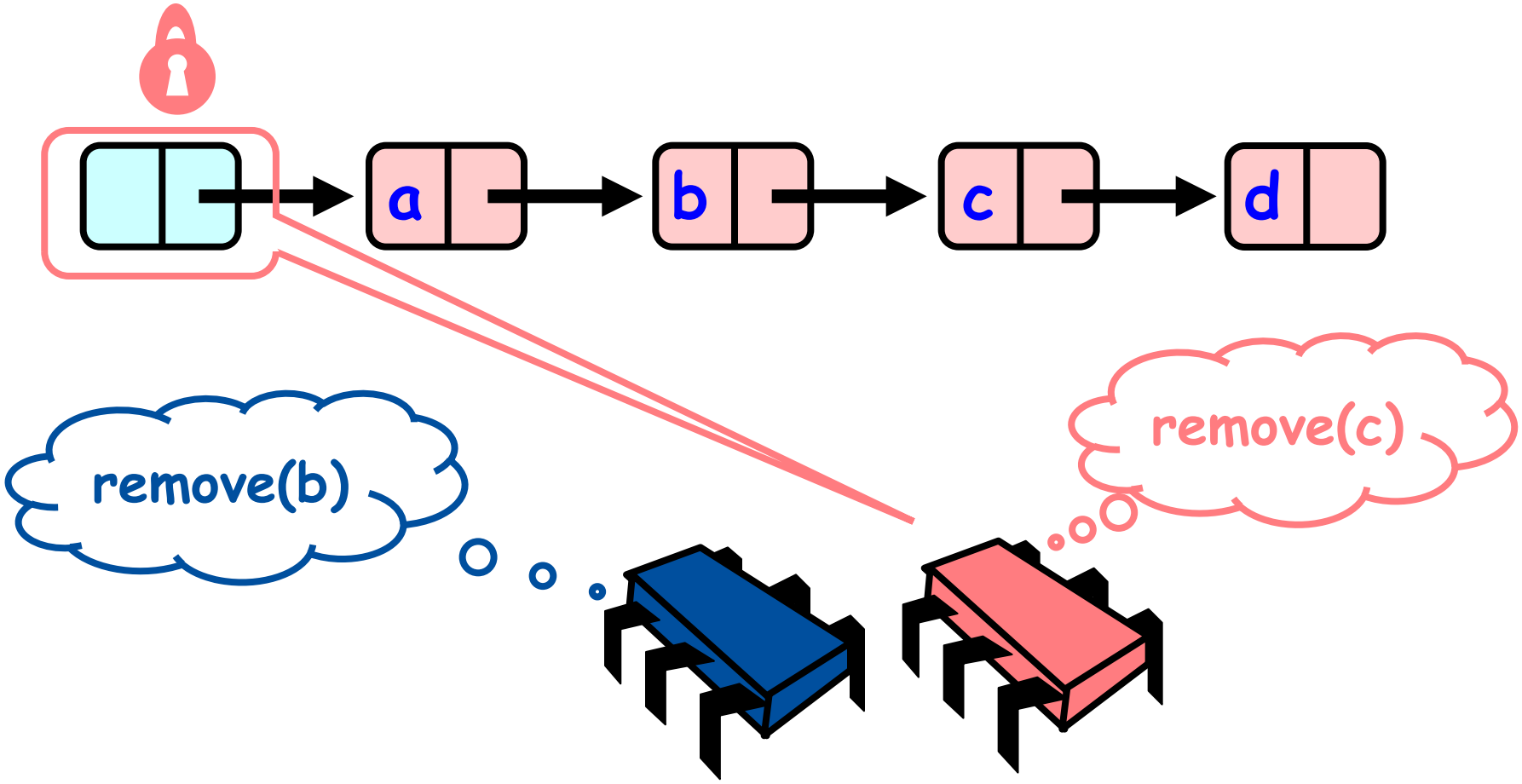


Removing a Node

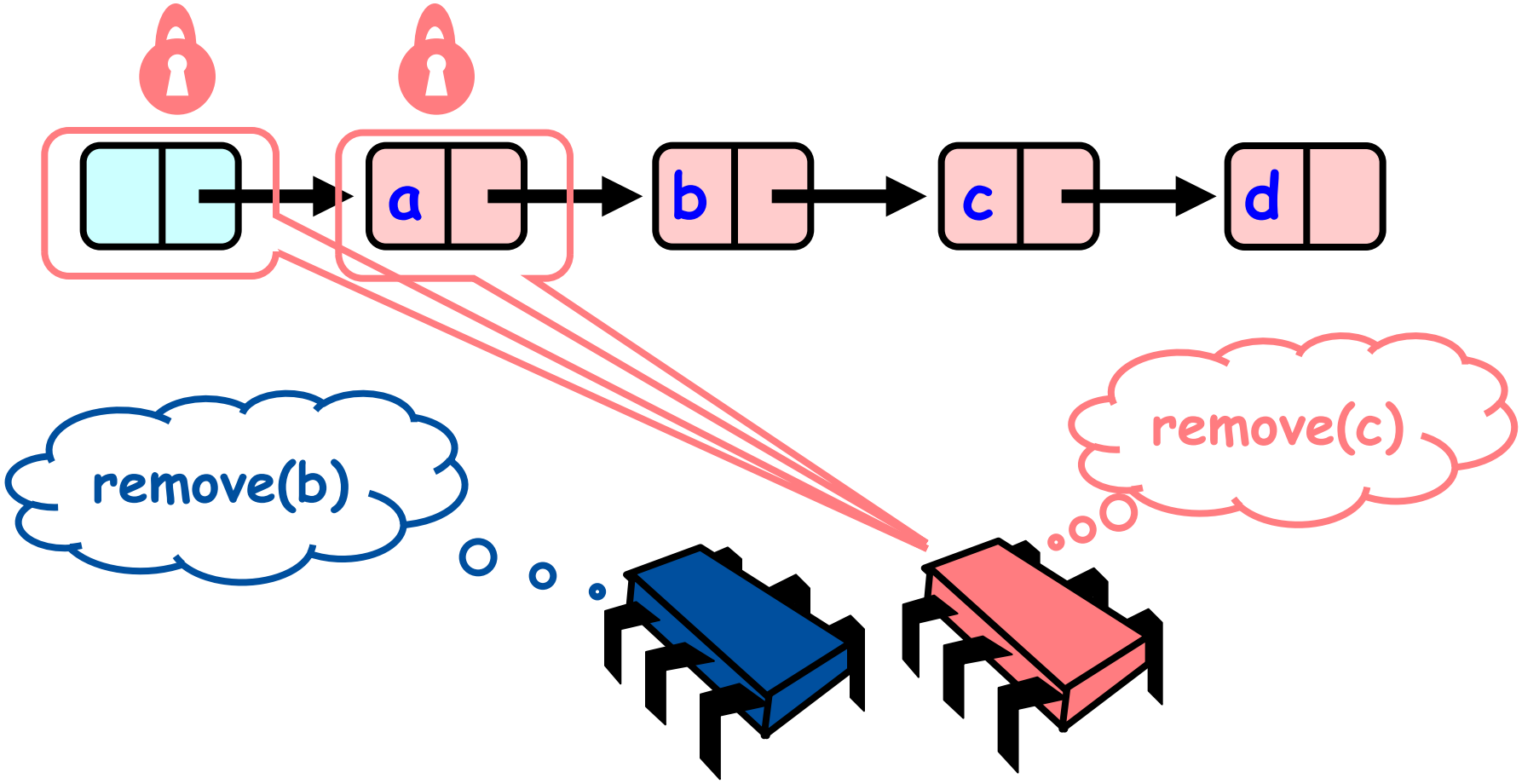




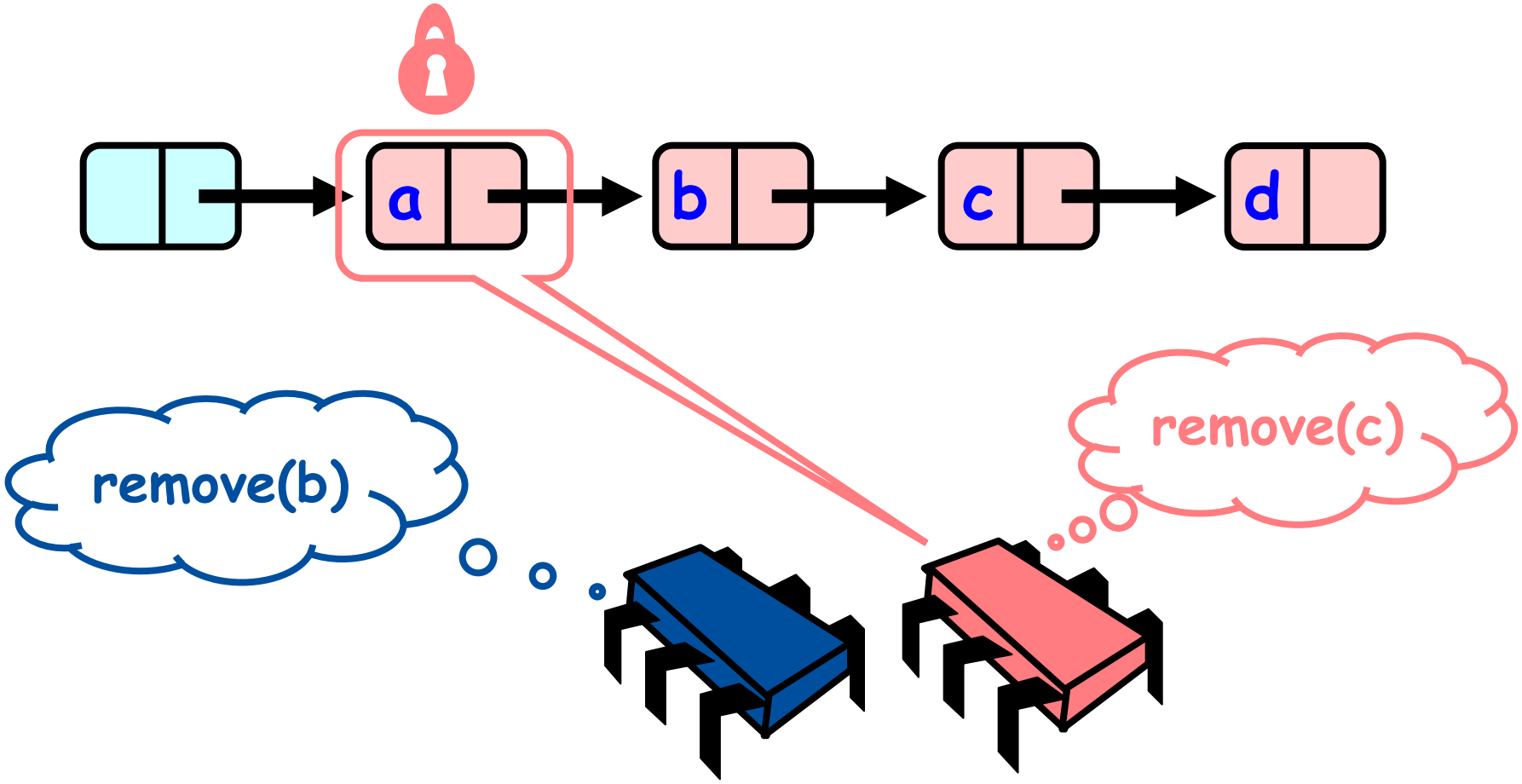
Removing a Node



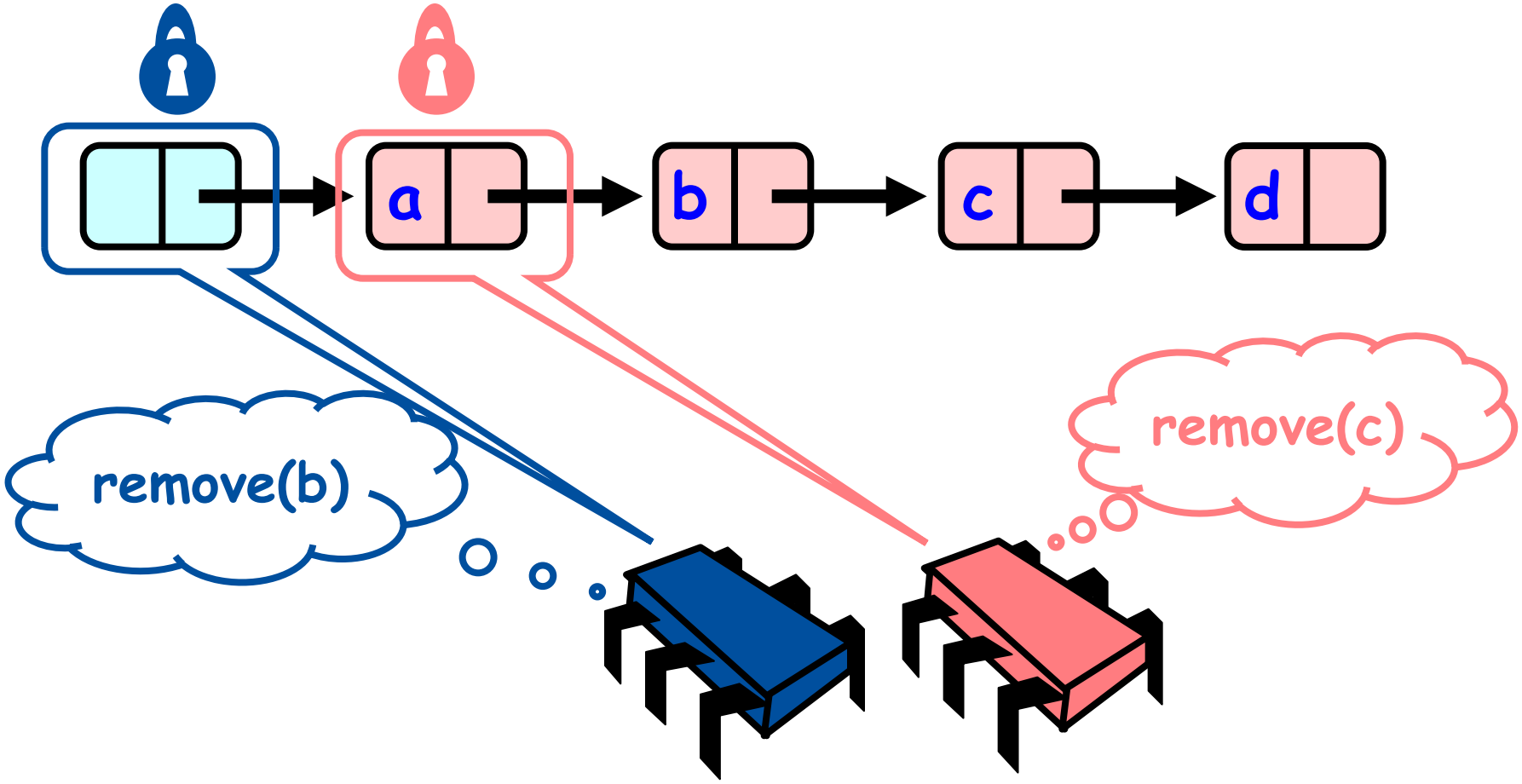
Removing a Node



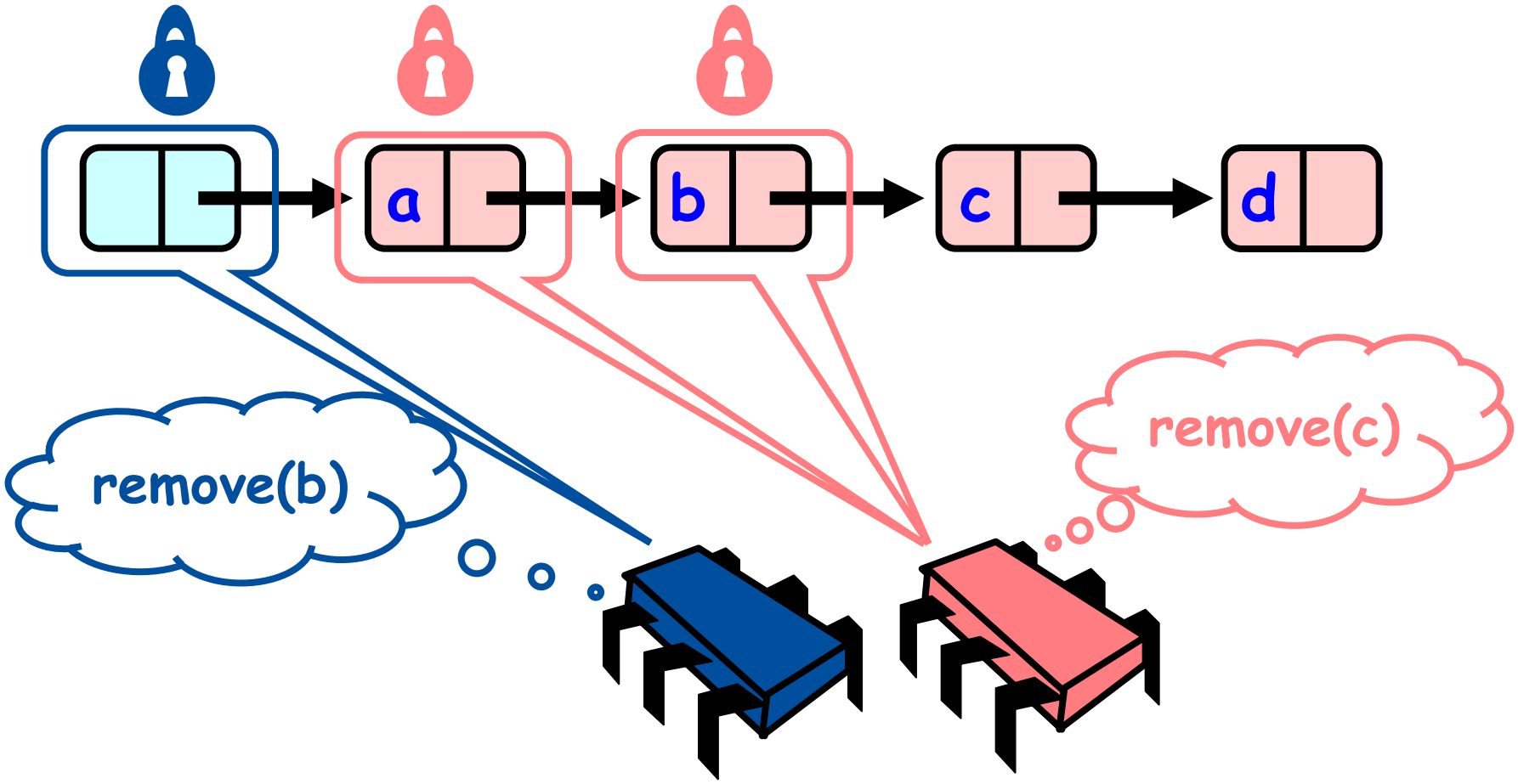
Removing a Node



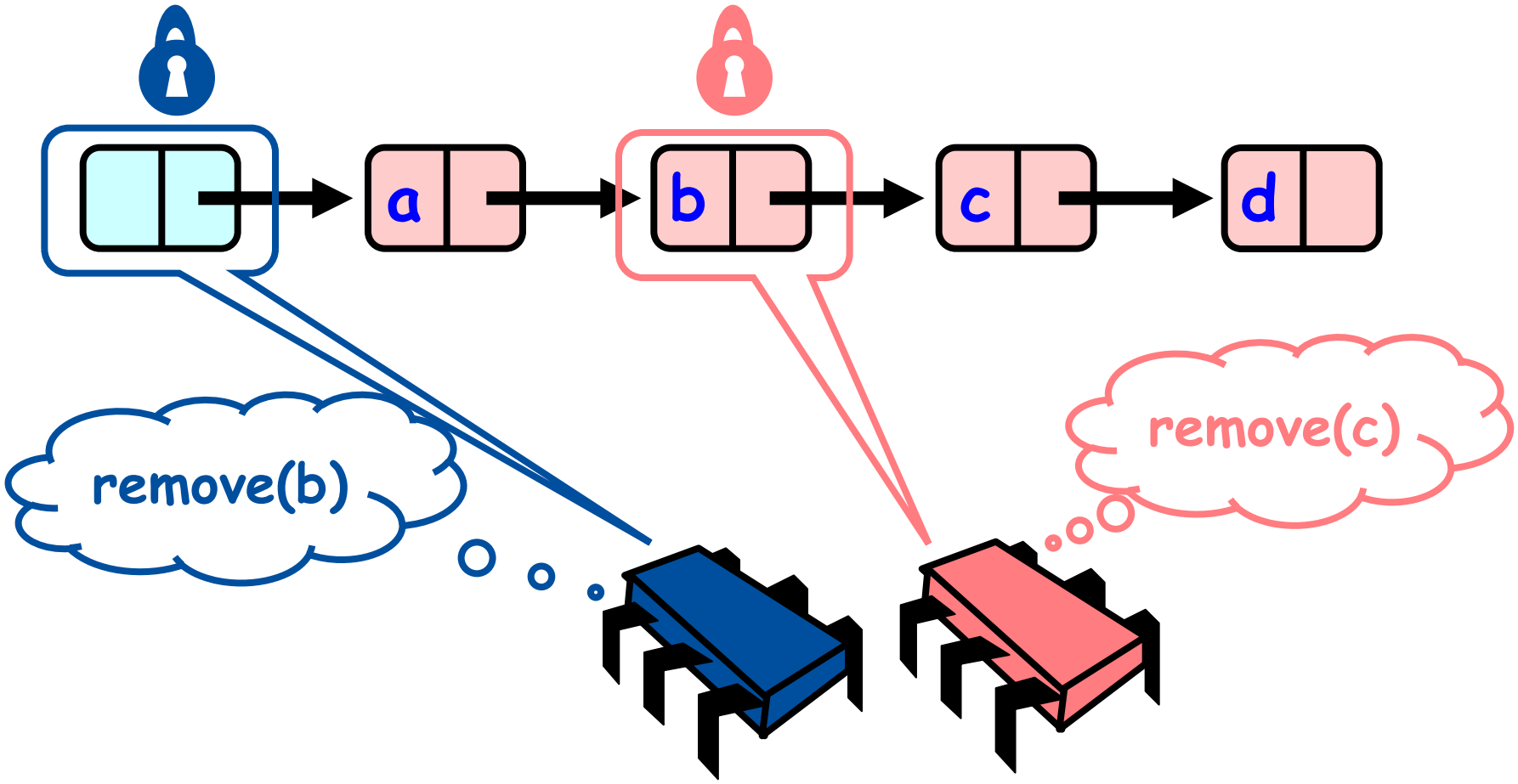
Removing a Node



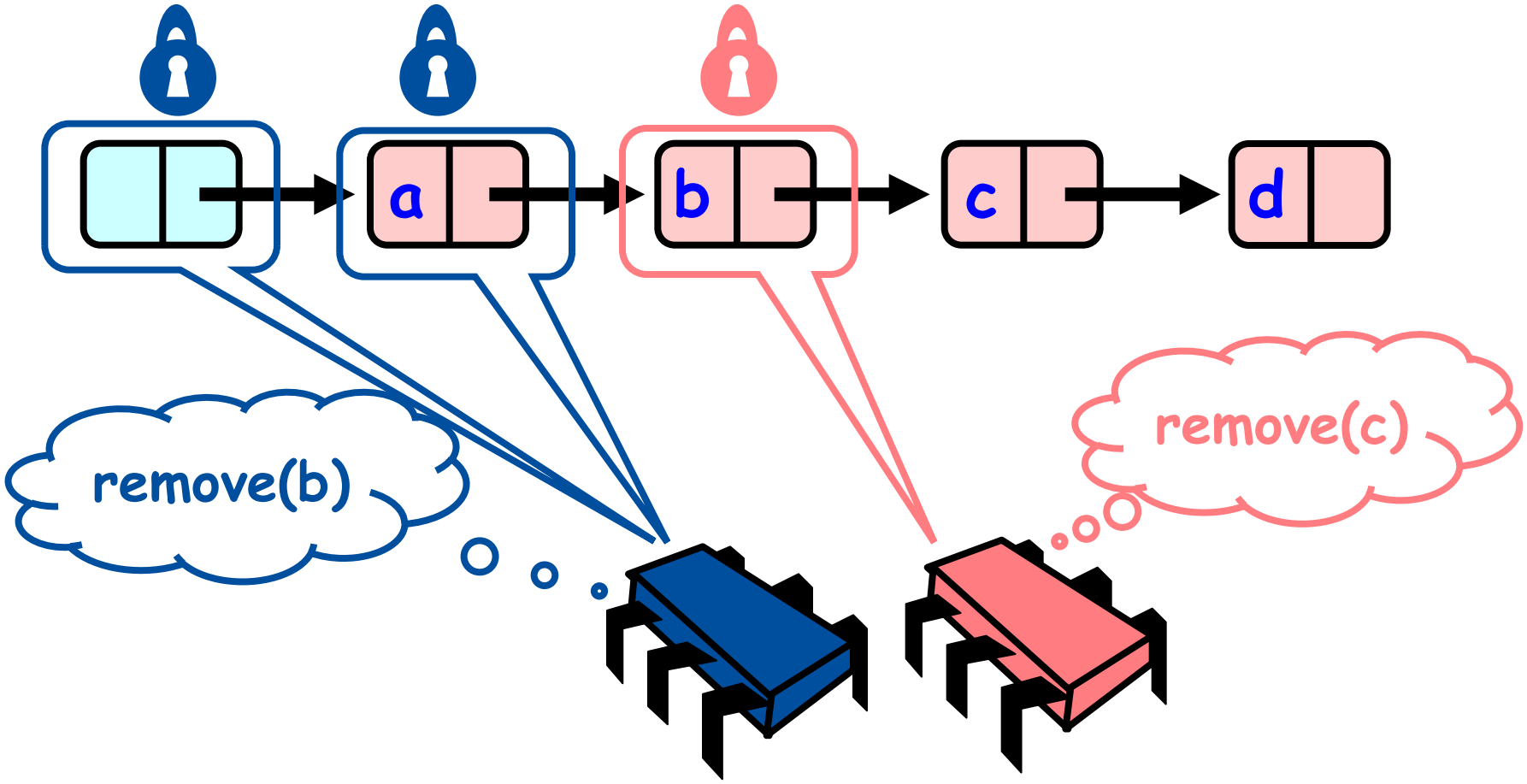
Removing a Node



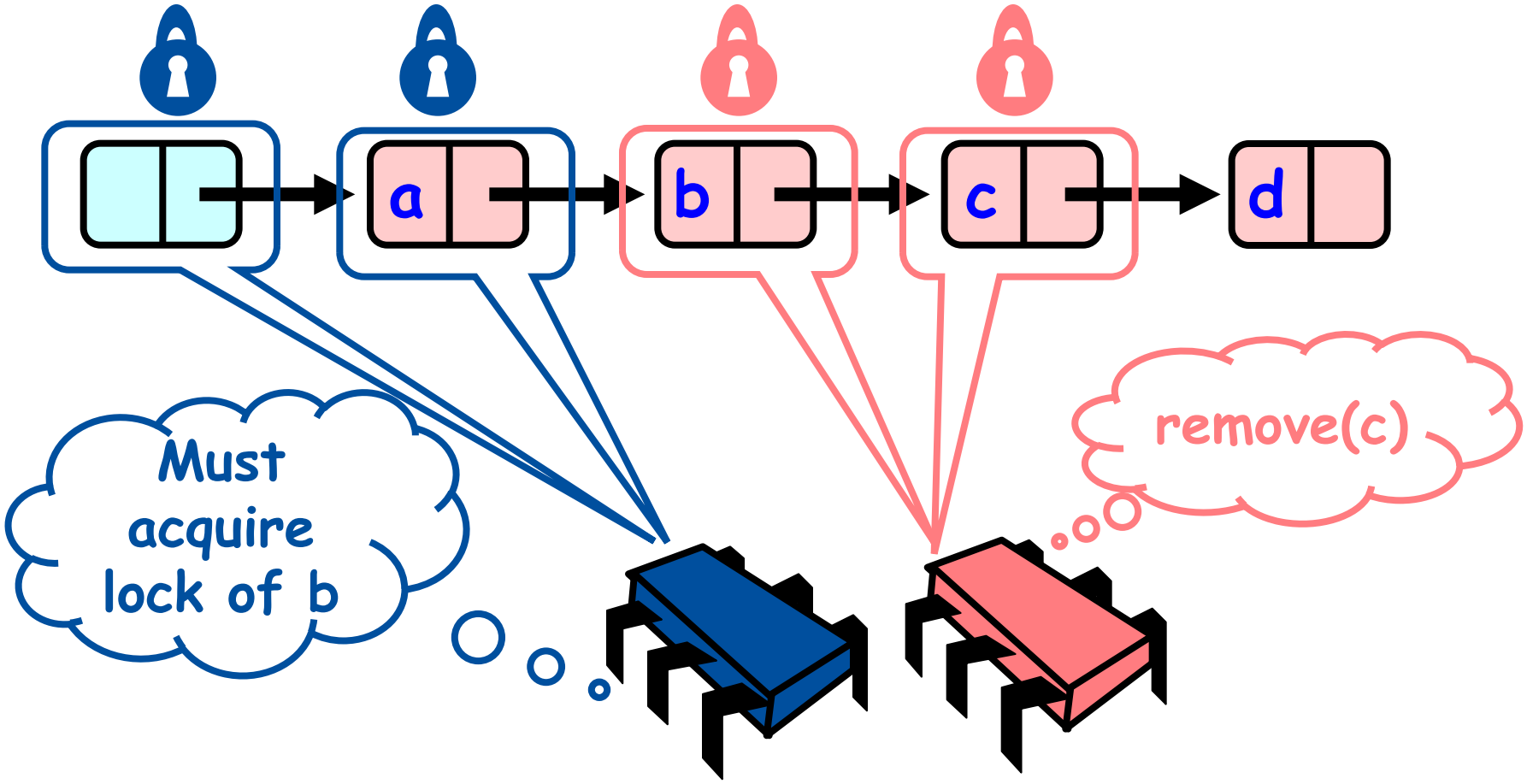
Removing a Node



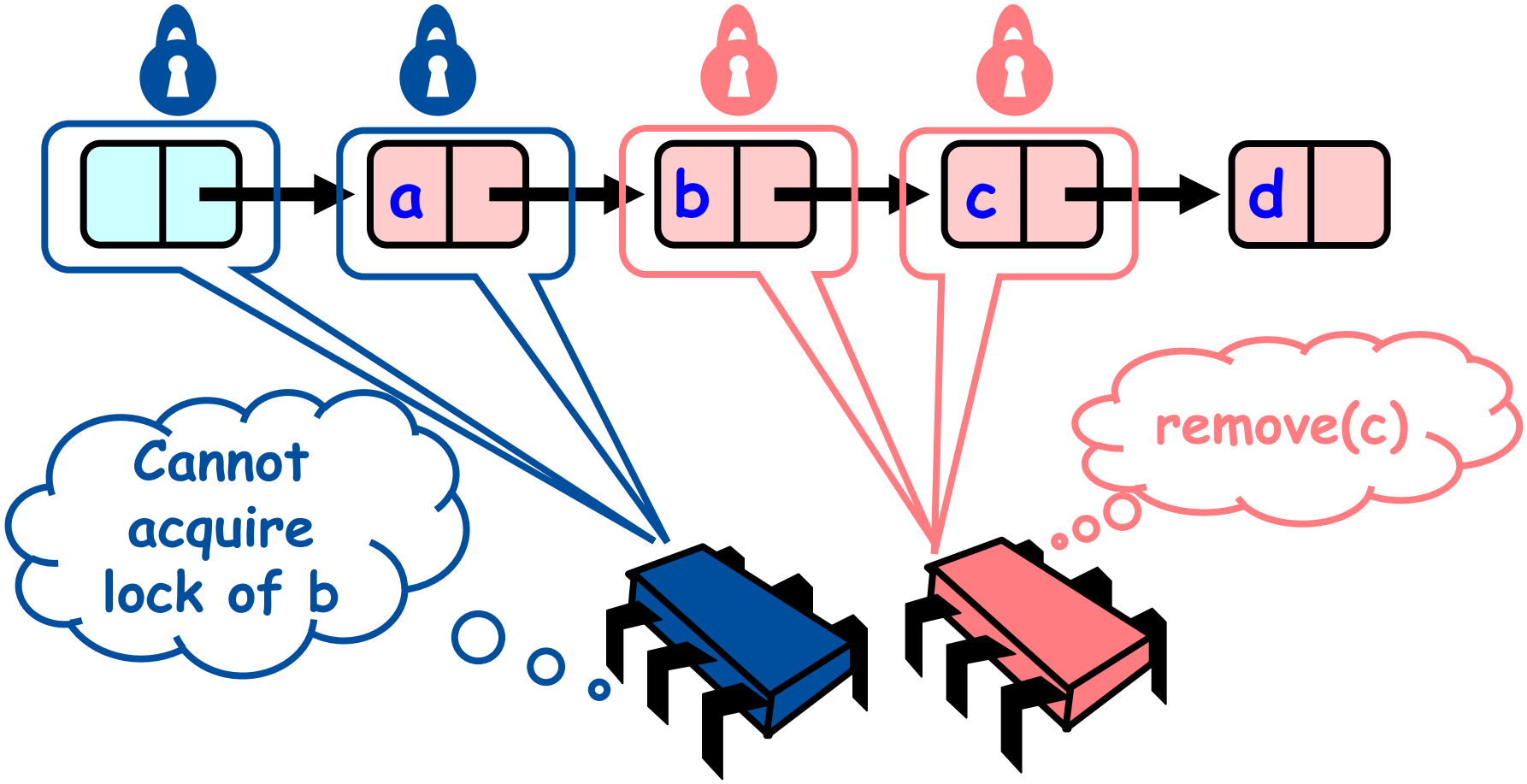
Removing a Node



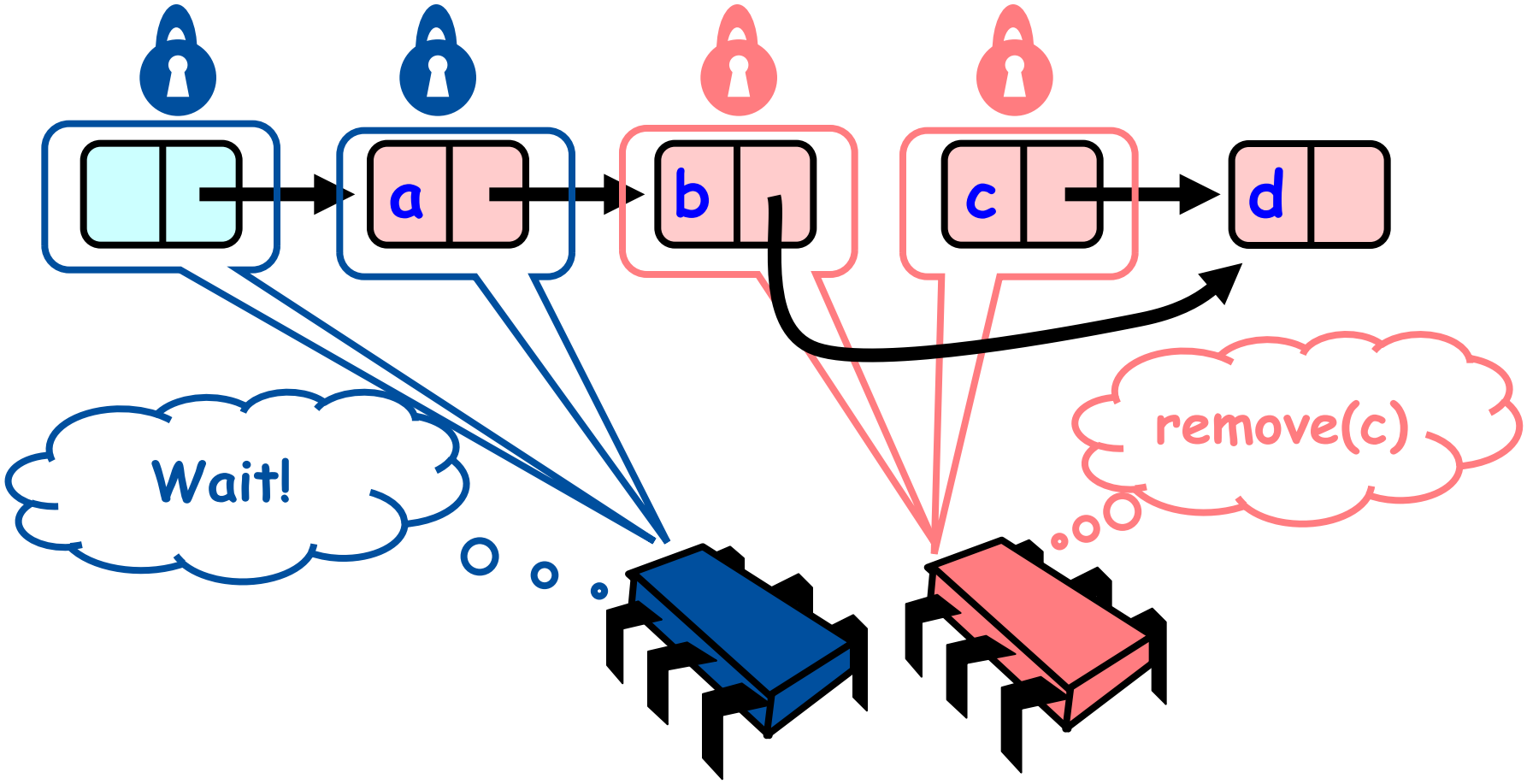
Removing a Node



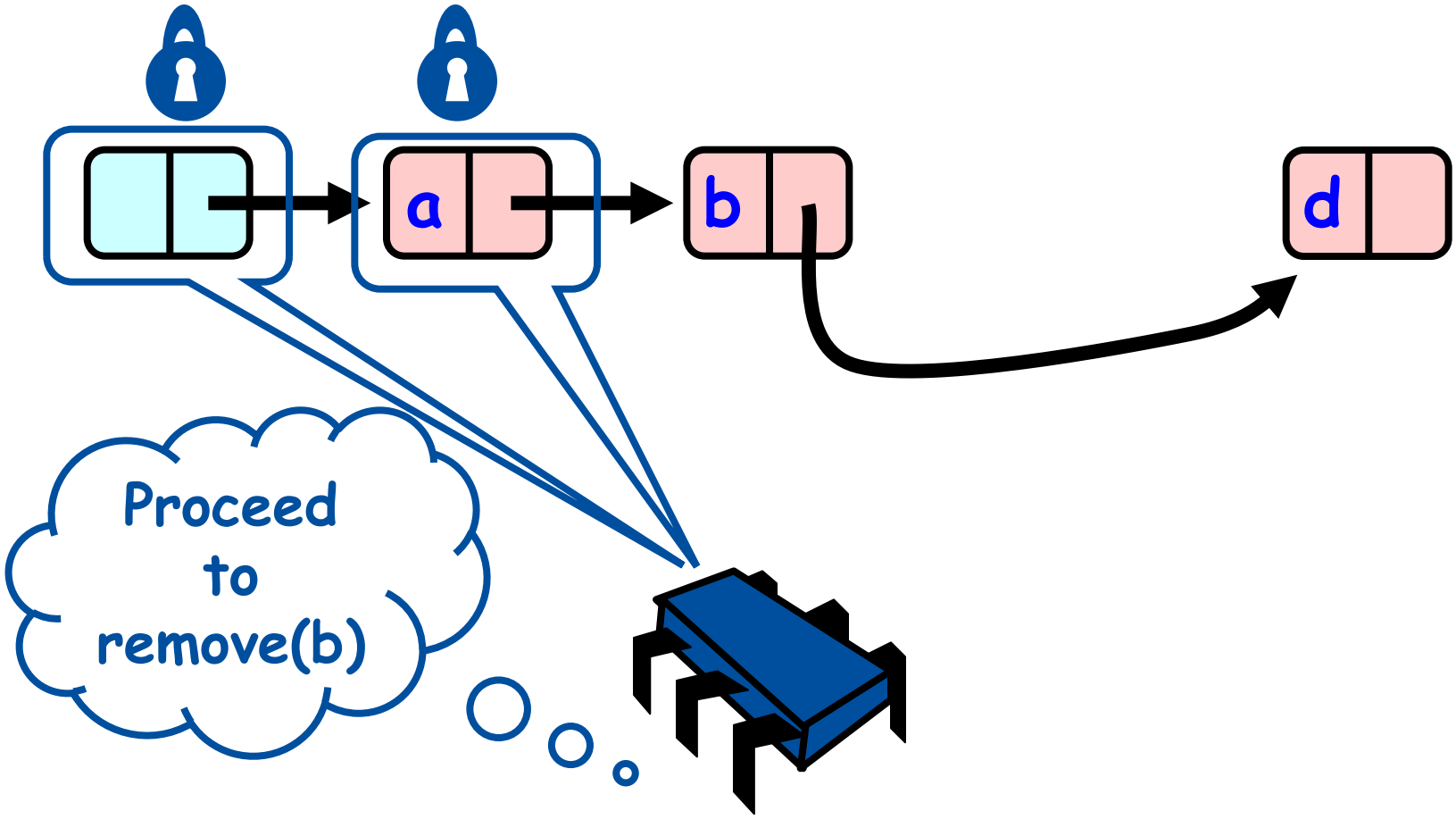
Removing a Node



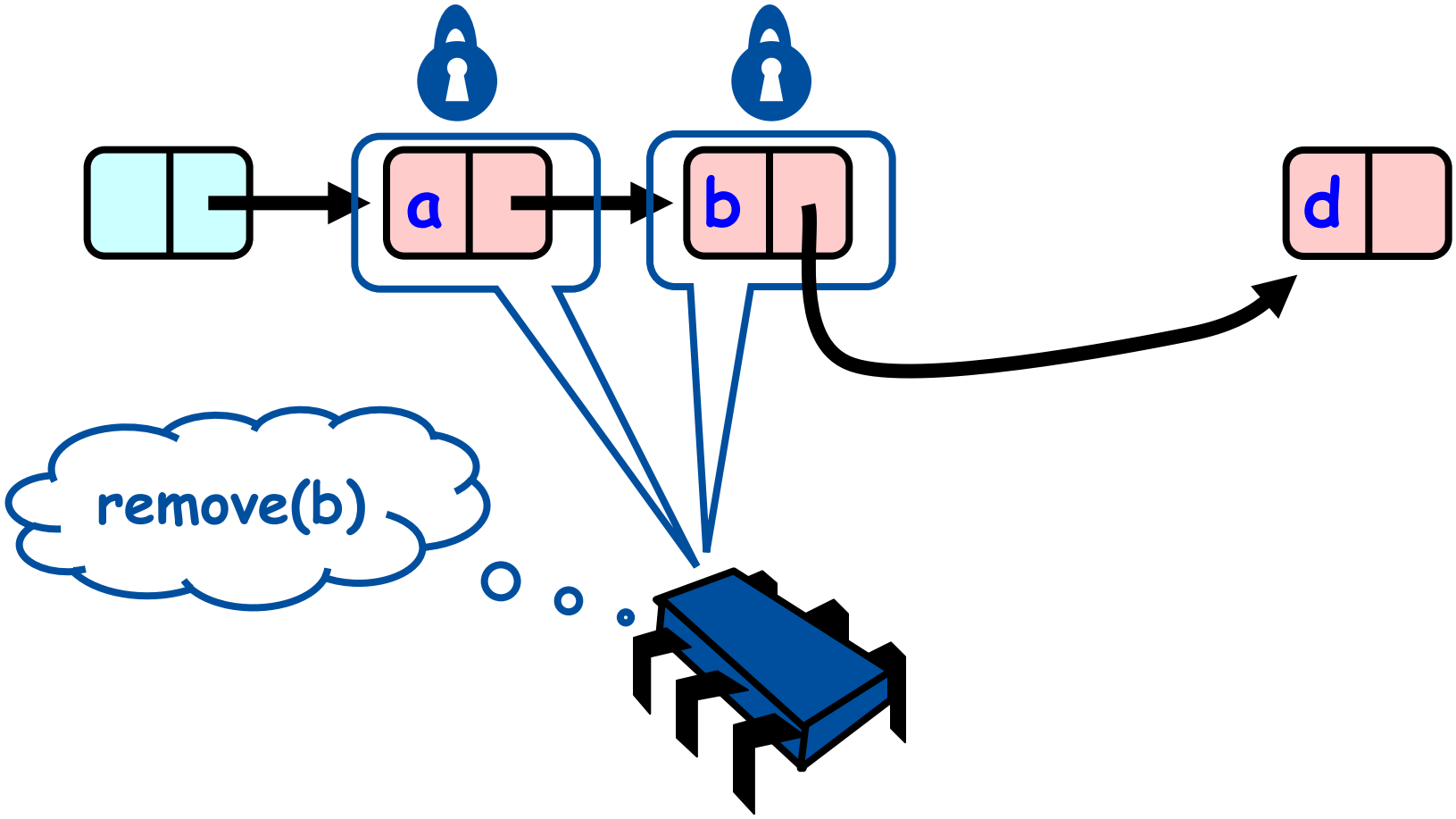
Removing a Node



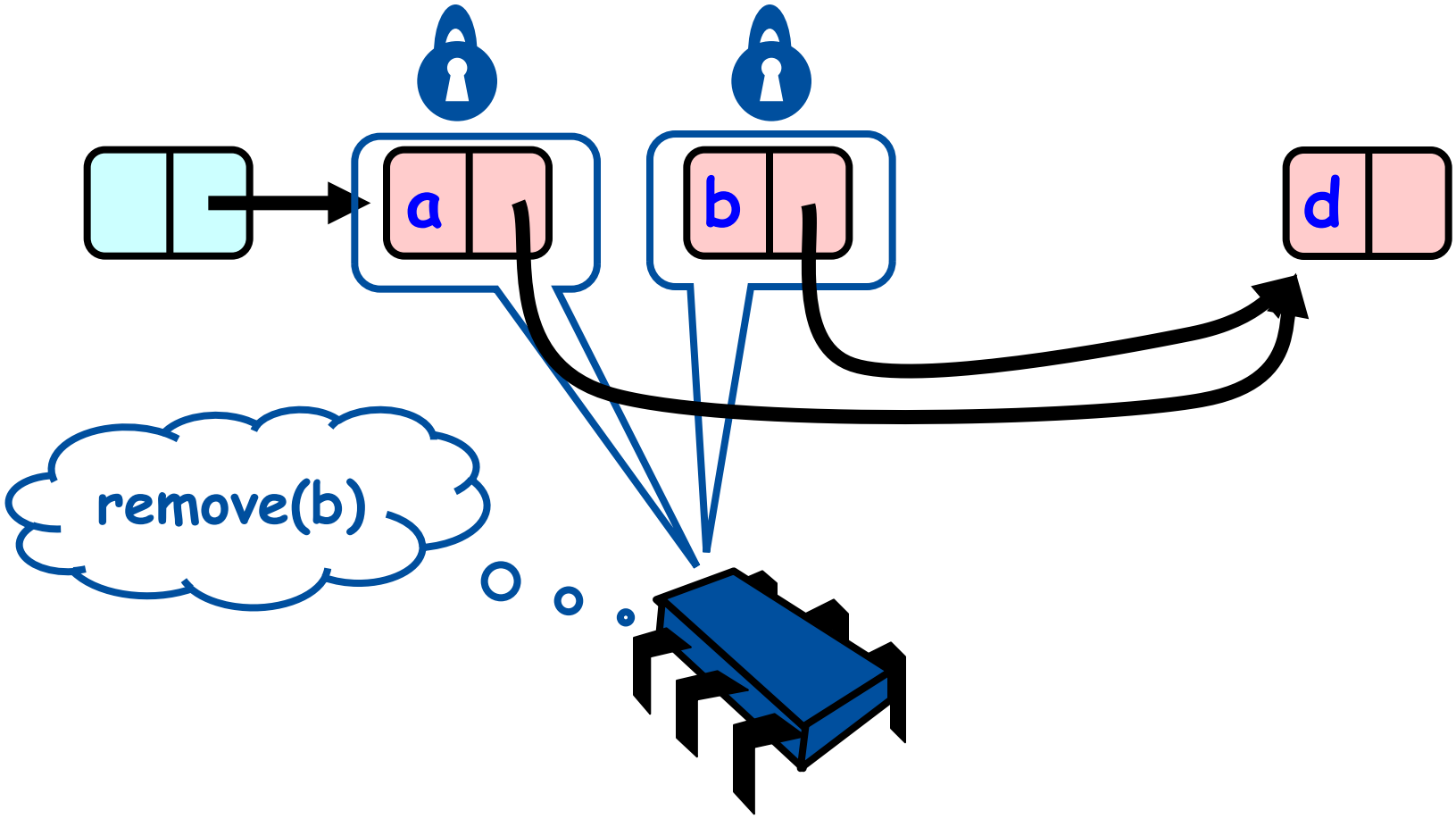
Removing a Node



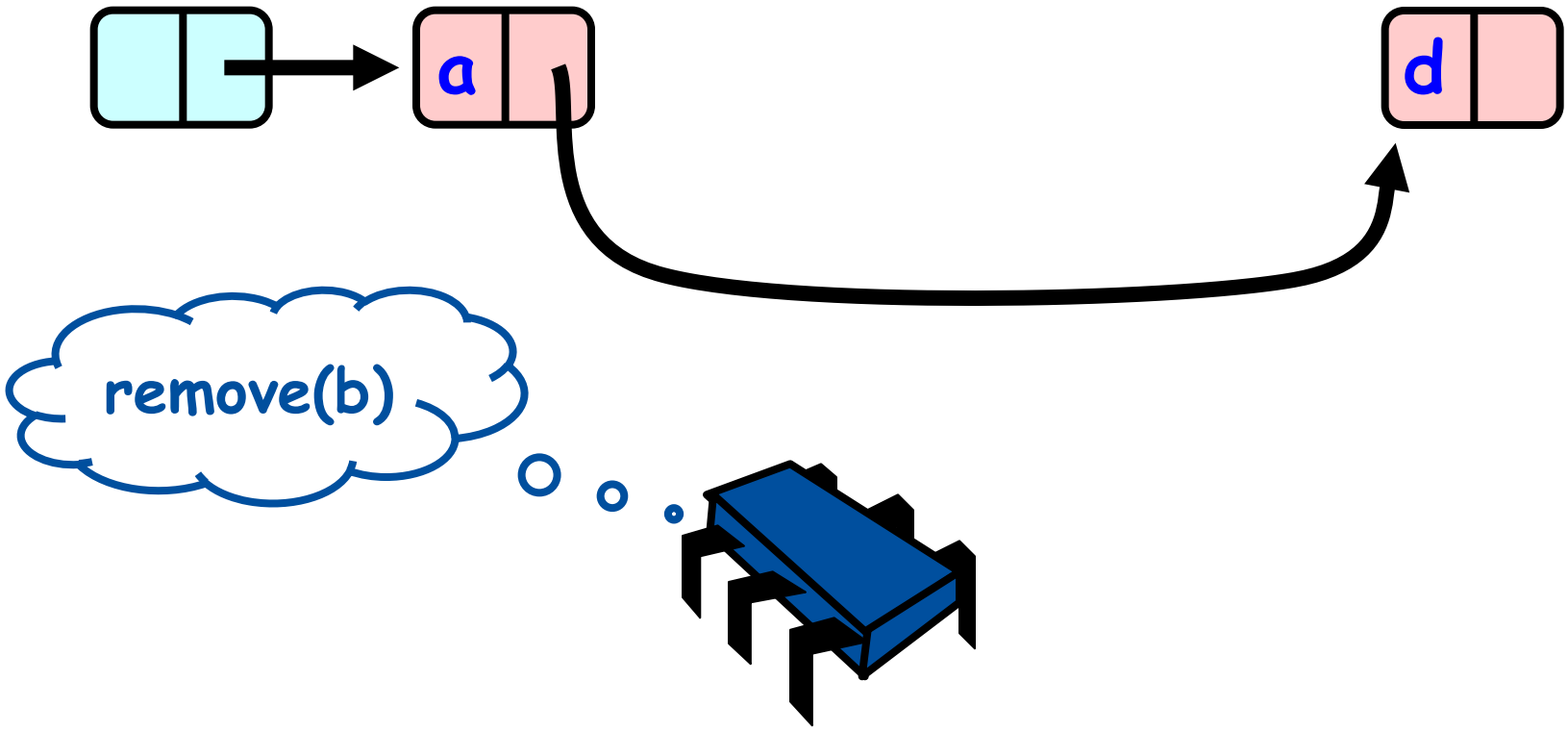
Removing a Node



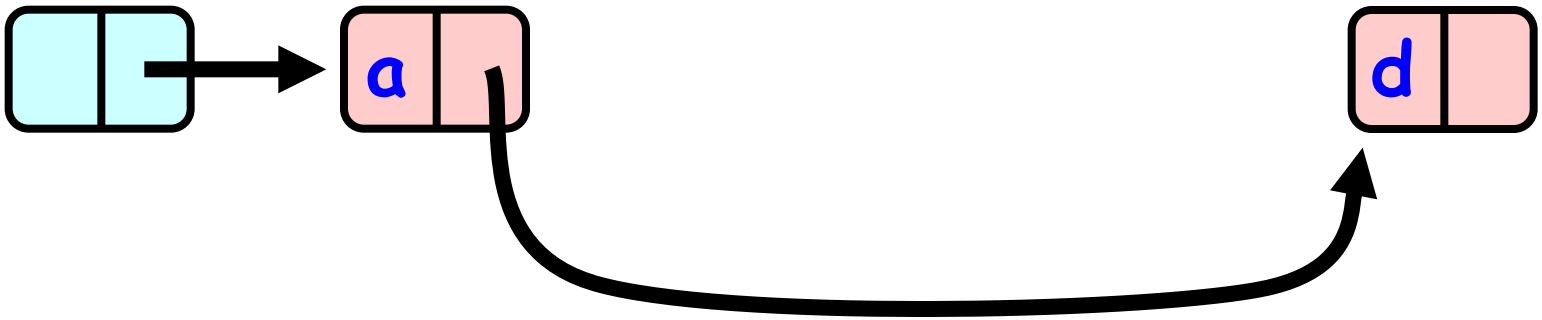
Removing a Node



Removing a Node



Removing a Node



Adding & Removing a Node

- Remove: Lock the node to be deleted and its predecessor

- Add: Lock the predecessor and its successor
  (neither can be deleted)
  (no new node can be added)
- Can successor lock be released?
- What about after search?

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Linearization points?

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    }
26
27  }
28
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

- pred reachable from head
- curr is pred.next
- So curr.item is in the set

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24
25    }
26
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Linearization point if item is present

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Item not present

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23
24
25    }
26
27    }
28  }
```

- pred reachable from head
- curr is pred.next
- pred.key < key
- key < curr.key

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Linearization point

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    }
26
27  }
28
```

- pred reachable from head
- curr is pred.next
- So curr.item is in the set

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Linearization point

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
```

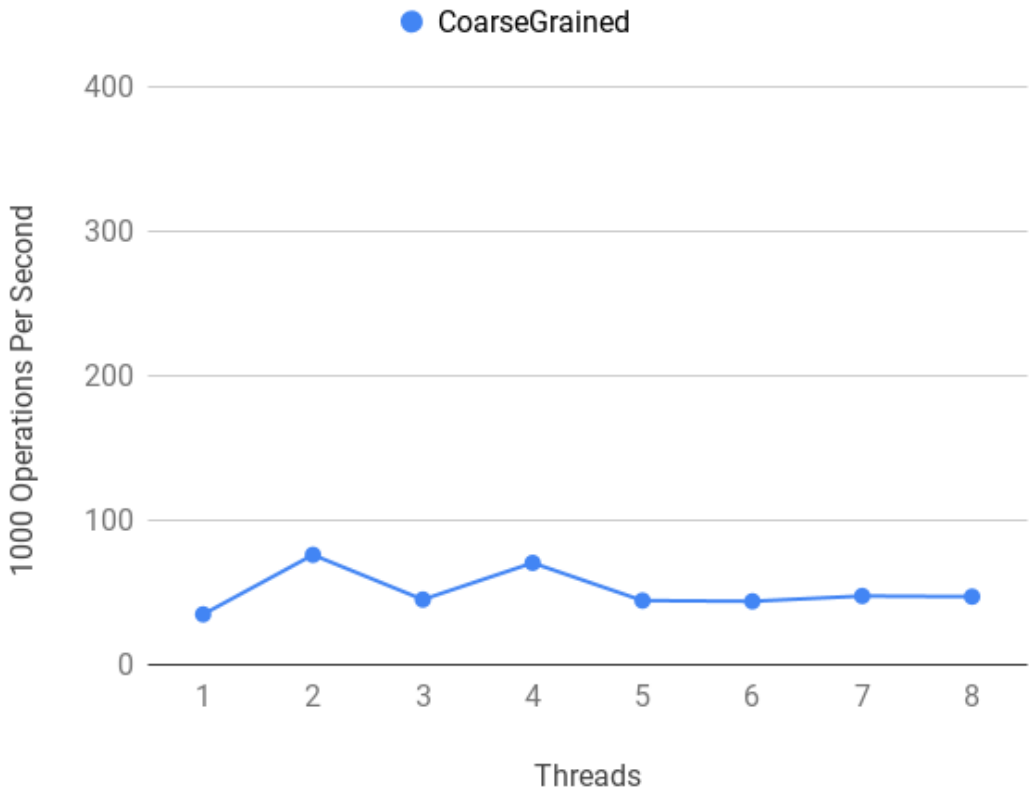
- pred reachable from head
- curr is pred.next
- pred.key < key
- key < curr.key

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

Linearization point

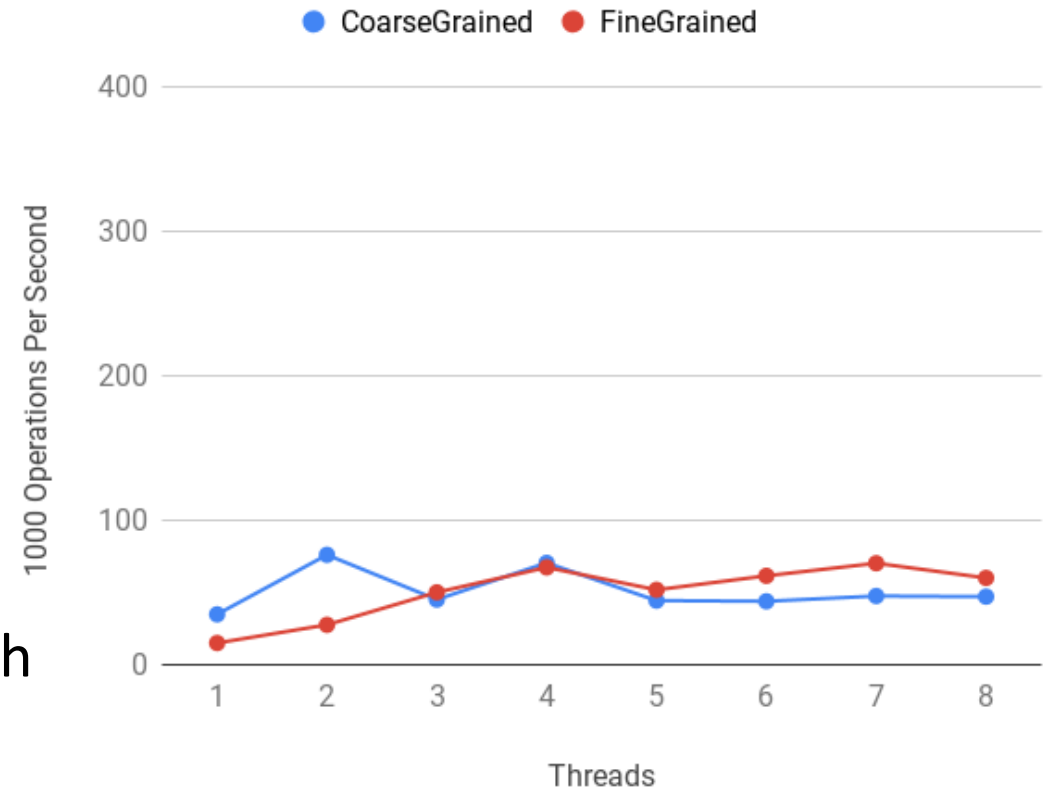
```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Fine-Grained Synchronization



Fine-Grained Synchronization

- Better than coarse-grained
  - Threads can traverse in parallel
- Not ideal – Why?
  - Long chain of acquire/release
  - Threads on different components block each other

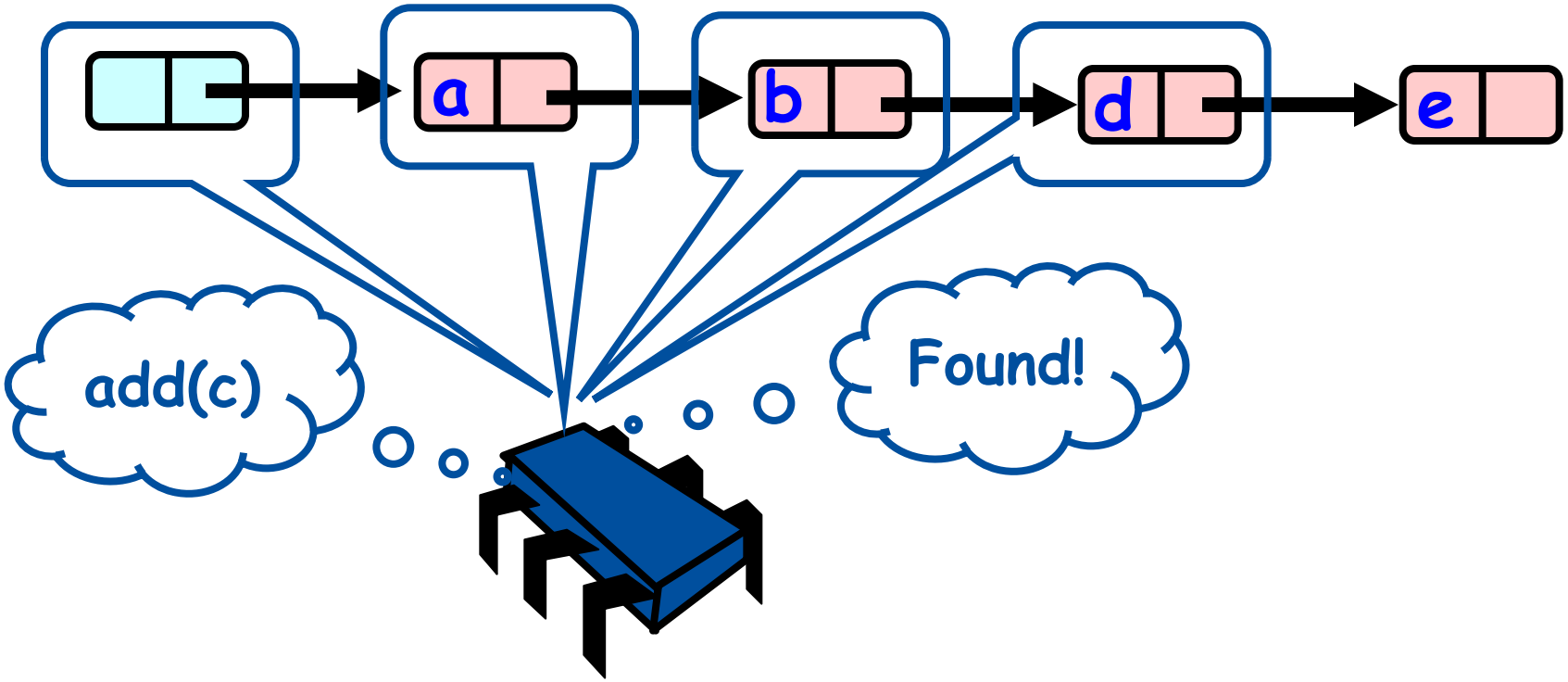


Insight (for Optimistic Strategy)

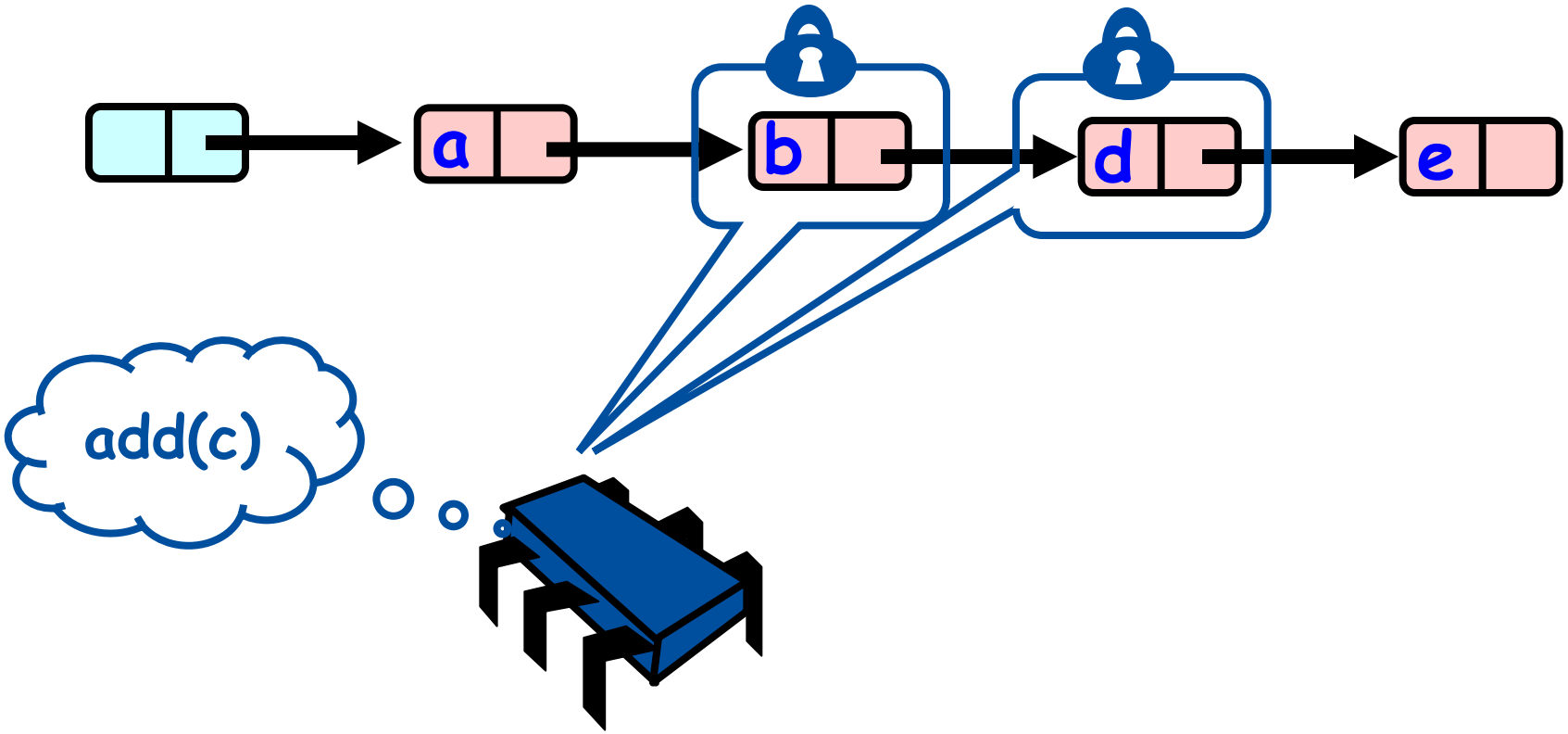
- Two main sub-steps: search and (if found) add/remove
- Majority of time spent in searching
  - The “real” addition and removal

- Optimistic strategy:
  - Perform search without locking
  - If found, lock and perform addition/removal

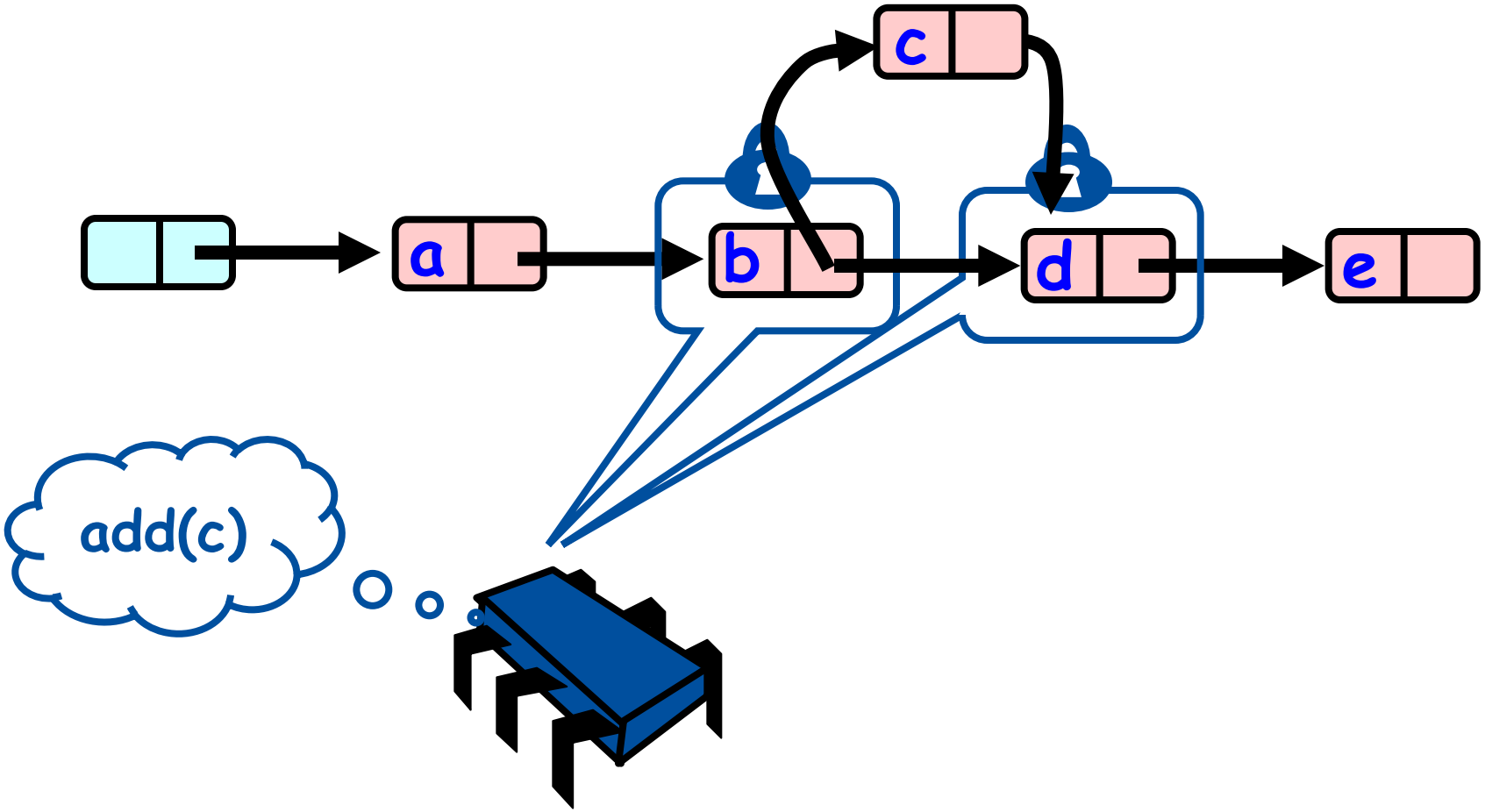
Optimistic Traversing Without Locks



Optimistic Traversing Without Locks

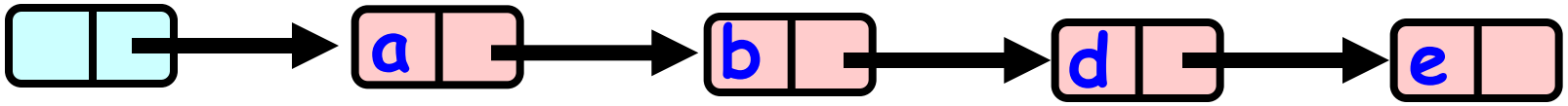


Optimistic Traversing Without Locks



Optimistic Traversing Without Locks

What can go wrong?

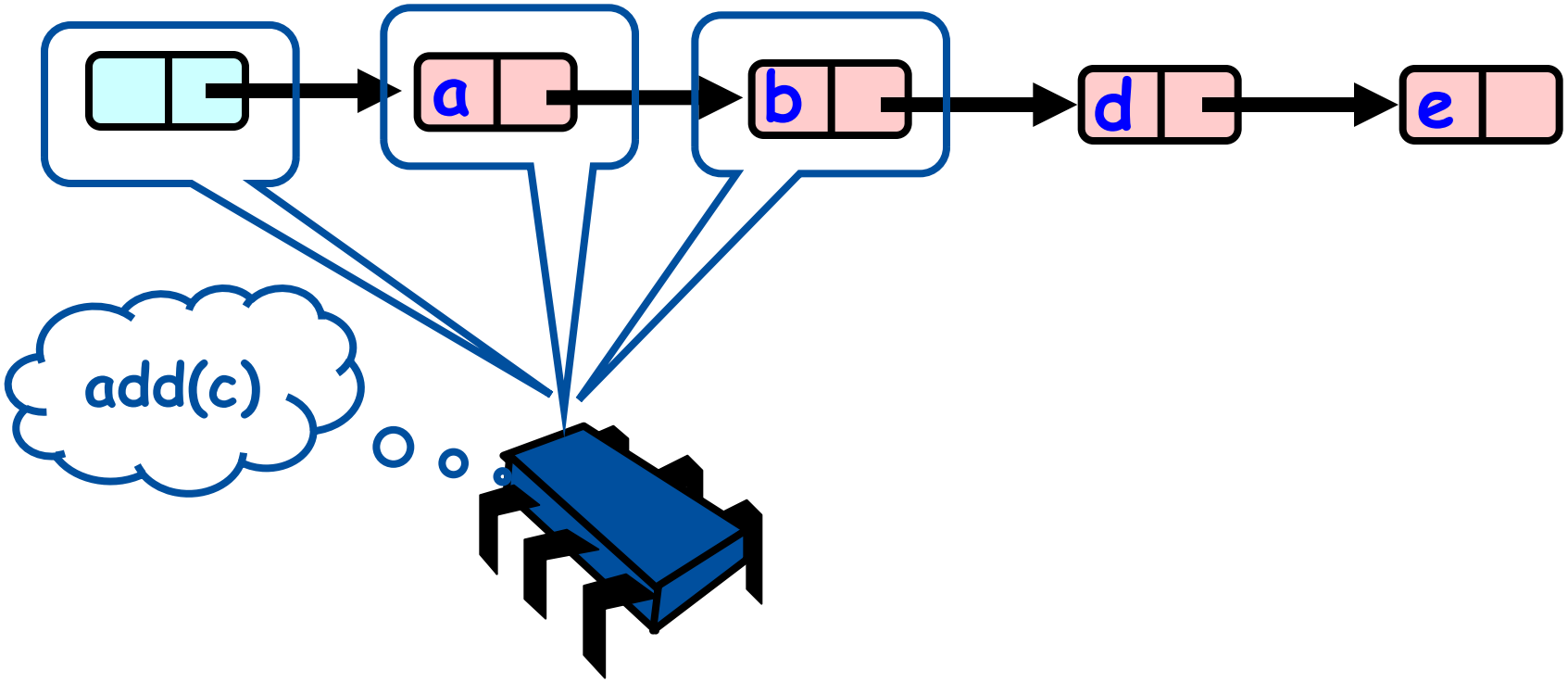


Optimistic Strategy

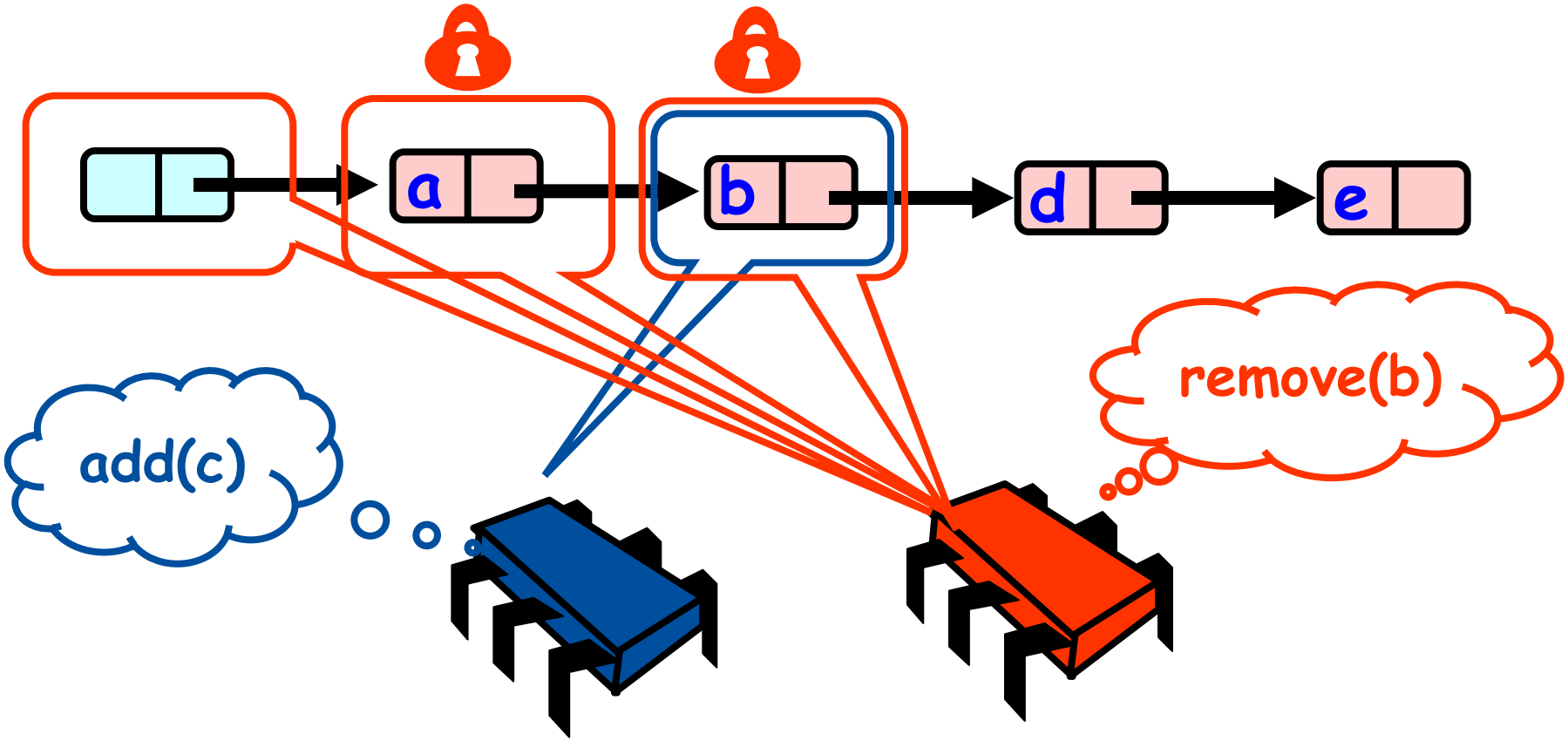
Perform search without locking

If found, lock and perform addition/removal

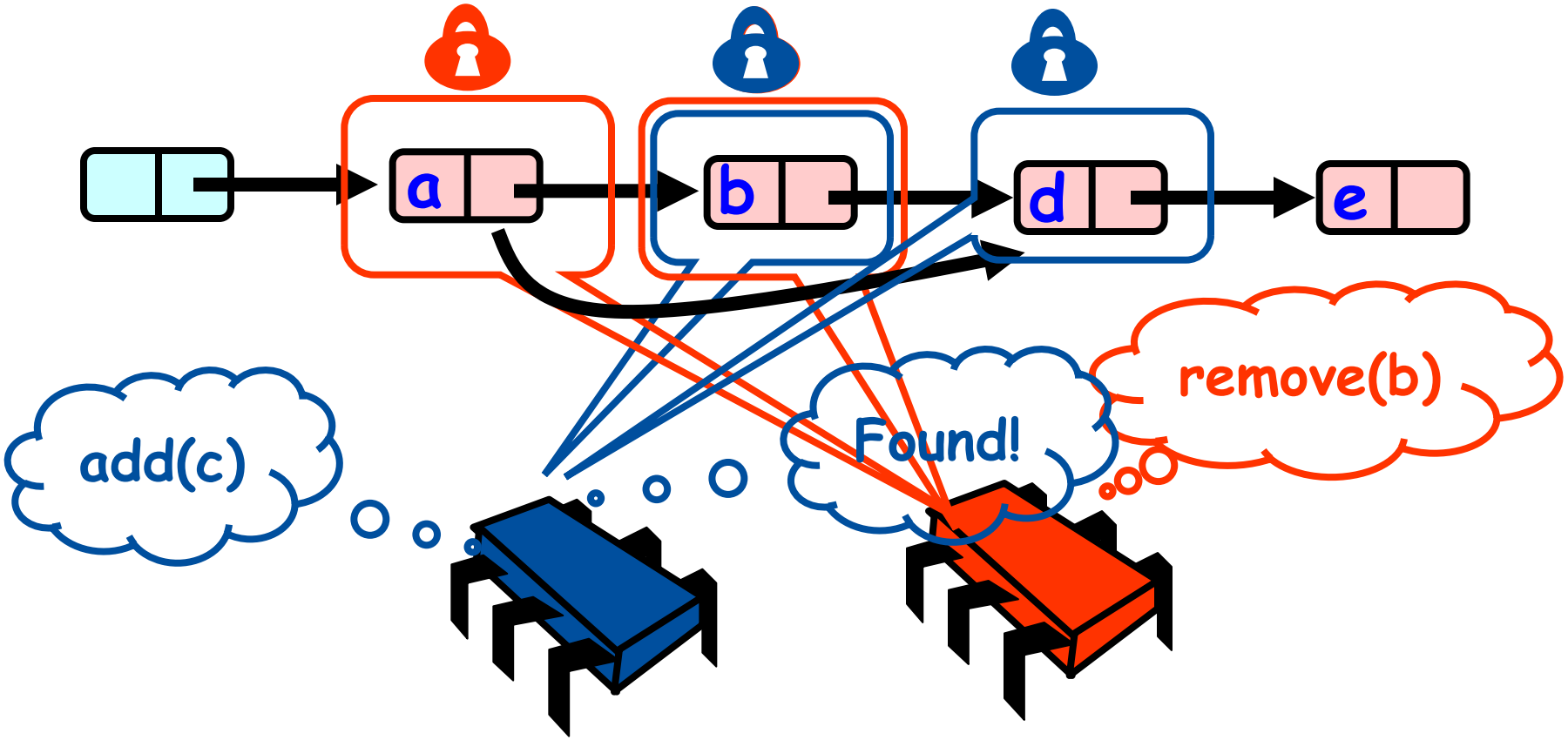
Optimistic: What can go wrong?



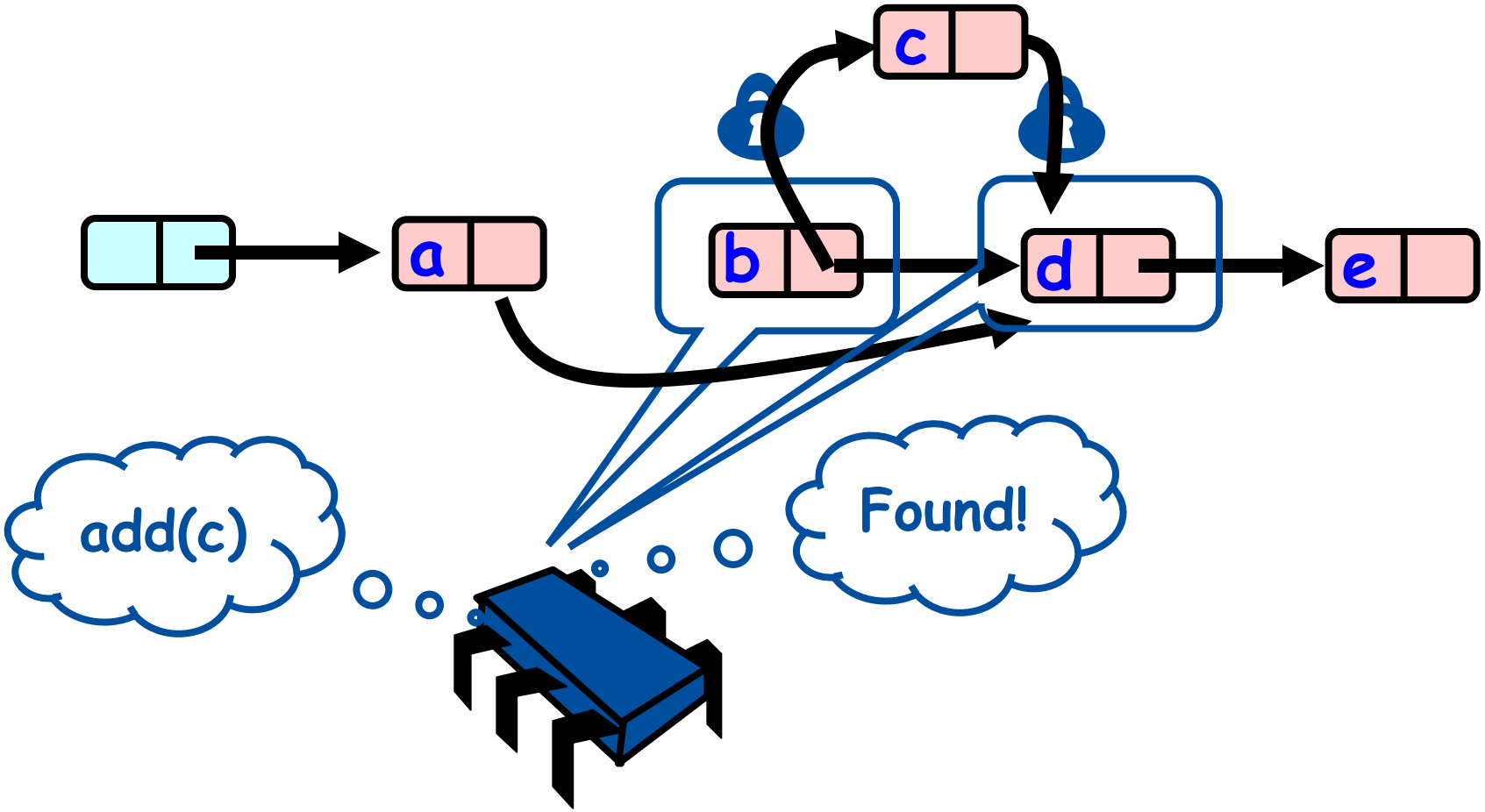
Optimistic: What can go wrong?



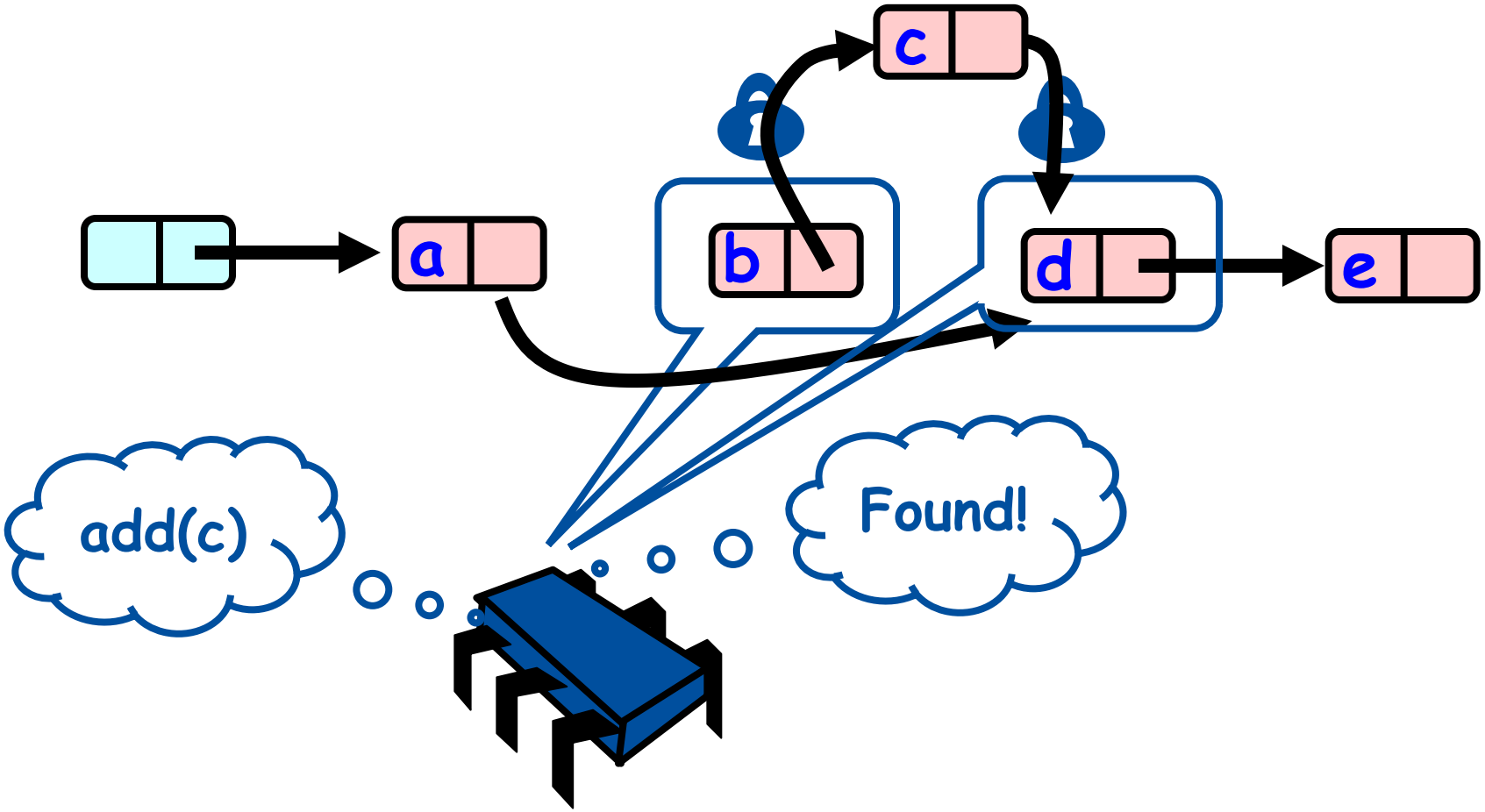
Optimistic: What can go wrong?



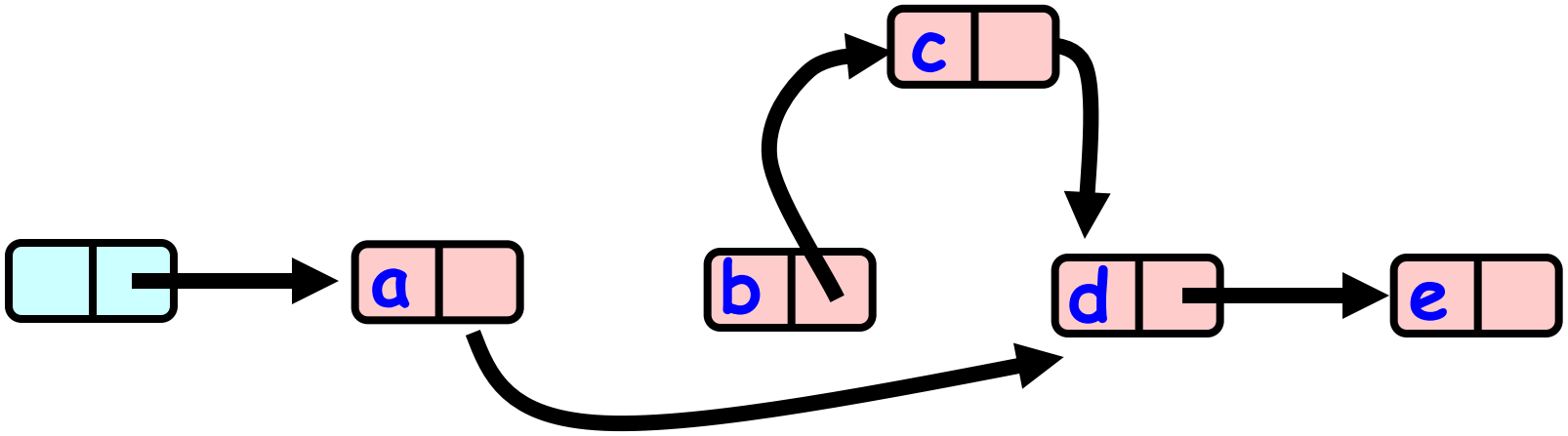
Optimistic: What can go wrong?



Optimistic: What can go wrong?

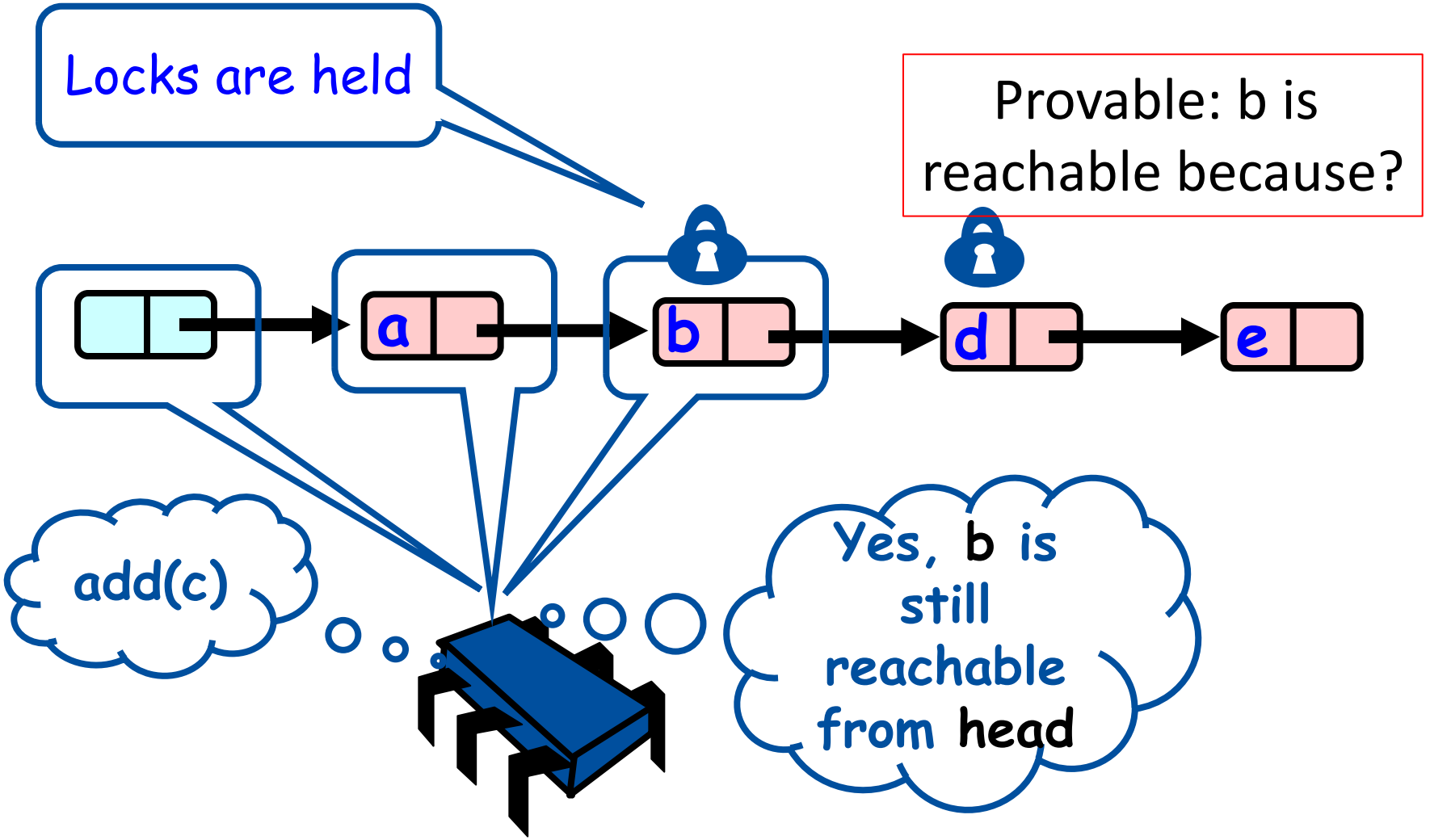


Optimistic: What can go wrong?



Need to validate whether b is reachable

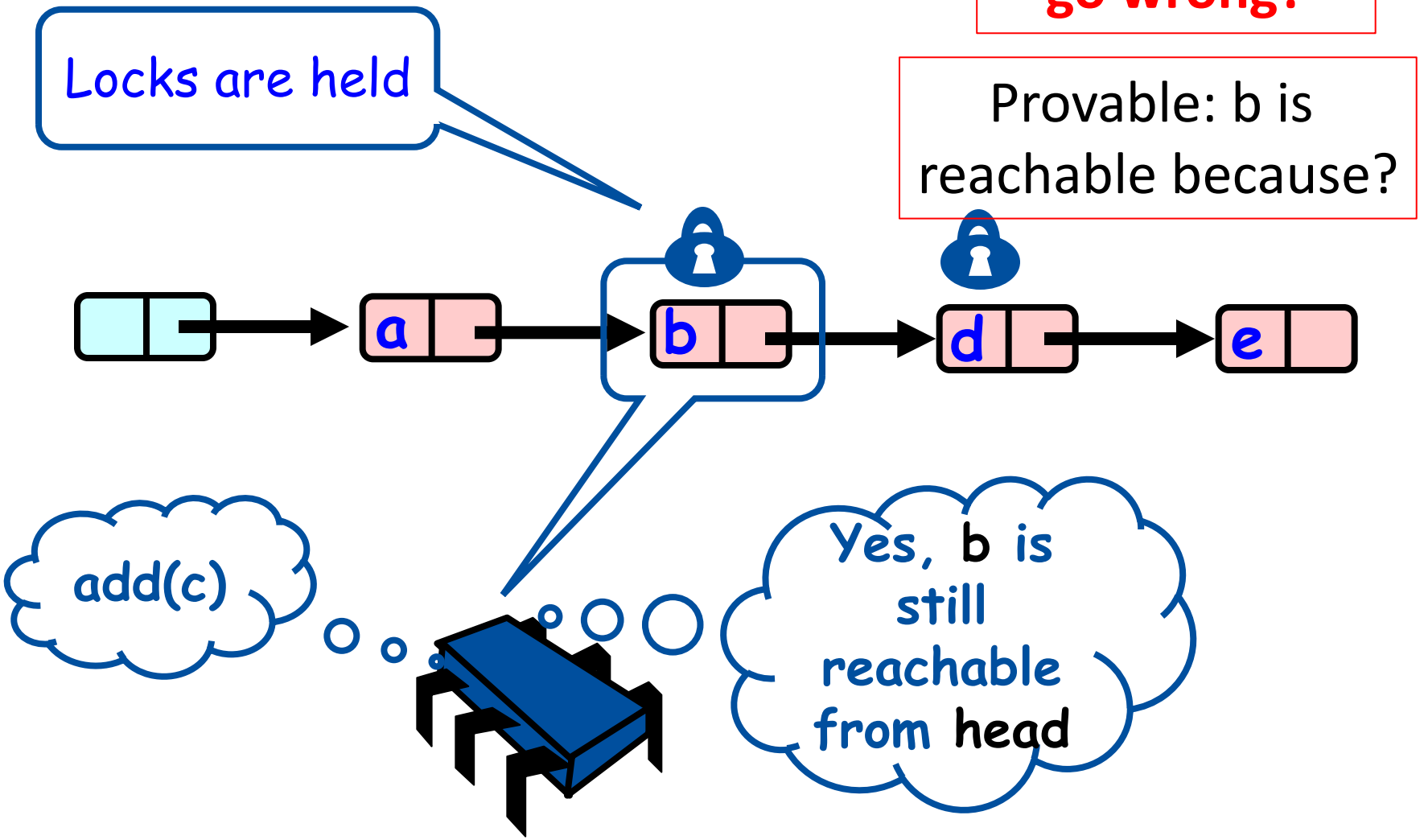
Optimistic: Validation 1



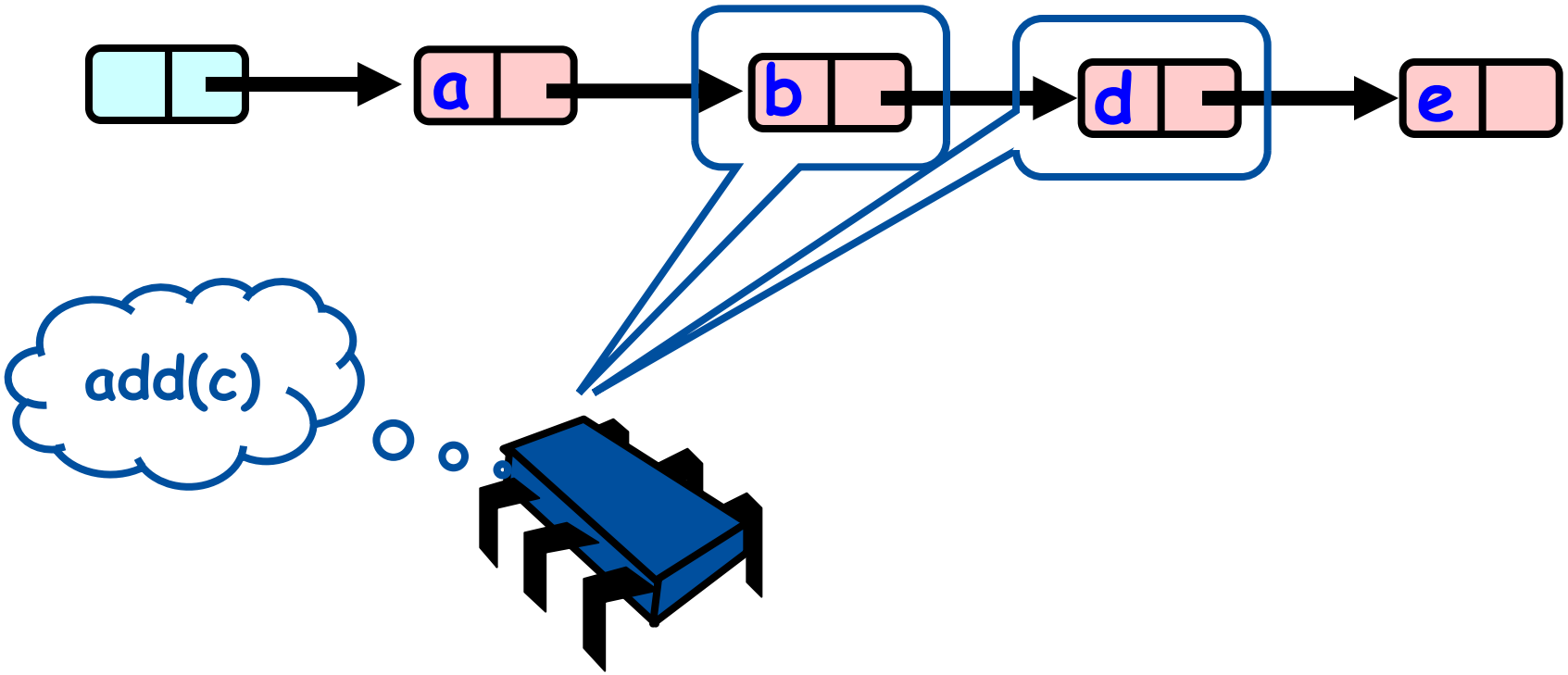
Optimistic: Validation 1

What else can go wrong?

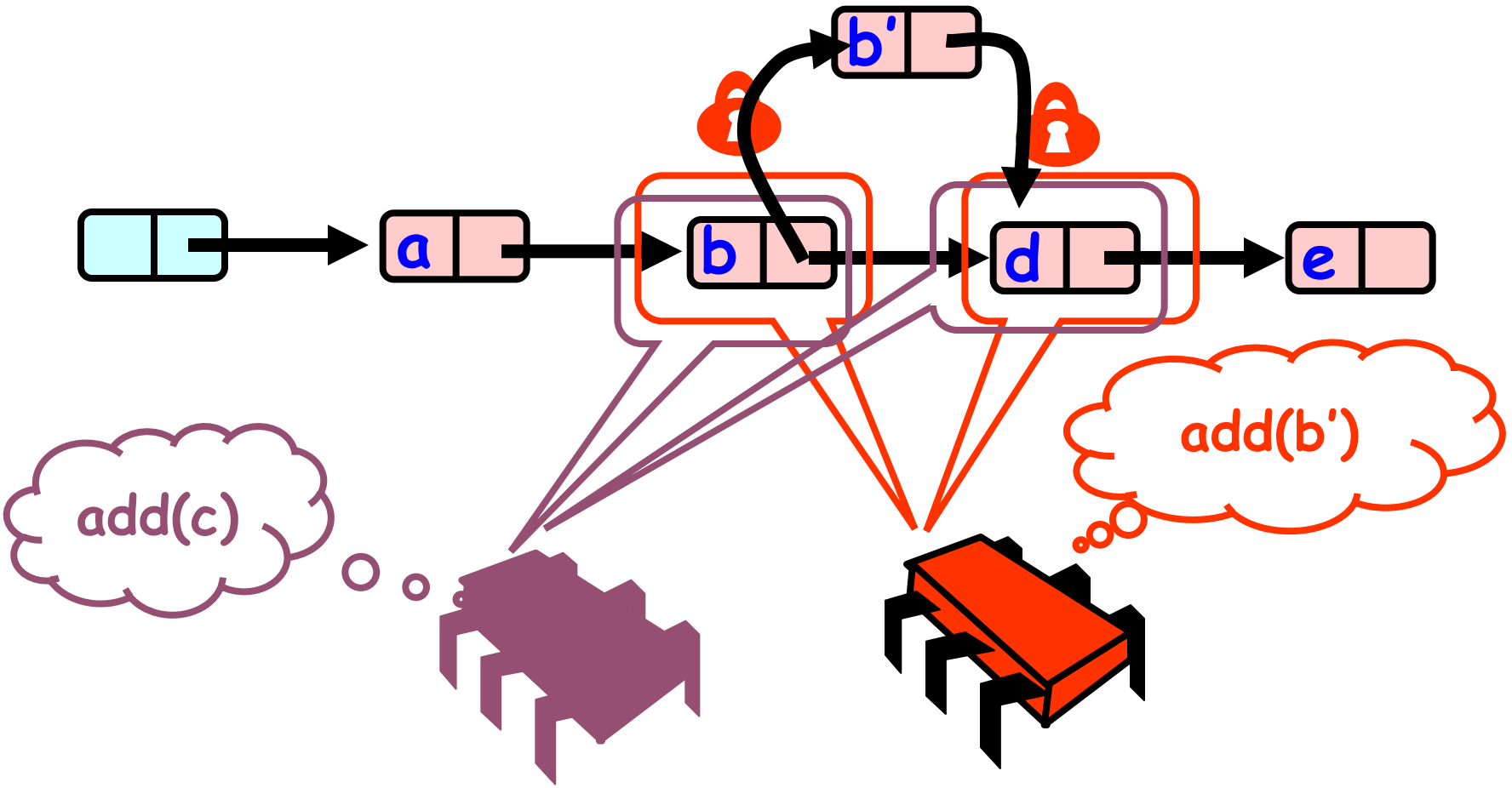
Provable: b is reachable because?



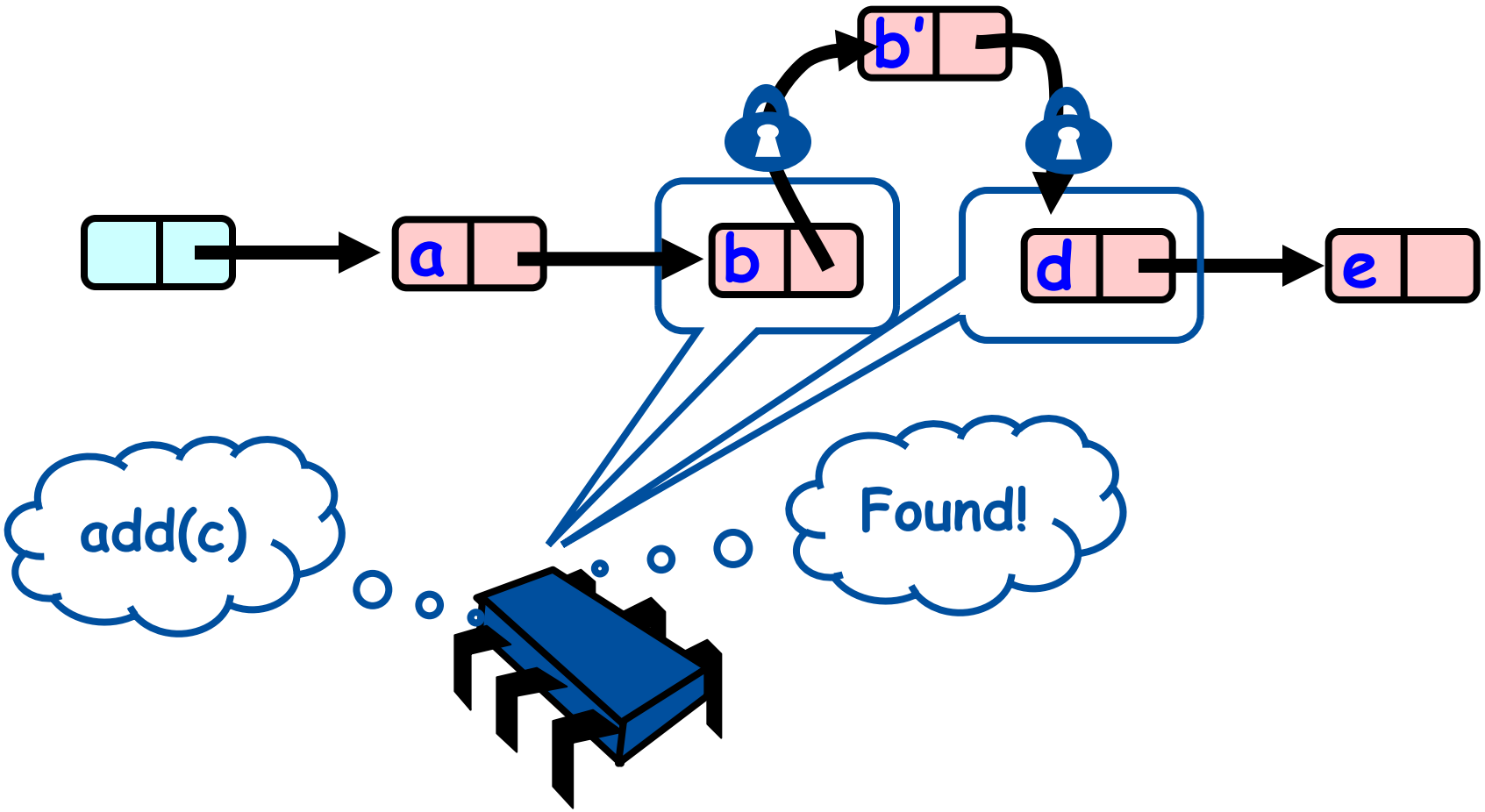
Optimistic: What else can go wrong?



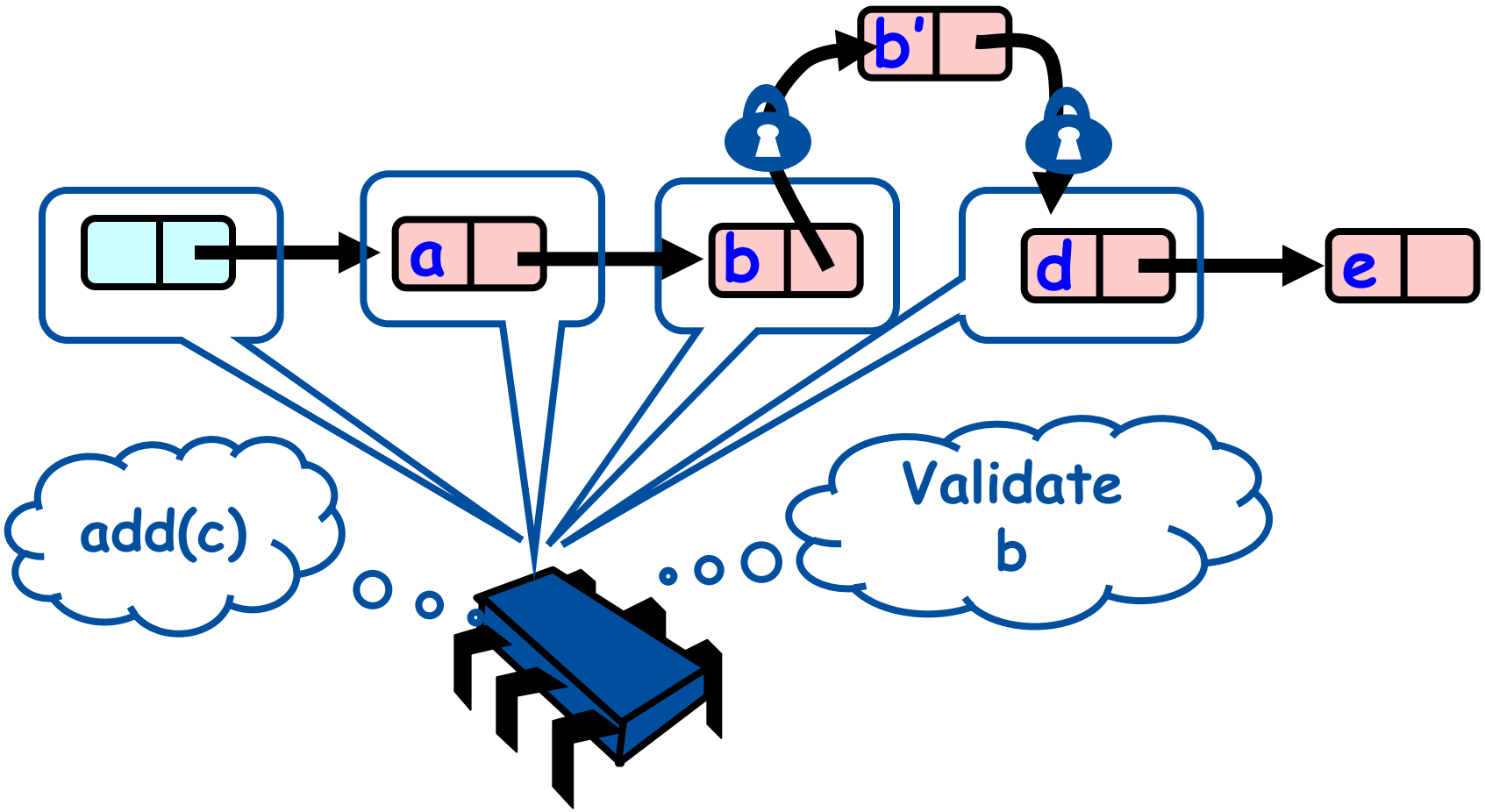
Optimistic: What else can go wrong?



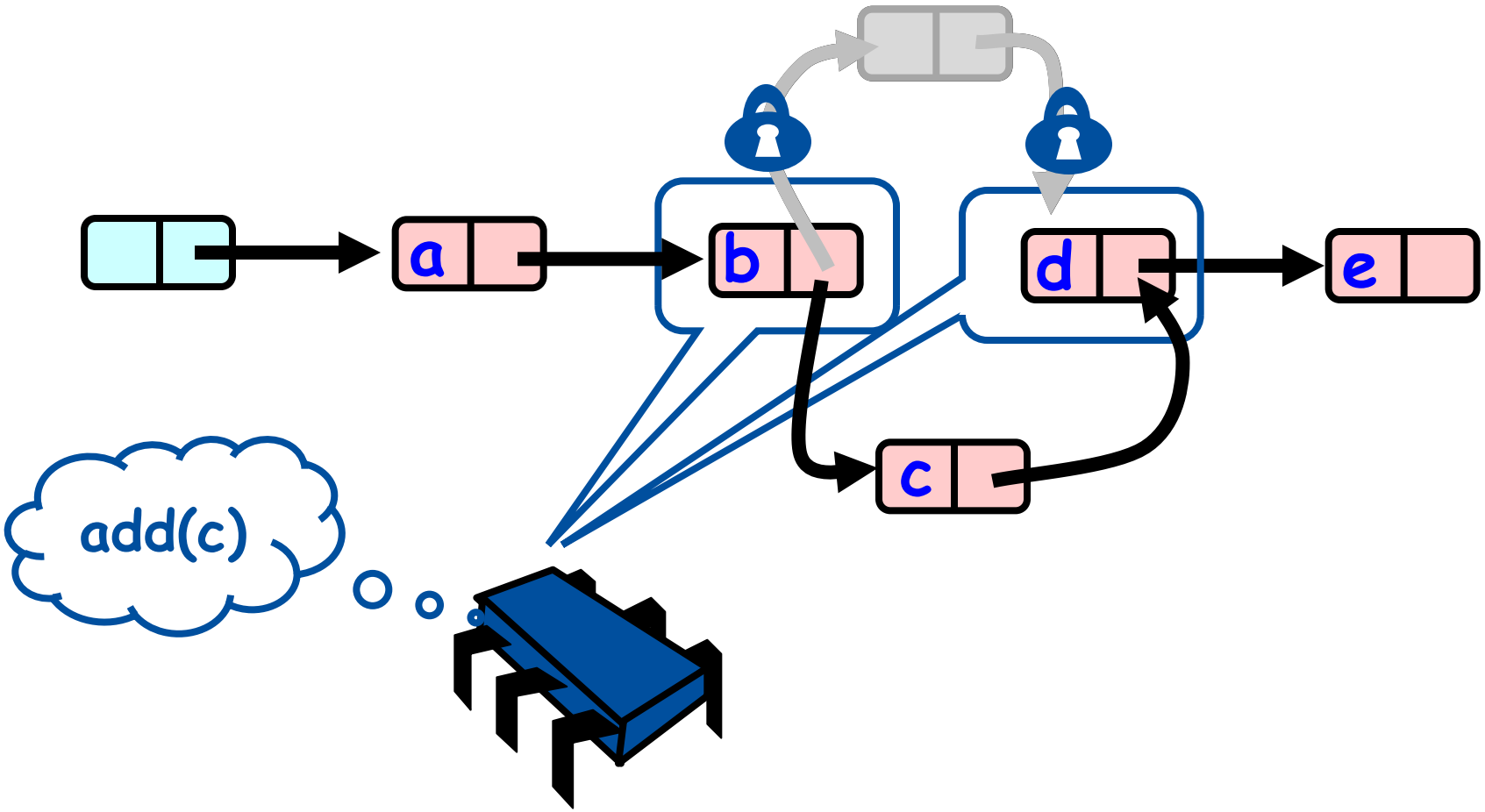
Optimistic: What else can go wrong?



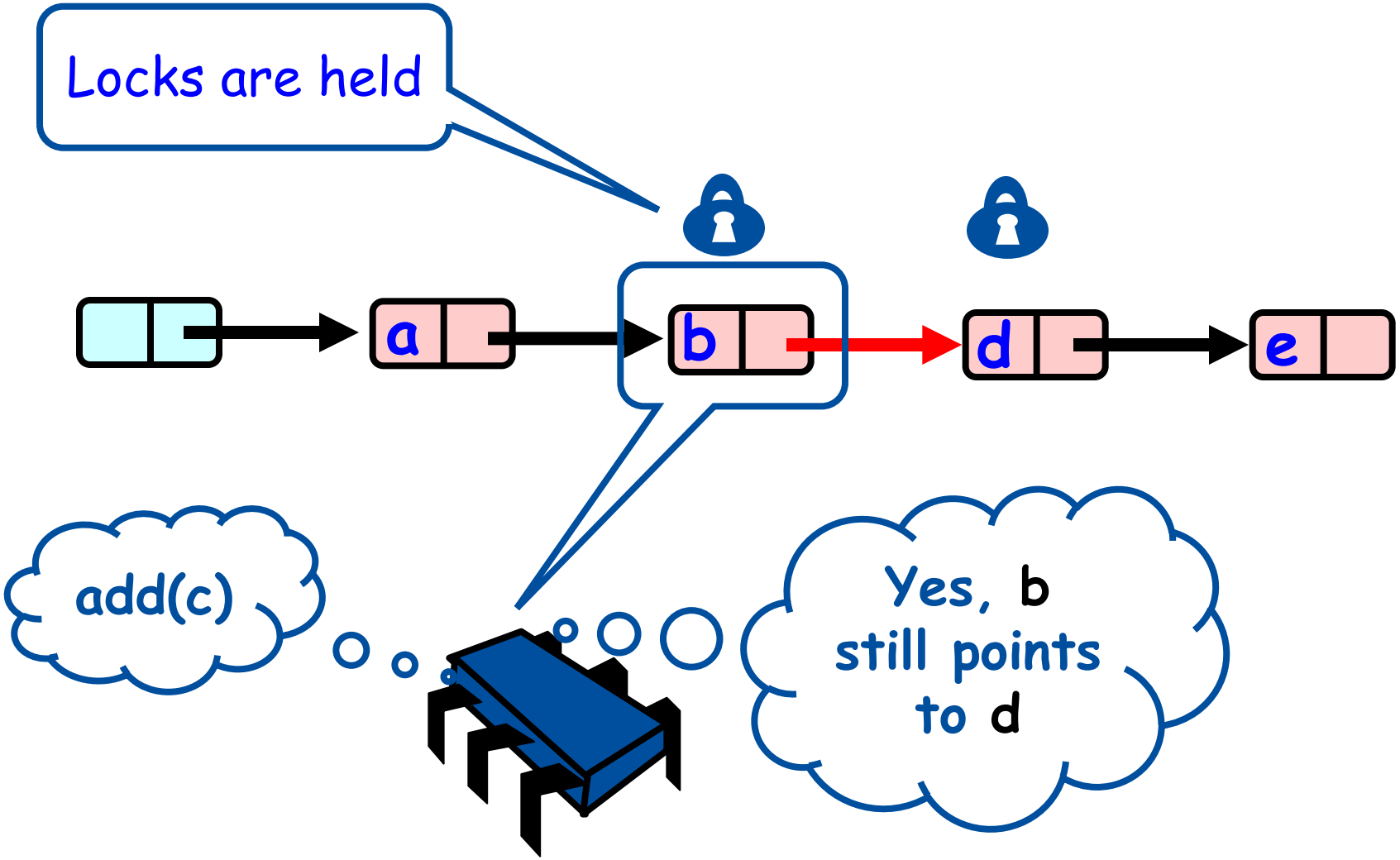
Optimistic: What else can go wrong?



Optimistic: What else can go wrong?



Optimistic: Validation 2



Optimistic Synchronization

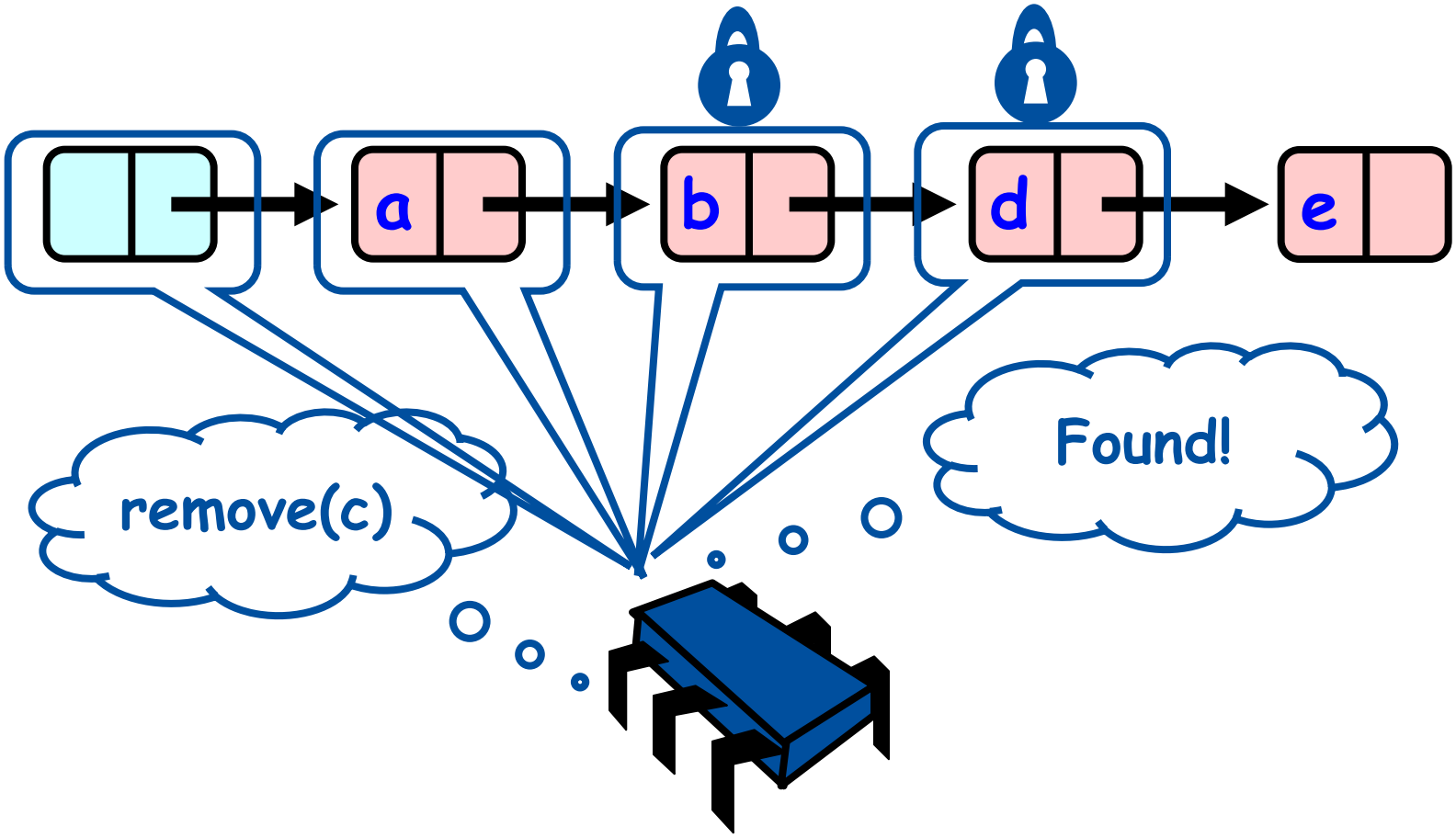
- Perform search without locking
- If found required nodes, lock them
- Validate that the locked nodes are correct
  - First node is reachable from head
  - First node points to the second node
- If conflict detected, unlock nodes and start over
- If everything okay, proceed with add/remove

Correctness

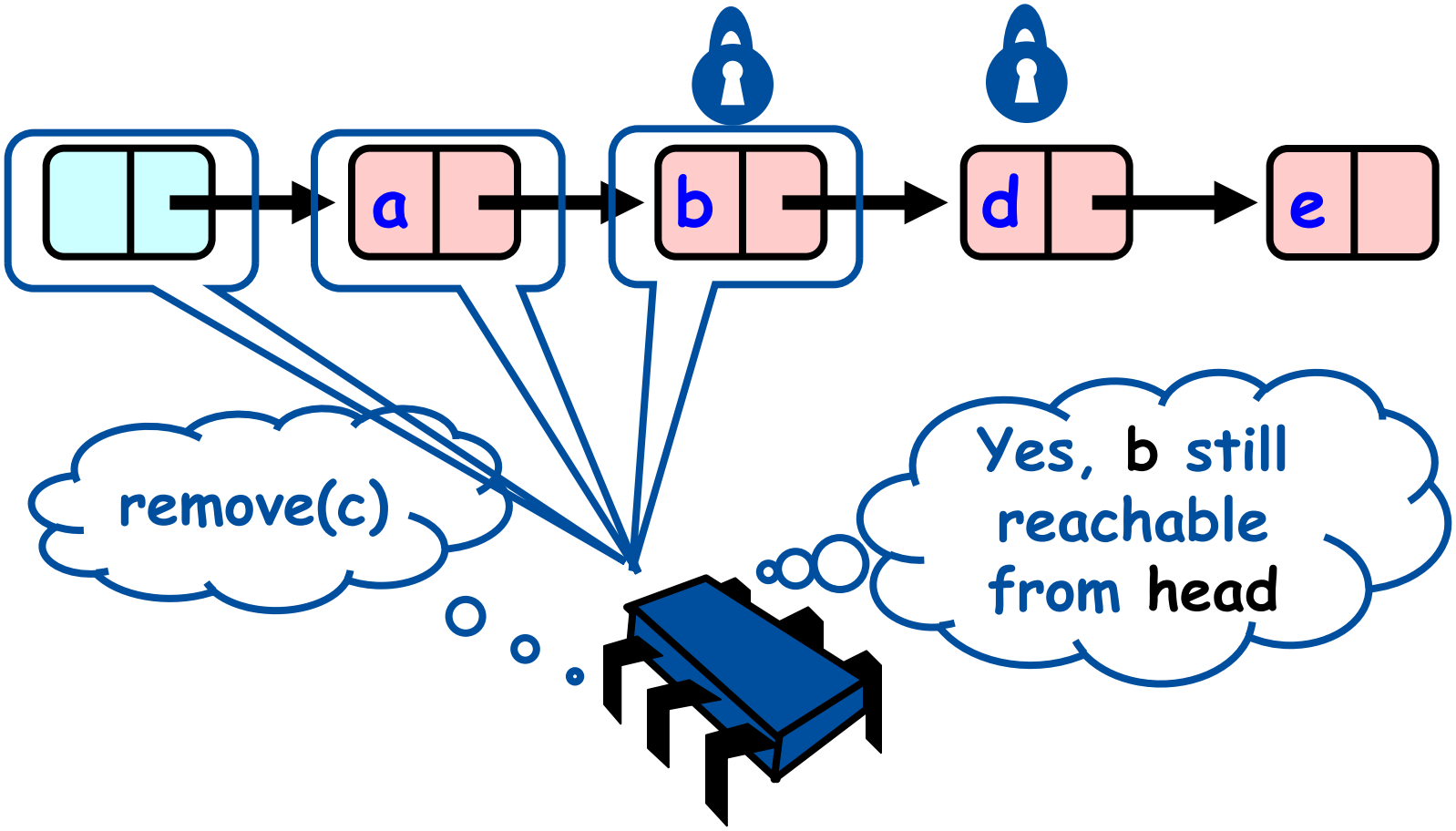
- We may traverse deleted nodes
- Correctness properties established by validation after we lock target nodes

- If nodes b and c both locked AND node b still accessible AND node c still successor to b, then
- Neither b nor c will be deleted
- Hence, okay to delete and return true

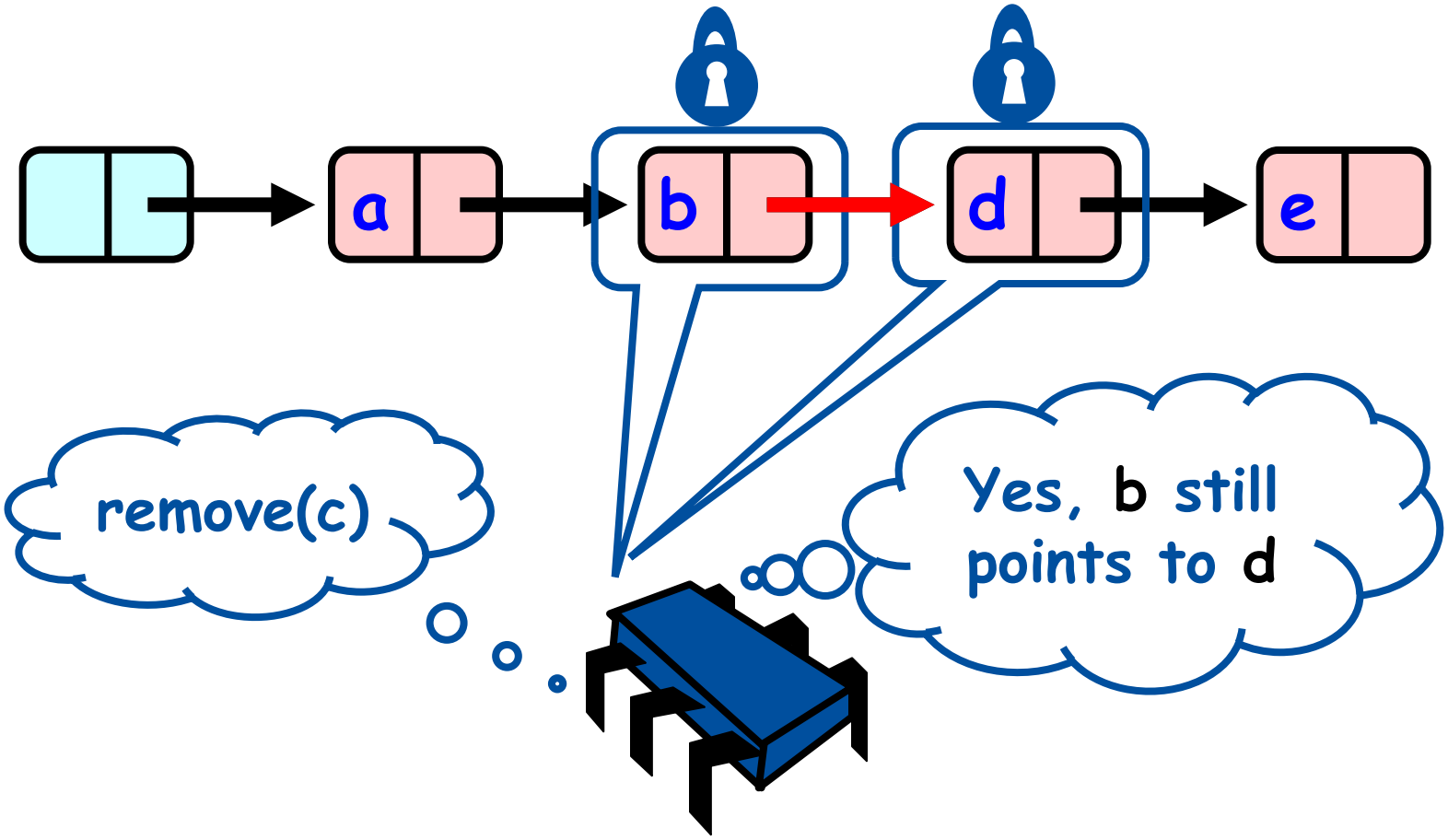
Optimistic: Unsuccessful Remove



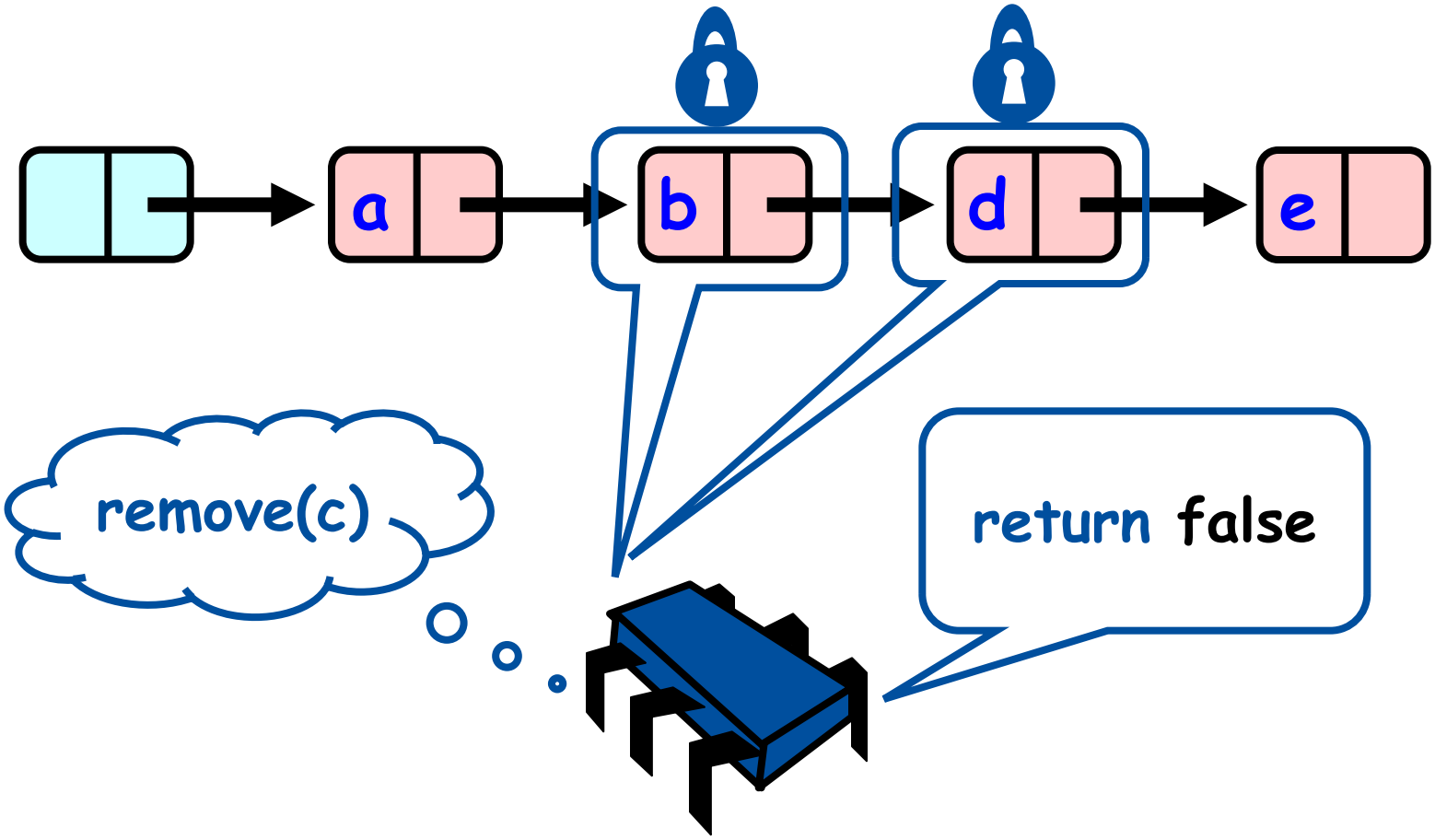
Optimistic: Validation 1



Optimistic: Validation 2



Optimistic: Validation Successful



Optimistic Synchronization: Validation

```
67   private boolean validate(Node pred, Node curr) {
68     Node node = head;
69     while (node.key <= pred.key) {
70       if (node == pred)
71         return pred.next == curr;
72       node = node.next;
73     }
74     return false;
75   }
```

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4       Node pred = head;
5       Node curr = pred.next;
6       while (curr.key <= key) {
7         pred = curr; curr = curr.next;
8       }
9       pred.lock(); curr.lock();
10      try {
11        if (validate(pred, curr)) {
12          if (curr.key == key) {
13            return false;
14          } else {
15            Node node = new Node(item);
16            node.next = curr;
17            pred.next = node;
18            return true;
19          }
20        }
21      } finally {
22        pred.unlock(); curr.unlock();
23      }
24    }
25  }
```

```
26  public boolean remove(T item) {
27    int key = item.hashCode();
28    while (true) {
29      Node pred = head;
30      Node curr = pred.next;
31      while (curr.key < key) {
32        pred = curr; curr = curr.next;
33      }
34      pred.lock(); curr.lock();
35      try {
36        if (validate(pred, curr)) {
37          if (curr.key == key) {
38            pred.next = curr.next;
39            return true;
40          } else {
41            return false;
42          }
43        }
44      } finally {
45        pred.unlock(); curr.unlock();
46      }
47    }
48  }
```

Optimistic Synchronization: Contains

```
49  public boolean contains(T item) {
50    int key = item.hashCode();
51    while (true) {
52     Node pred = this.head; // sentinel node;
53     Node curr = pred.next;
54     while (curr.key < key) {
55      pred = curr; curr = curr.next;
56      }
57      try {
58        pred.lock(); curr.lock();
59        if (validate(pred, curr)) {
60          return (curr.key == key);
61        }
62      } finally {                    // always unlock
63        pred.unlock(); curr.unlock();
64      }
65    }
66  }
```

ation
ed?

Optimistic Synchronization

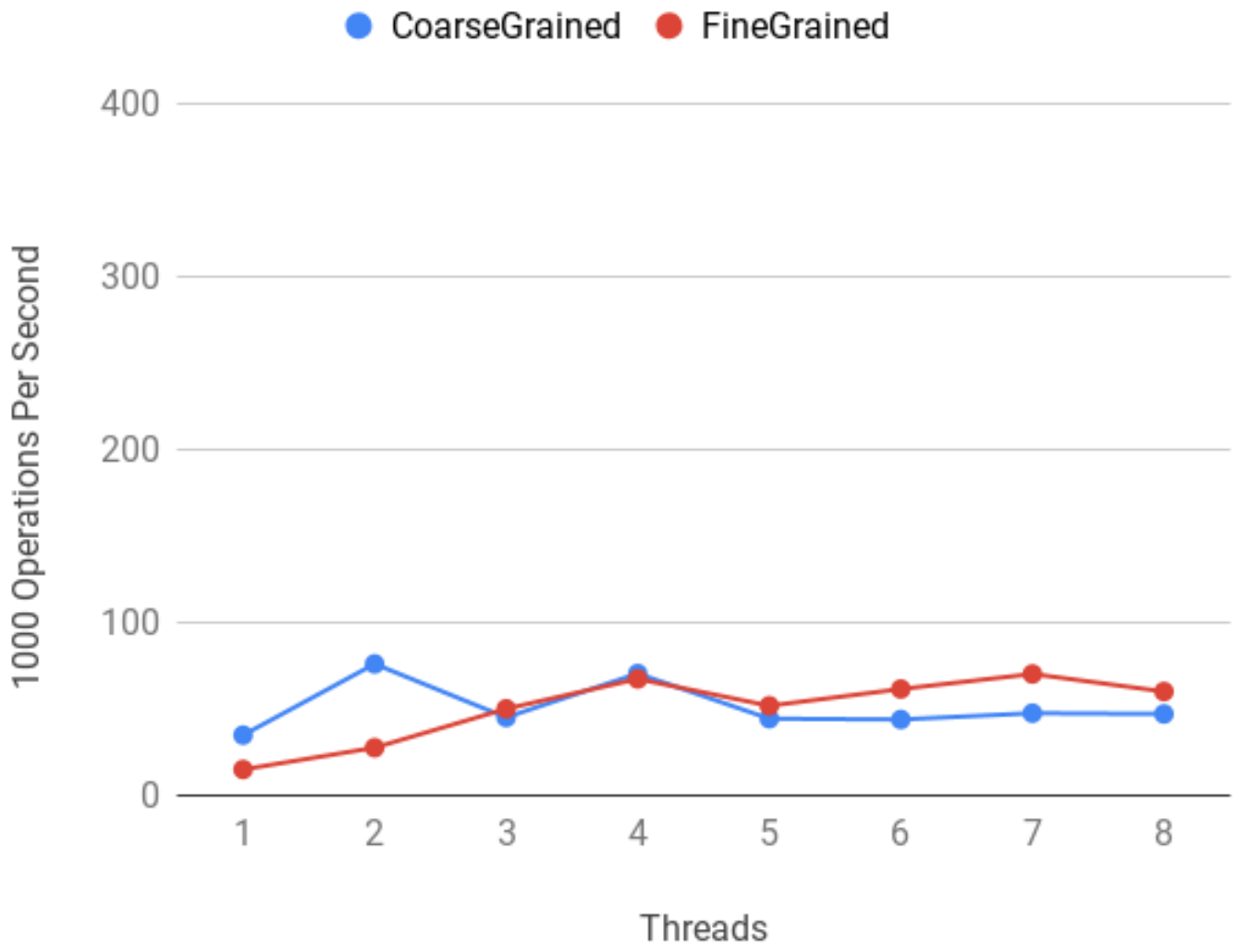
- Limited contention
  - Contention only at targets of add(), remove(), contains()
- Traversals are wait-free

- Much less lock acquisition/release
  - Performance
  - Concurrency

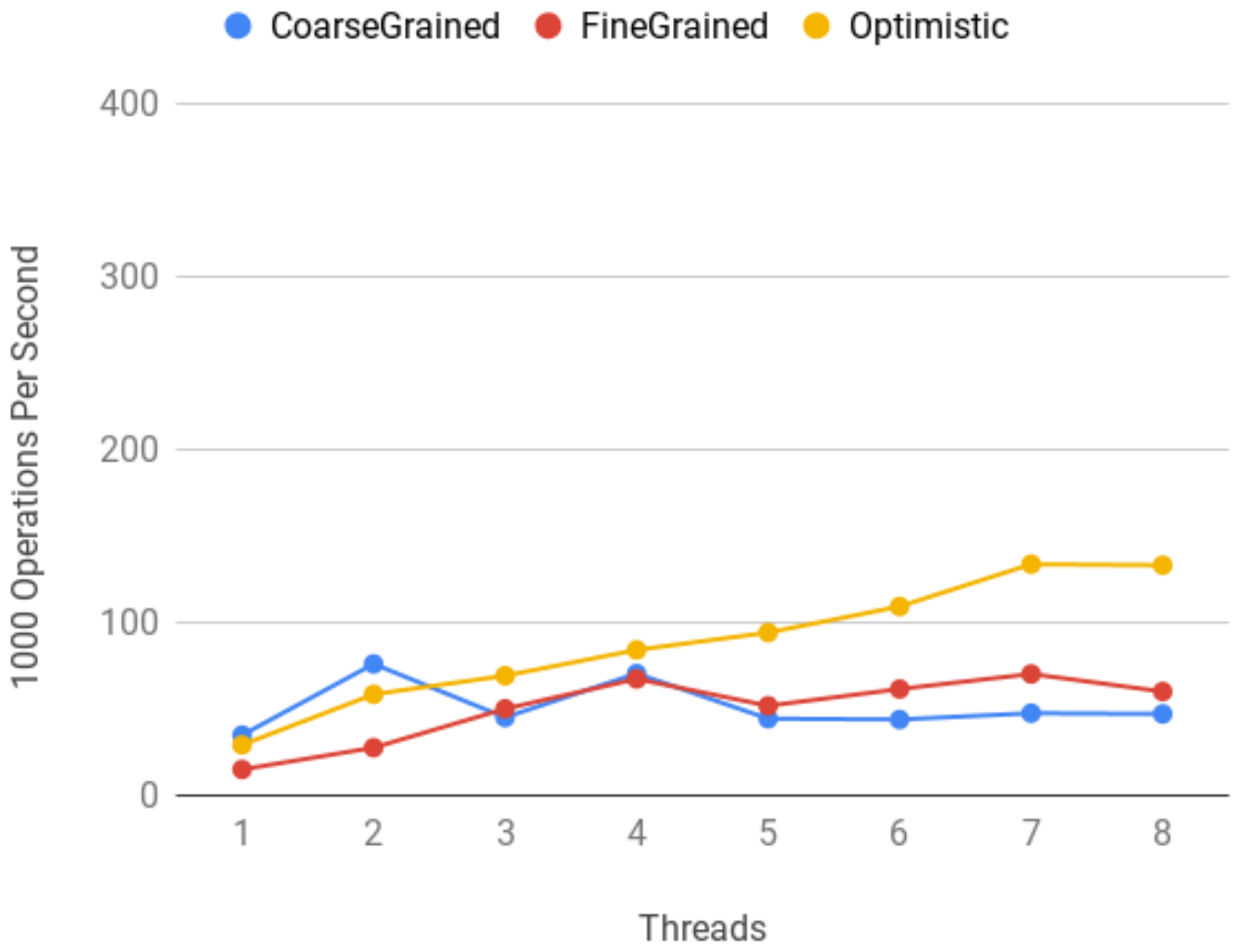
Optimistic Synchronization: Issues

- Need to traverse list twice
- Effective when:
  - Cost of scanning twice without locks is less than cost of scanning once with locks
- contains() method acquires locks
  - Majority of calls in many applications

Optimistic Synchronization



Optimistic Synchronization



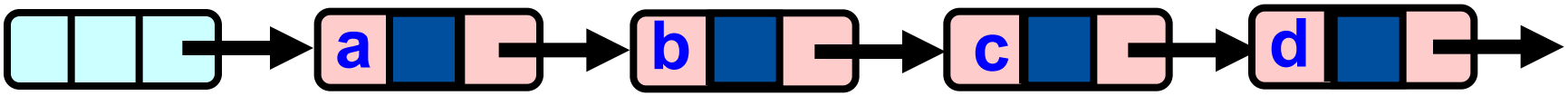
Lazy Synchronization

- Approach similar to optimistic list, except that we scan only once (and that contains never locks)
- Removing nodes causes trouble, so we will do it lazily

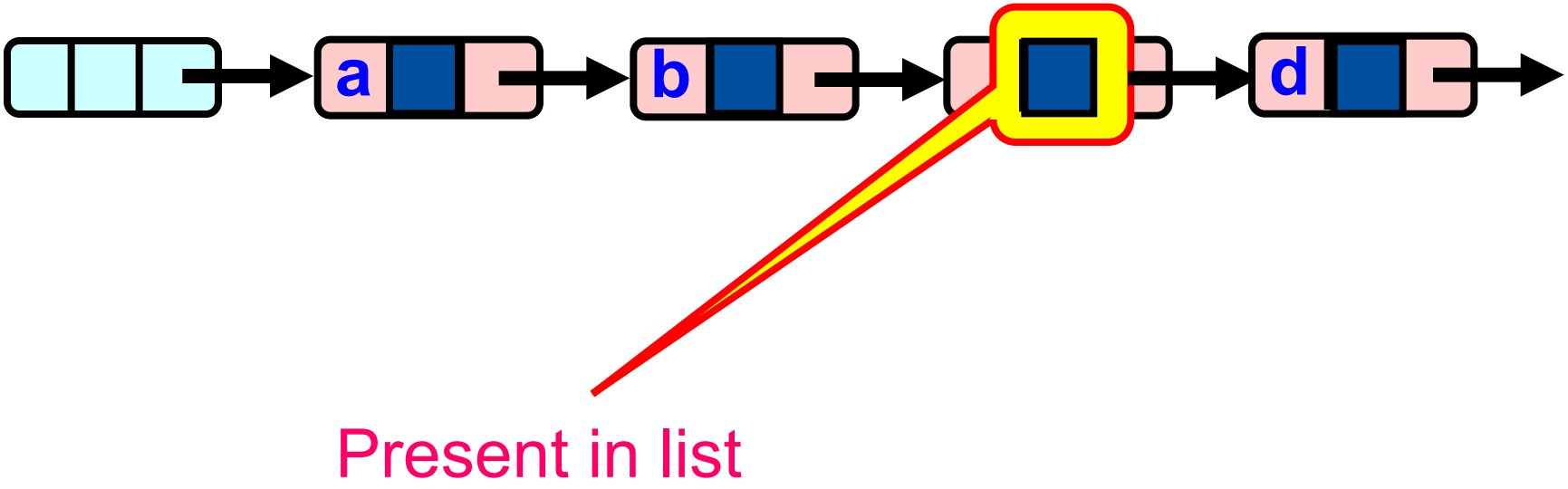
- Split remove into logical remove and physical remove
- Logical remove marks the current node as removed
- Physical remove redirects predecessor's next

- Invariant: Item in the set if it is reachable and not marked

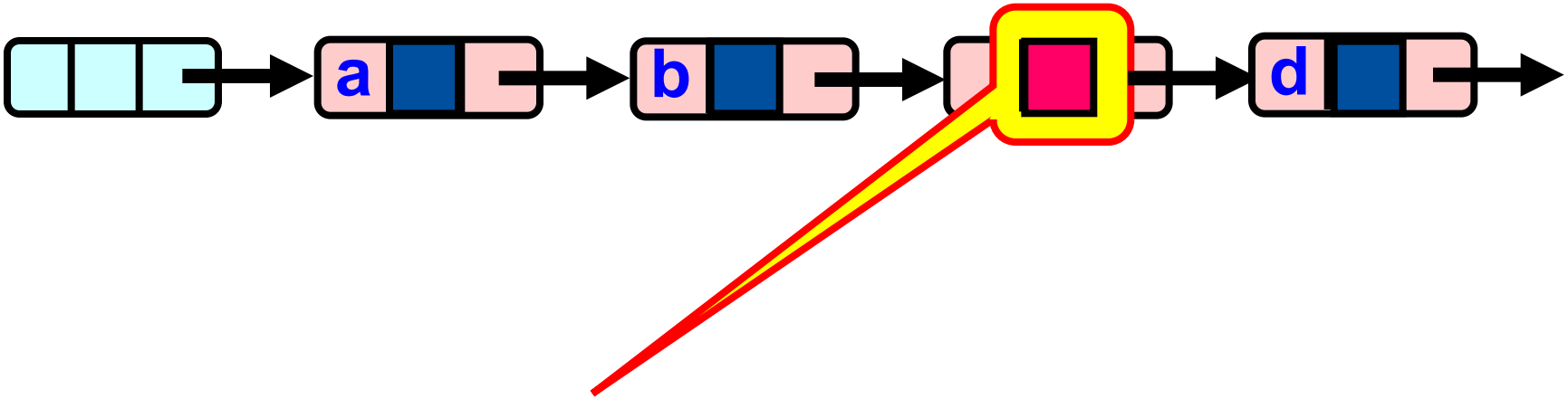
Lazy Removal



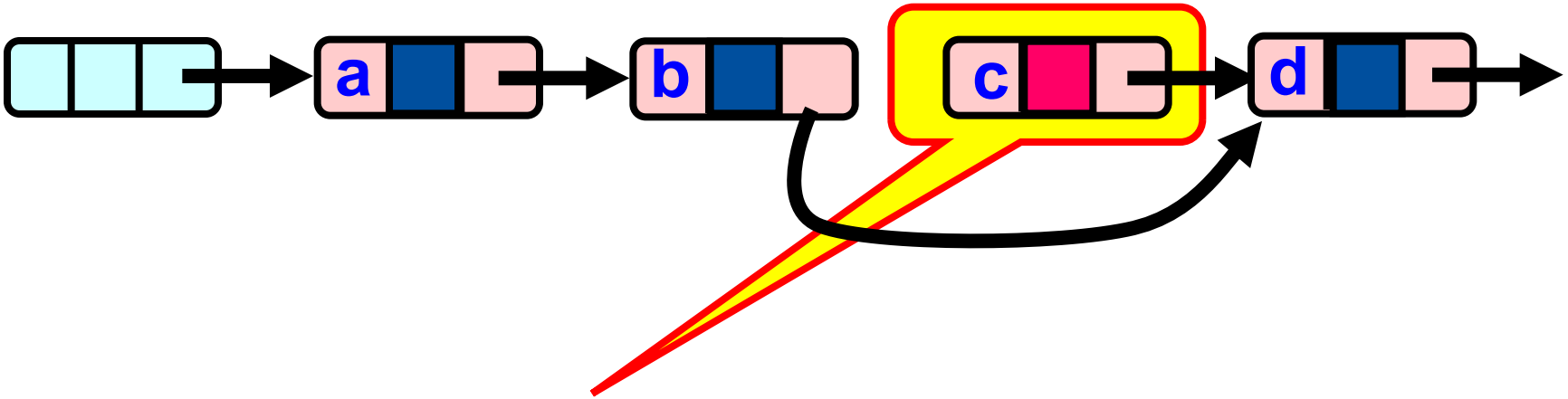
Lazy Removal



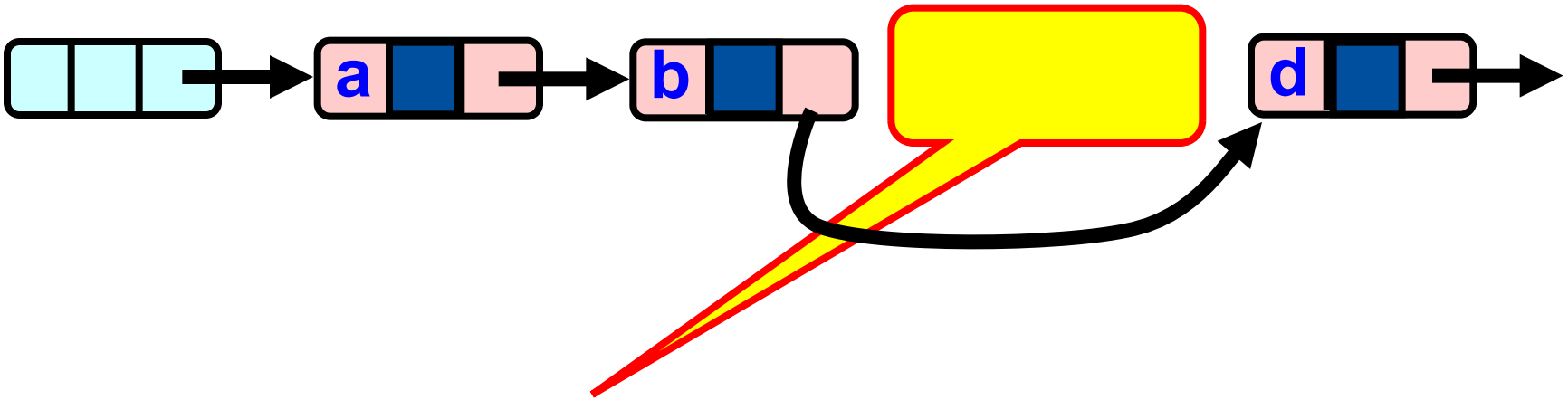
Lazy Removal



Lazy Removal



Lazy Removal

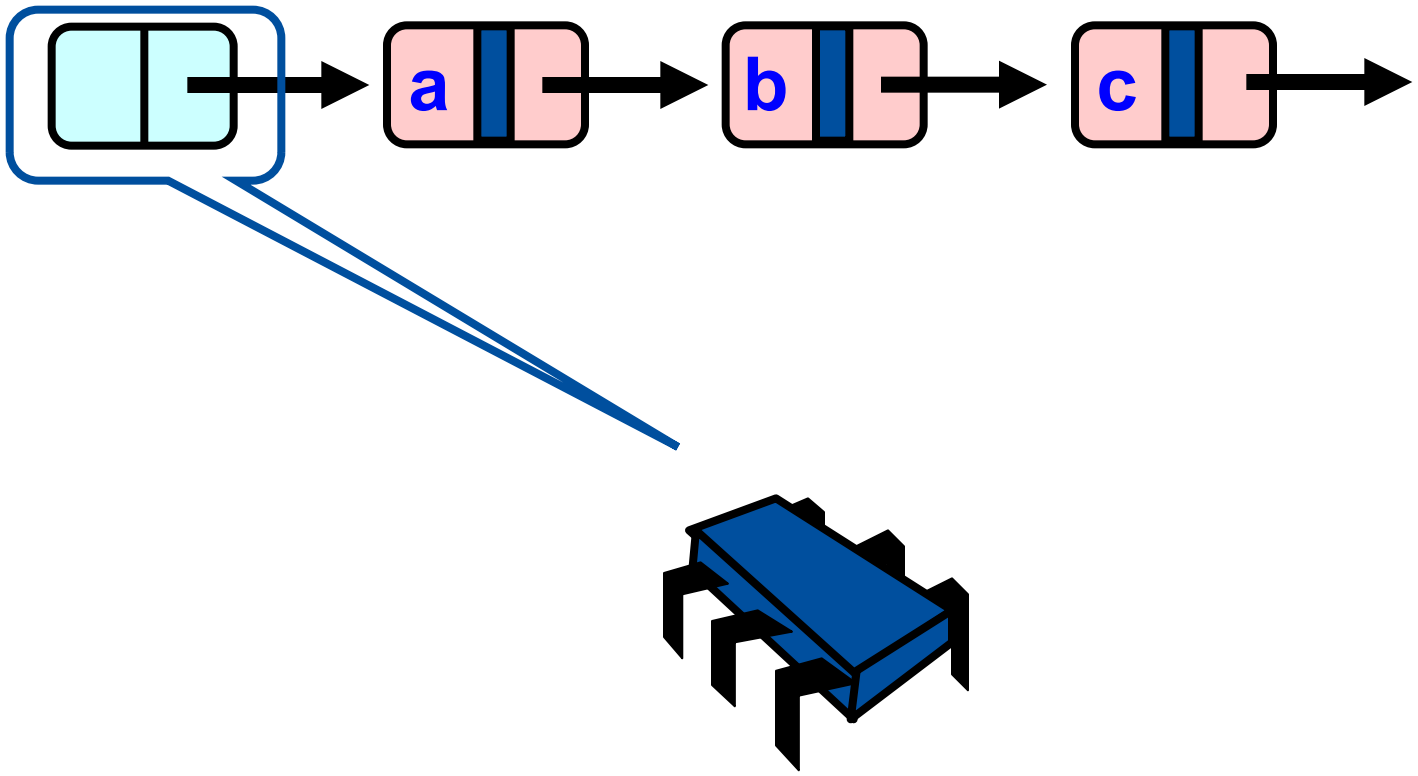


Lazy Synchronization

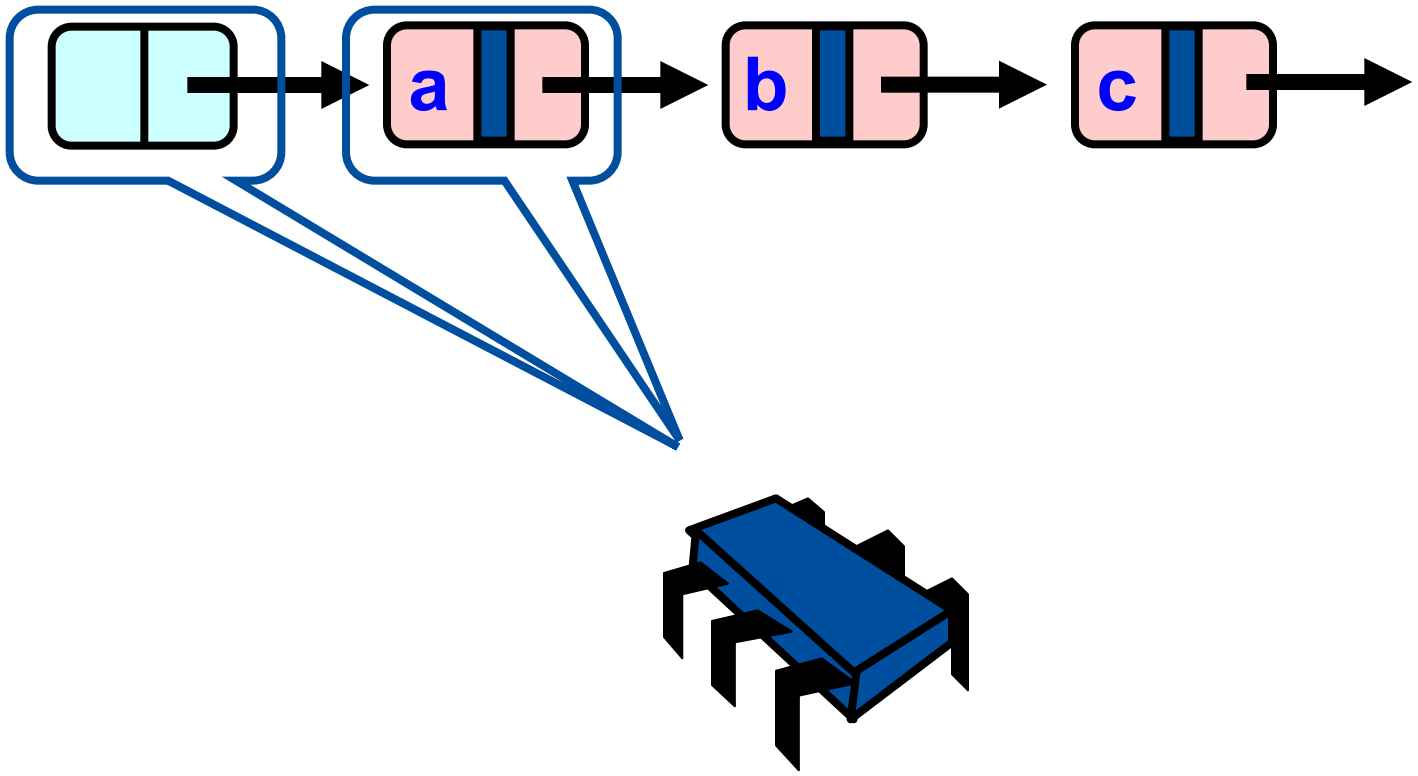
- All methods scan through locked and marked nodes
- Removing a node shouldn't slow down other method calls
  - How?
- Must still lock pred and curr nodes

- Validation doesn't need to rescan the list
- Validation checks:
  - pred is not marked
  - curr is not marked
  - pred points to curr

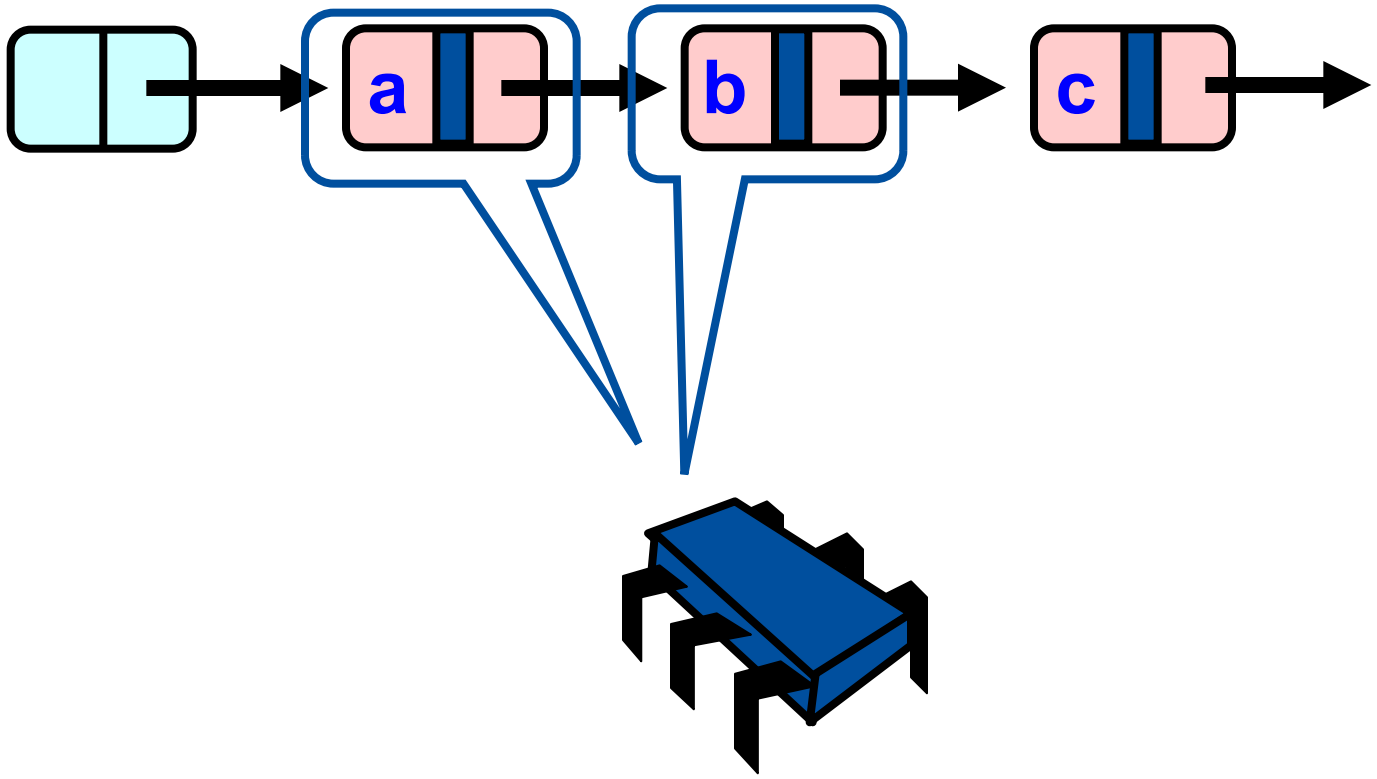
Lazy Synchronization



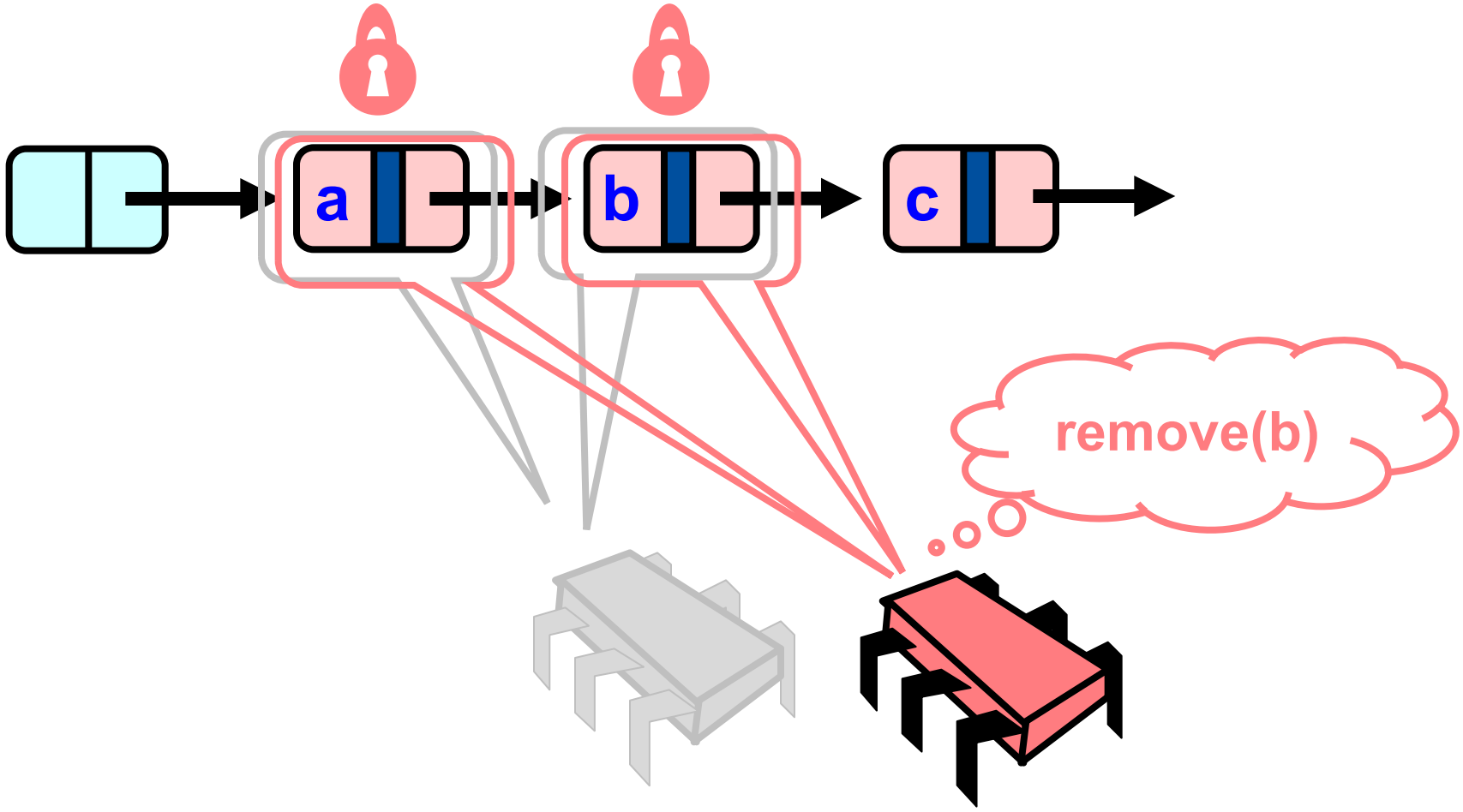
Lazy Synchronization



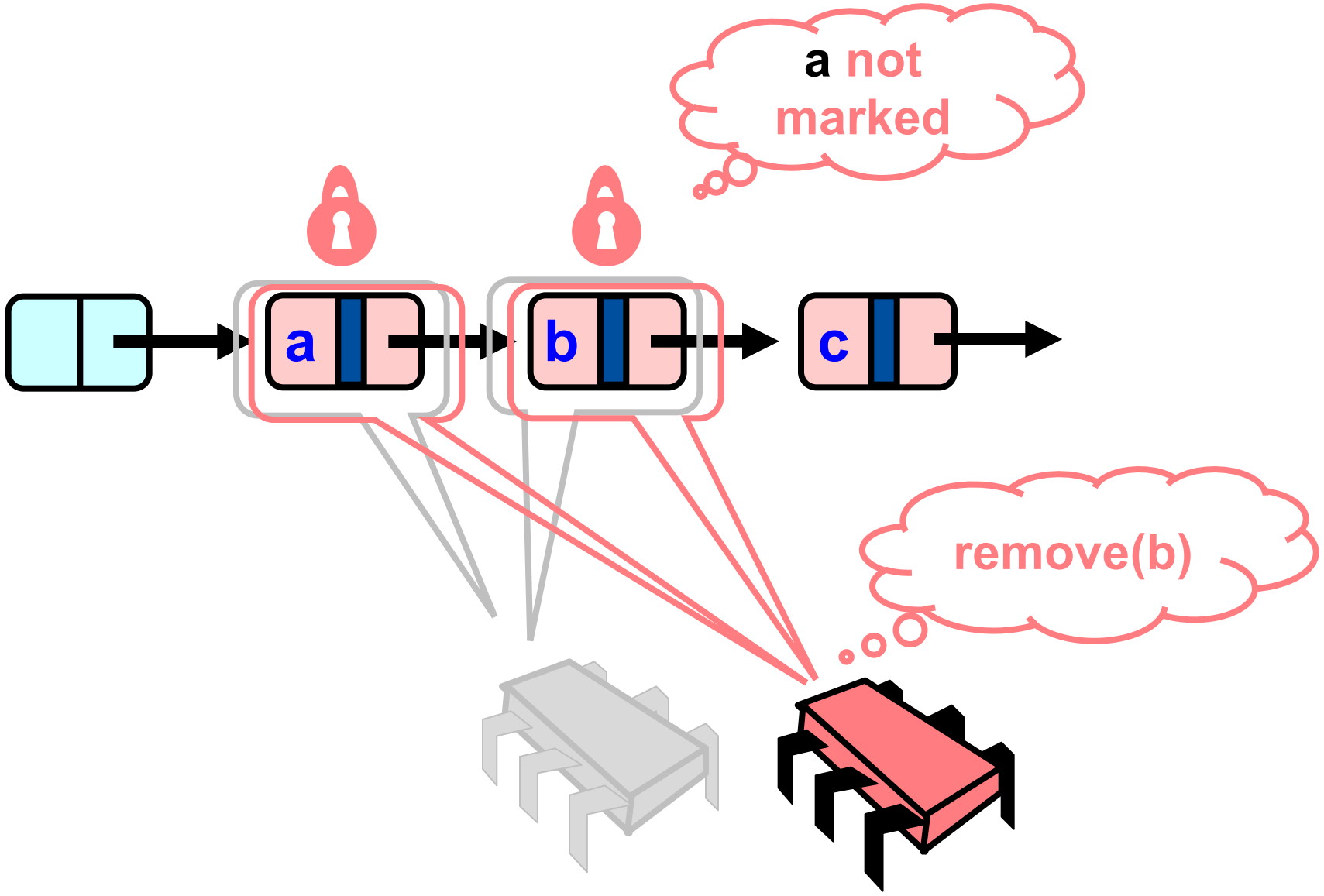
Lazy Synchronization



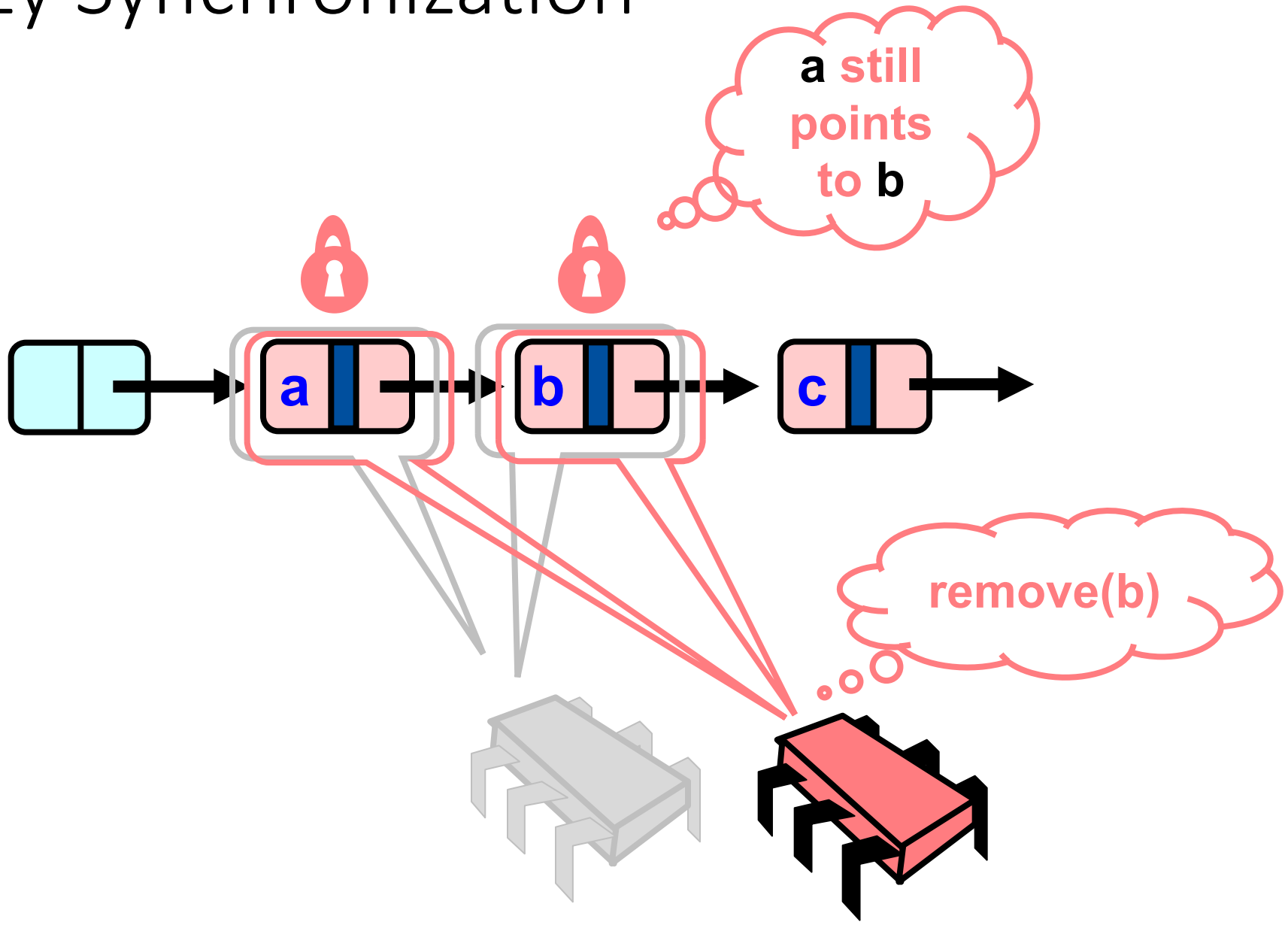
Lazy Synchronization



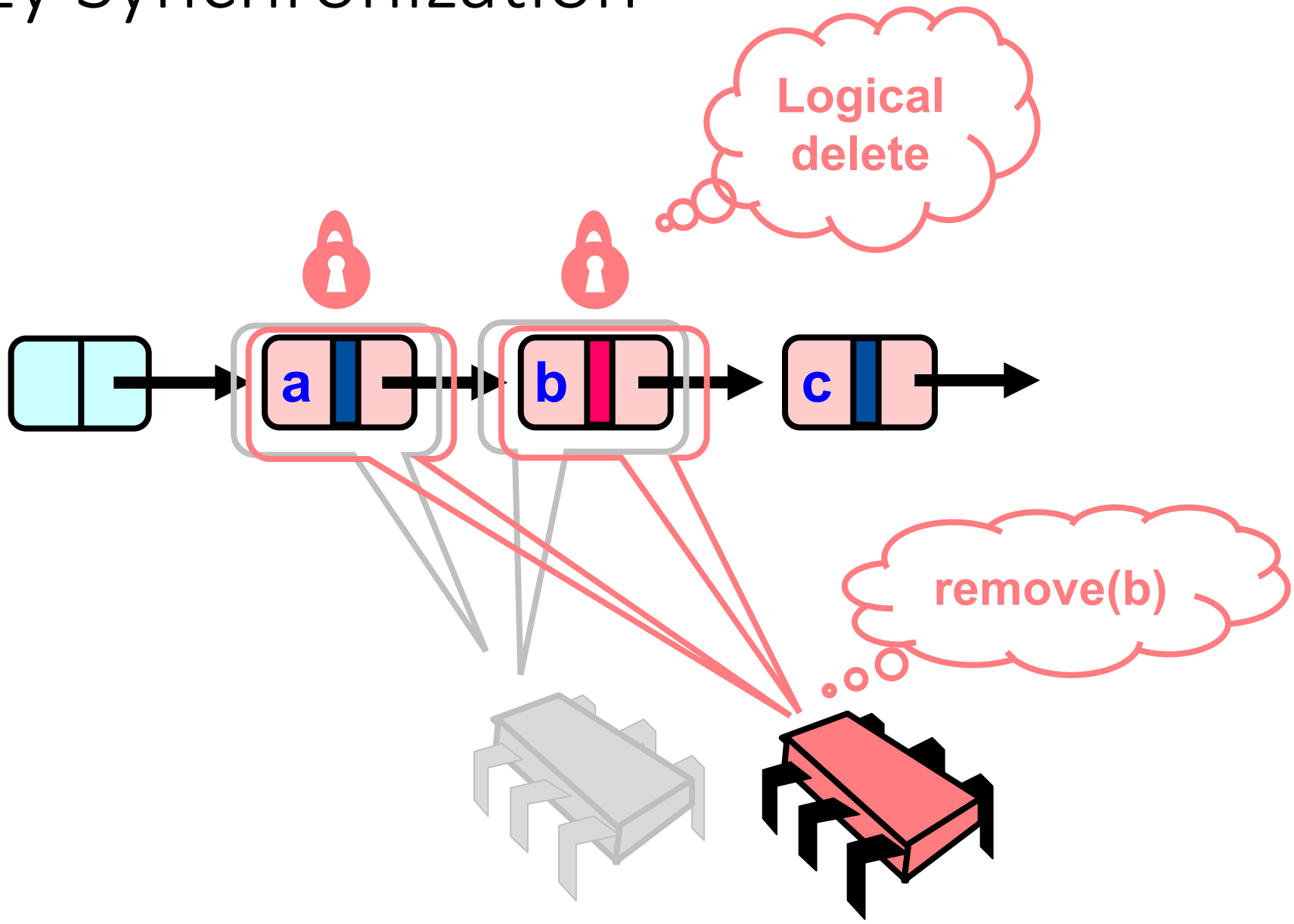
Lazy Synchronization



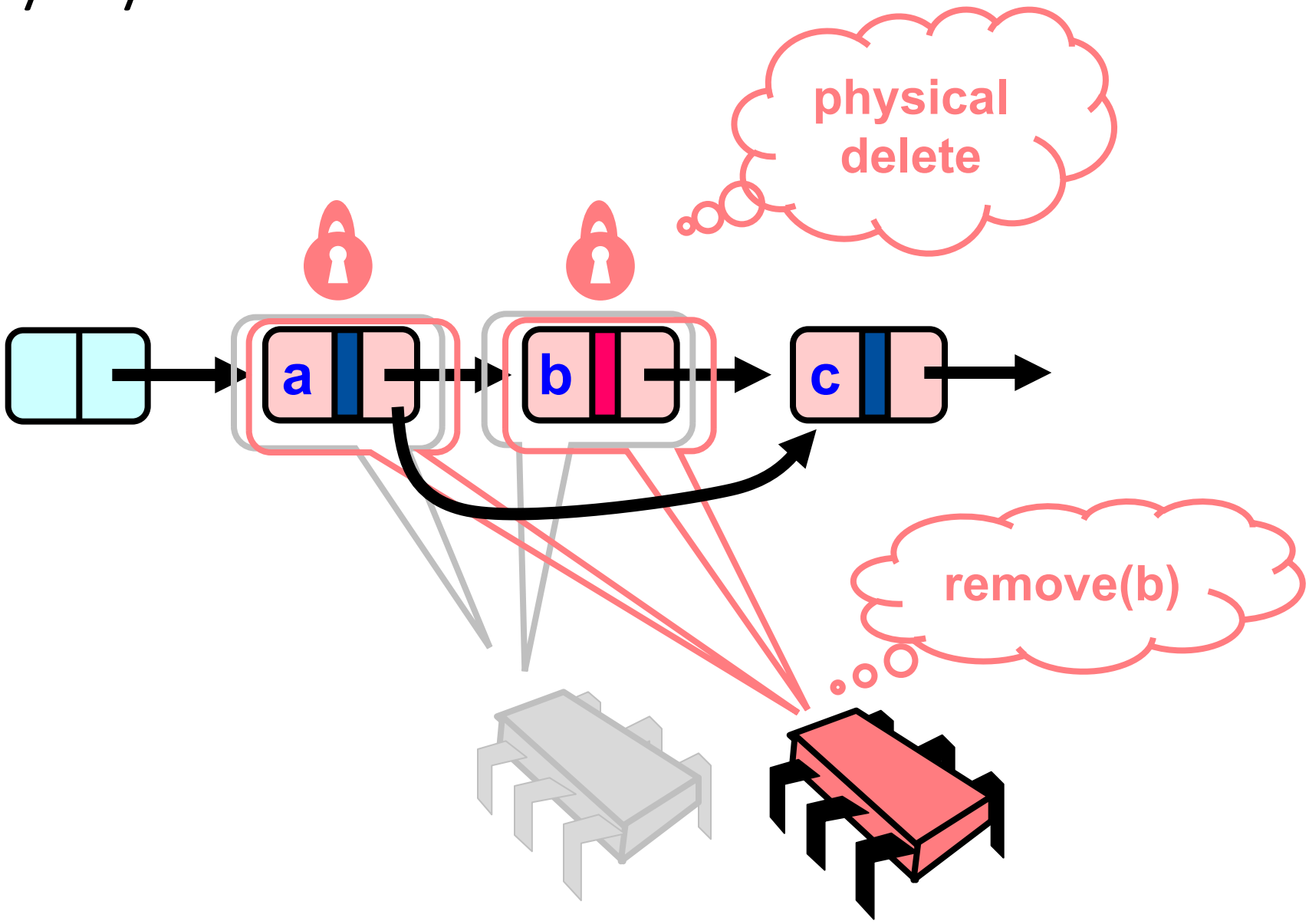
Lazy Synchronization



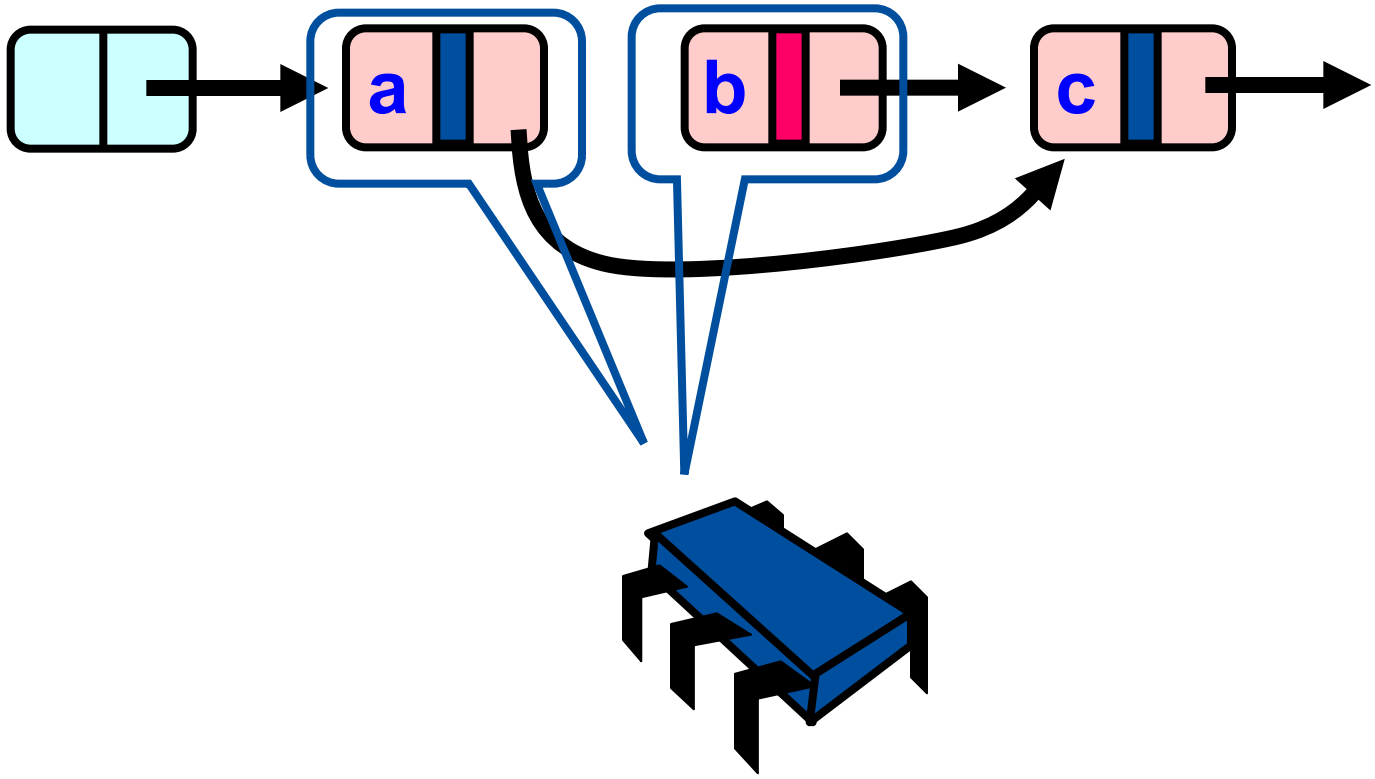
Lazy Synchronization



Lazy Synchronization



Lazy Synchronization



Lazy Synchronization

- Invariant: Item in the set if it is reachable and not marked

```
1    private boolean validate(Node pred, Node curr) {
2      return !pred.marked && !curr.marked && pred.next == curr;
3    }
```

Lazy Synchronization

- Invariant: Item in the set if it is reachable and not marked

```
1    private boolean validate(Node pred, Node curr) {
2      return !pred.marked && !curr.marked && pred.next == curr;
3    }
```

```
1    public boolean contains(T item) {
2      int key = item.hashCode();
3      Node curr = head;
4      while (curr.key < key)
5        curr = curr.next;
6      return curr.key == key && !curr.marked;
7    }
```

contains is wait-free

```
1   public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4       Node pred = head;
5       Node curr = head.next;
6       while (curr.key < key) {
7        pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
13          if (validate(pred, curr)) {
14            if (curr.key == key) {
15              return false;
16            } else {
17              Node node = new Node(item);
18              node.next = curr;
19              pred.next = node;
20              return true;
21            }
22          }
23        } finally {
24          curr.unlock();
25        }
26      } finally {
27        pred.unlock();
28      }
29    }
30  }
```

```
1   public boolean remove(T item) {
2     int key = item.hashCode();
3     while (true) {
4      Node pred = head;
5      Node curr = head.next;
6      while (curr.key < key) {
7       pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
13          if (validate(pred, curr)) {
14            if (curr.key != key) {
15              return false;
16            } else {
17              curr.marked = true;
18              pred.next = curr.next;
19              return true;
20            }
21          }
22        } finally {
23          curr.unlock();
24        }
25      } finally {
26        pred.unlock();
27      }
28    }
29  }
```

```
1   public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4       Node pred = head;
5       Node curr = head.next;
6       while (curr.key < key) {
7        pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
13          if (validate(pred, curr)) {
14            if (curr.key ==
15              return false;
16            } else {
17              Node node = new Node(item);
18              node.next = curr;
19              pred.next = node;
20              return true;
21            }
22          }
23        } finally {
24          curr.unlock();
25        }
26      } finally {
27        pred.unlock();
28      }
29    }
30  }
```

```
1   public boolean remove(T item) {
2     int key = item.hashCode();
3     while (true) {
4      Node pred = head;
5      Node curr = head.next;
6      while (curr.key < key) {
7       pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
          if (validate(pred, curr)) {
            if (curr.key != key) {
              return false;
            } else {
              curr.marked = true;
              pred.next = curr.next;
              return true;
            }
          finally {
            curr.unlock();
          }
        } finally {
          pred.unlock();
        }
      }
    }
```

Logical remove

Physical remove

```
1   public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4       Node pred = head;
5       Node curr = head.next;
6       while (curr.key < key) {
7        pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
13          if (validate(pred, curr)) {
14            if (curr.key == key) {
15              return false;
16            } else {
17              Node node = new Node(item);
18              node.next = curr;
19              pred.next = node;
20              return true;
21            }
22          }
23        } finally {
24          curr.unlock();
25        }
26      } finally {
27        pred.unlock();
28      }
29    }
30  }
```

```
1   public boolean remove(T item) {
2     int key = item.hashCode();
3     while (true) {
4      Node pred = head;
5      Node curr = head.next;
6      while (curr.key < key) {
7       pred = curr; curr = curr.next;
8       }
9       pred.lock();
10      try {
11        curr.lock();
12        try {
13          if (validate(pred, curr)) {
14            if (curr.key != key) {
15              return false;
16            } else {
17              curr.marked = true;
18              pred.next = curr.next;
19              return true;
20            }
21          }
22        } finally {
23          curr.unlock();
24        }
25      } finally {
26        pred.unlock();
27      }
28    }
29  }
```

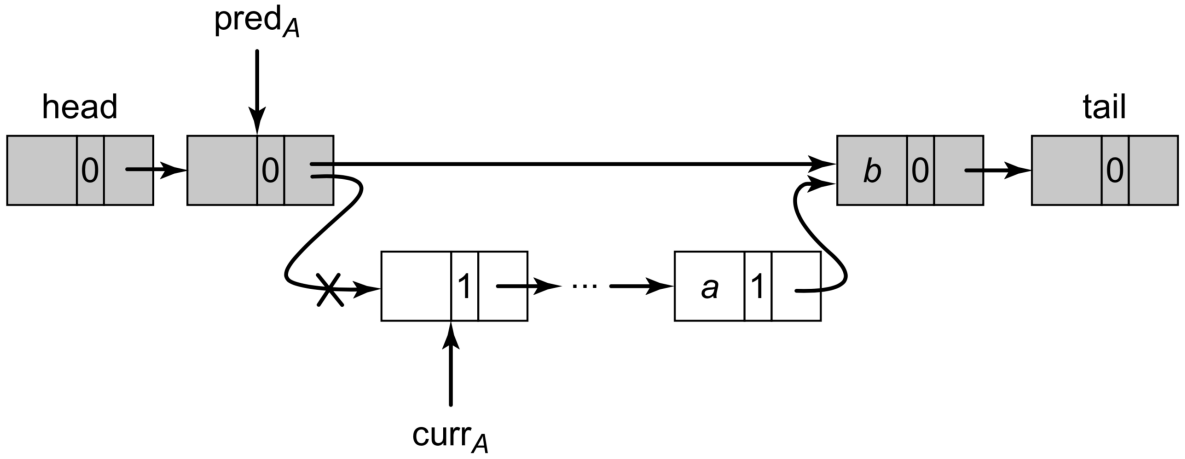
Linearization Point

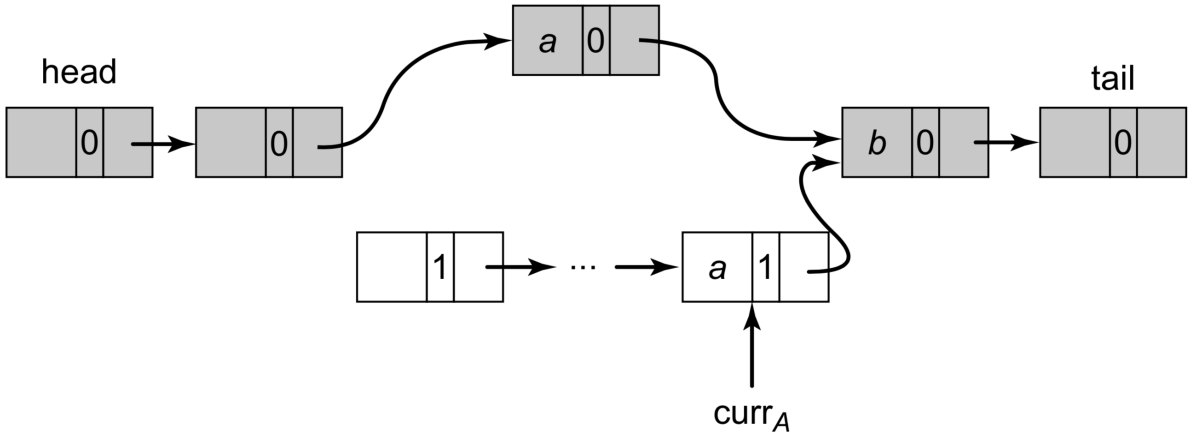
Lazy Synchronization - Contains

```
1    public boolean contains(T item) {
2      int key = item.hashCode();
3      Node curr = head;
4      while (curr.key < key)
5        curr = curr.next;
6      return curr.key == key && !curr.marked;
7    }
```

- When successful (i.e., when returns true), contains is linearized when an unmarked matching node is found
- When unsuccessful, contains is linearized when?
  - Traversal reaches a marked node with matching key, or a node with key greater than one being searched
  - Does this capture every unsuccessful call?

Lazy Synchronization - Contains





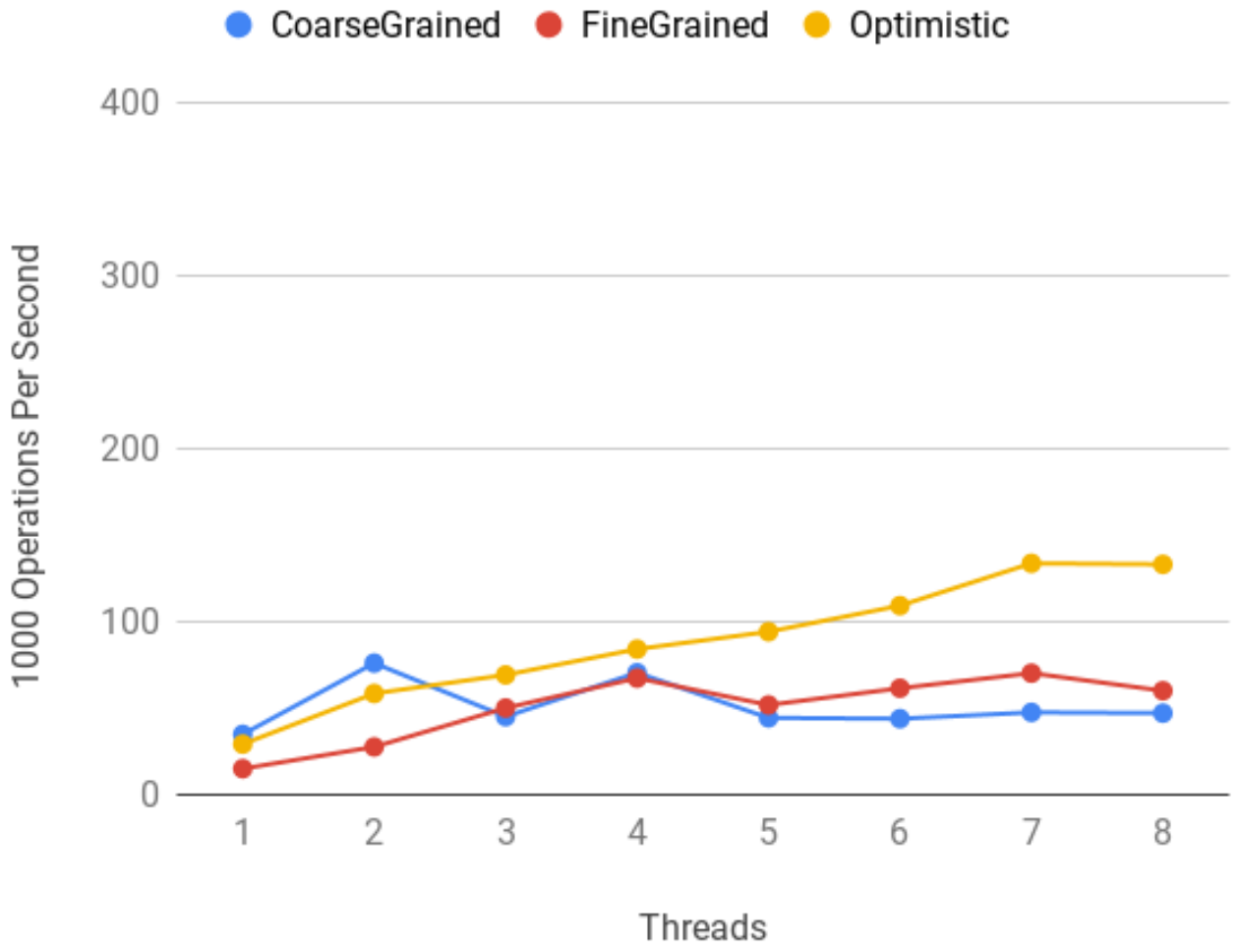
Lazy Synchronization - Contains

```
1    public boolean contains(T item) {
2      int key = item.hashCode();
3      Node curr = head;
4      while (curr.key < key)
5        curr = curr.next;
6      return curr.key == key && !curr.marked;
7    }
```

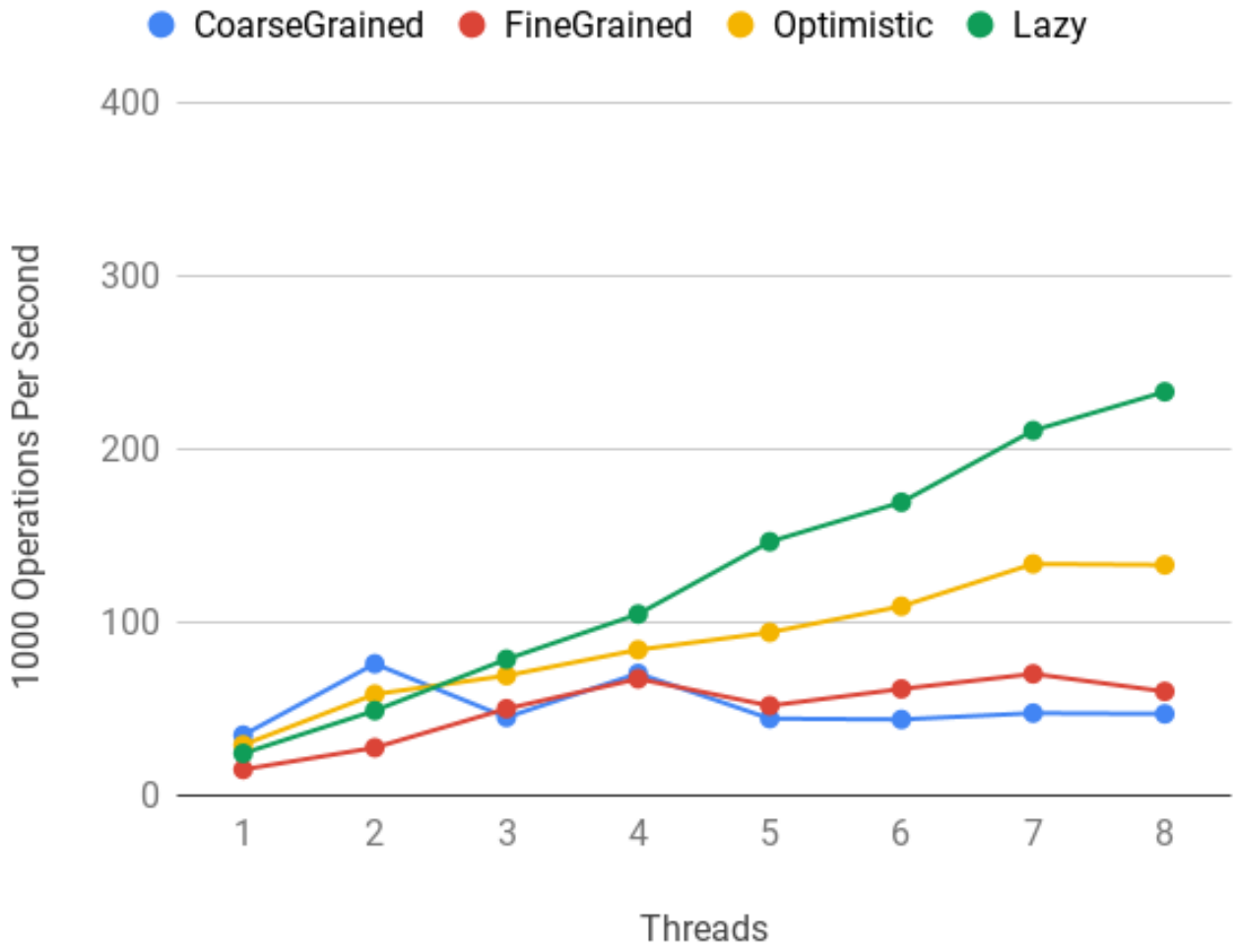
- When successful (i.e., when returns true), contains is linearized when an unmarked matching node is found
- When unsuccessful, linearization point is earliest of when:
  - Traversal reaches a marked node with matching key, or a node with key greater than one being searched
  - The point immediately before a new matching node is added

Linearization points not predetermined in method's code

Lazy Synchronization



Lazy Synchronization



Lazy Synchronization - Issues

- Contended add() and remove() calls must re-traverse
- Thread delays can affect other operations?

General Insight

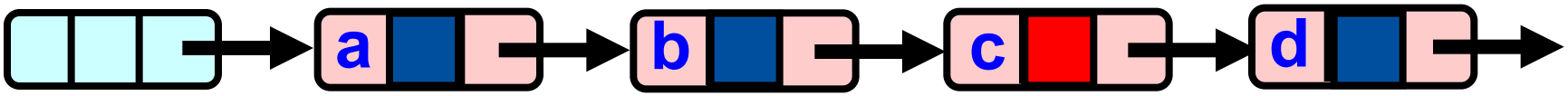
- Any concurrent data structure based on mutual exclusion has a weakness
- If a thread enters critical section, and then delays
  - Cache miss, page fault, scheduled out …
- Everyone else using that lock is stuck!

Non-Blocking Synchronization

- Lock-free Data Structures
  - Some thread will always complete a method call
    (guaranteed minimal progress in any execution)
    (even when other threads halt)
- Lock-free add() and remove() with wait-free contains()
- Atomics will be used instead of locks

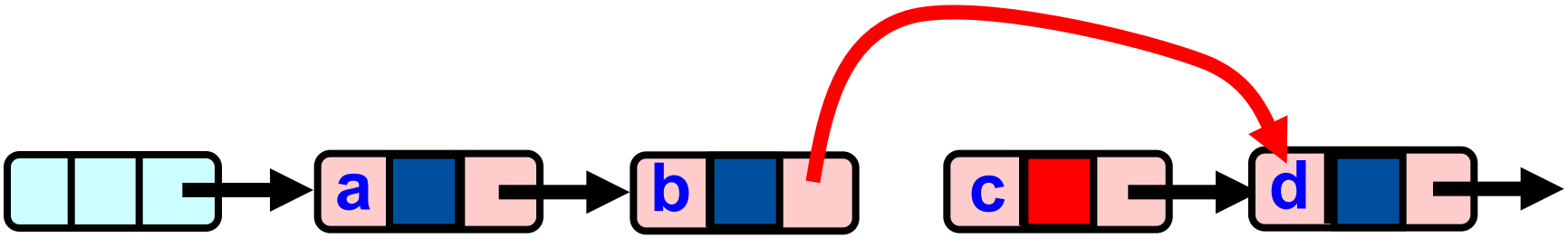
```
bool cas(pointer p, int old, int new) {
    if (*p ≠ old) { return false; }
    *p ← new;
    return true;
}
```

Non-Blocking Synchronization



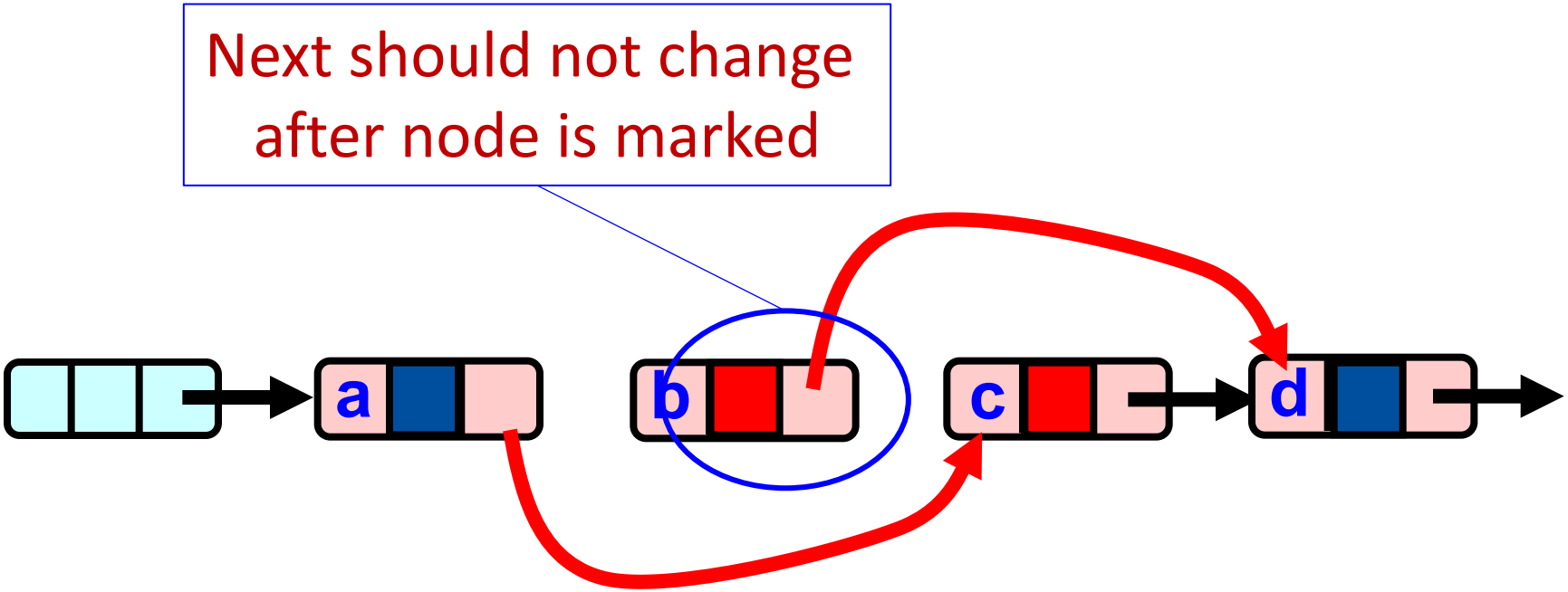
- Logical deletion – Marking the node

Non-Blocking Synchronization



- Logical deletion – Marking the node
- Physical deletion – Swing pointer using CAS

Problem



- Logical deletion – Marking the node
- Physical deletion – Swing pointer using CAS

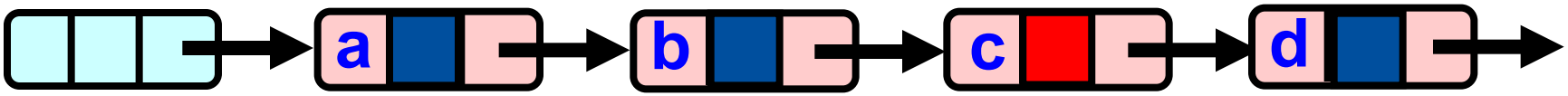
Clearly not enough!

Non-Blocking Synchronization

- AtomicMarkableReference
- Combine next pointer and mark into a single structure
  - Can atomically update both of them
  - Similar to stealing a bit from a pointer

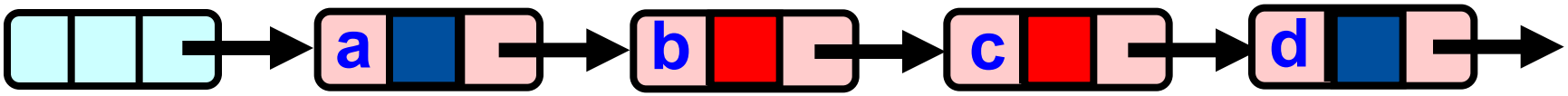
```
bool compareAndSet(T expectedReference,
                   T newReference,
                   bool expectedMark,
                   bool newMark);
bool attemptMark(T expectedReference,
                 bool newMark);
T get(bool[] marked);
```

With AtomicMarkableReference



- Concurrent deletion of b and c

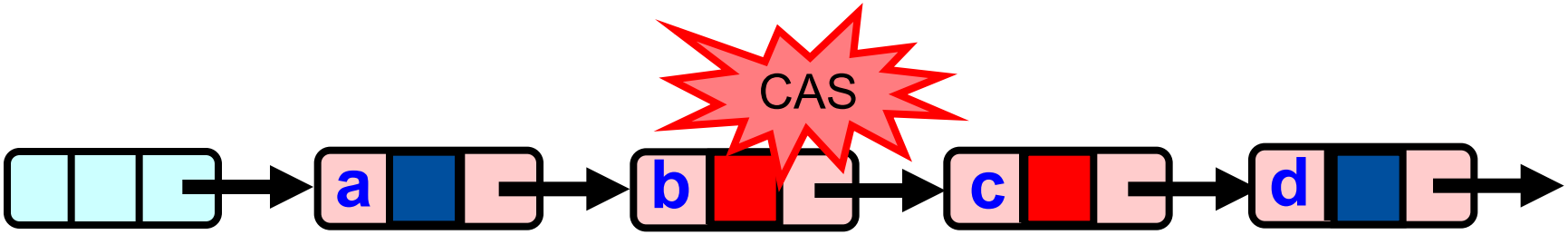
With AtomicMarkableReference



- Concurrent deletion of b and c

With AtomicMarkableReference

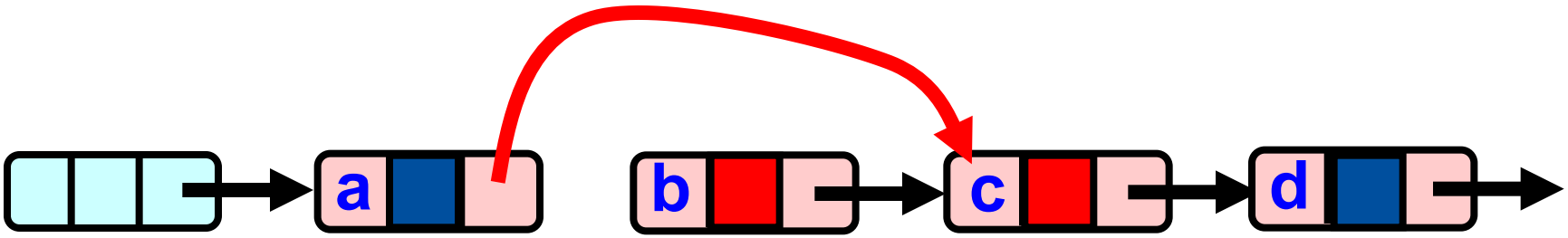
* Remember: CAS is on AtomicMarkable Reference



- Concurrent deletion of b and c
- CAS* on b.next fails

With AtomicMarkableReference

* Remember: CAS is on AtomicMarkable Reference



- Concurrent deletion of b and c
- CAS* on b.next fails
- CAS* on a.next succeeds

With AtomicMarkableReference

* Remember: CAS is on AtomicMarkable Reference

- Concurrent deletion of b and c
- CAS* on b.next fails
- CAS* on a.next succeeds
- What to do about the logically deleted node c?

Leave it!
Let other operations do the physical removal

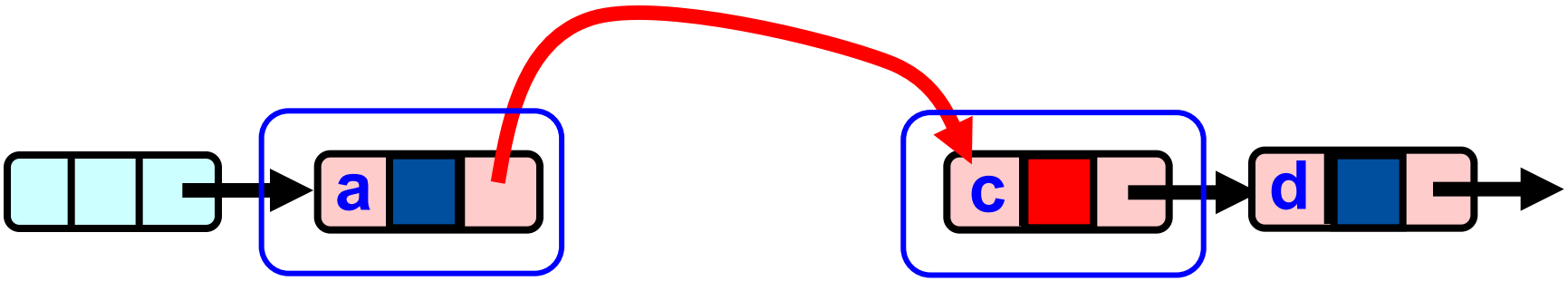
With AtomicMarkableReference

Traversal in add() or remove() do CAS for physical removal

With AtomicMarkableReference

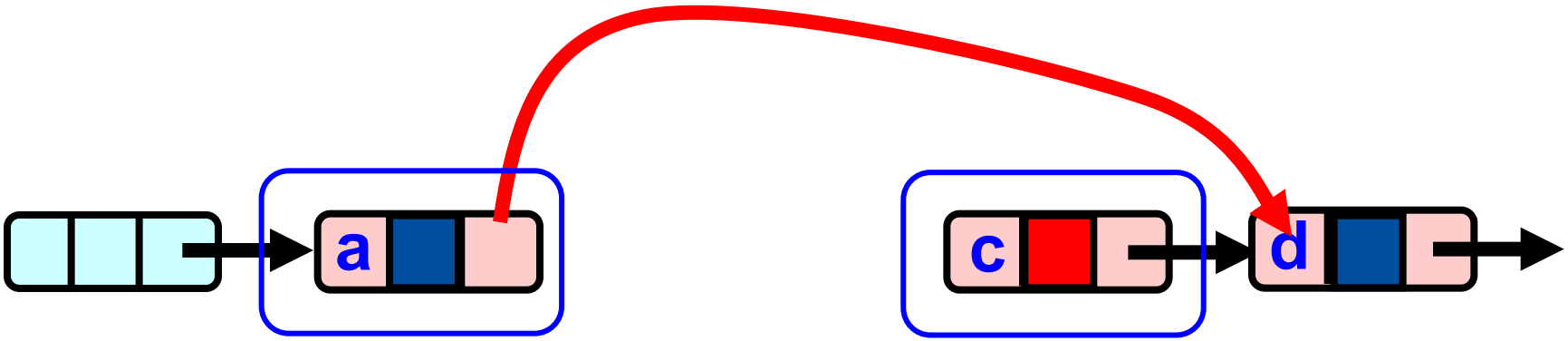
Traversal in add() or remove() do CAS for physical removal

With AtomicMarkableReference



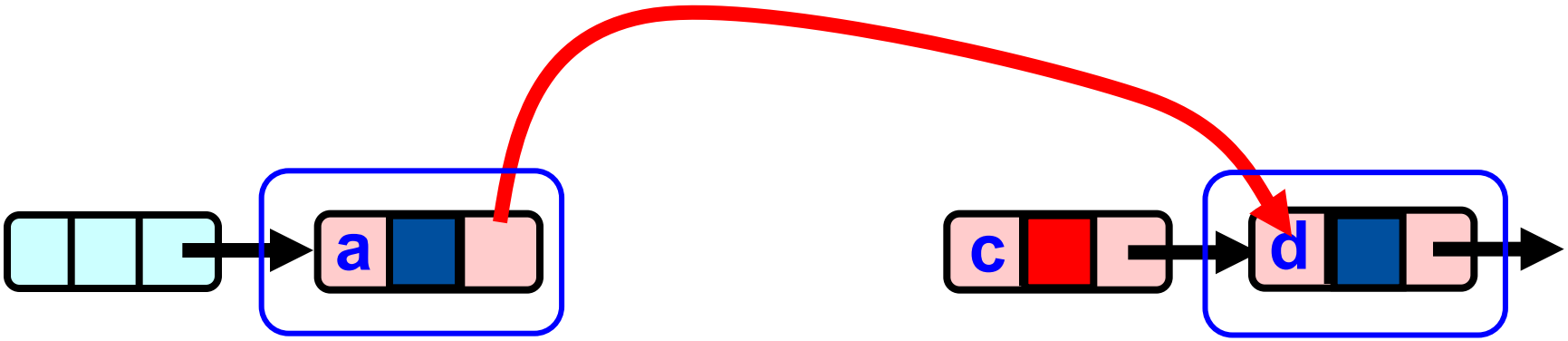
Traversal in add() or remove() do CAS for physical removal

With AtomicMarkableReference



Traversal in add() or remove() do CAS for physical removal

With AtomicMarkableReference



Traversal in add() or remove() do CAS for physical removal

- Lock-free
- find() does traversal + cleanup

```
1   class Window {
2     public Node pred, curr;
3     Window(Node myPred, Node myCurr) {
4       pred = myPred; curr = myCurr;
5     }
6   }
7   public Window find(Node head, int key) {
8     Node pred = null, curr = null, succ = null;
9     boolean[] marked = {false};
10    boolean snip;
11    retry: while (true) {
12      pred = head;
13      curr = pred.next.getReference();
14      while (true) {
15        succ = curr.next.get(marked);
16        while (marked[0]) {
17          snip = pred.next.compareAndSet(curr, succ, false, false);
18          if (!snip) continue retry;
19          curr = succ;
20          succ = curr.next.get(marked);
21        }
22        if (curr.key >= key)
23          return new Window(pred, curr);
24        pred = curr;
25        curr = succ;
26      }
27    }
28  }
```

```
1  public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4      Window window = find(head, key);
5      Node pred = window.pred, curr = window.curr;
6      if (curr.key == key) {
7          return false;
8        } else {
9          Node node = new Node(item);
10         node.next = new AtomicMarkableReference(curr, false);
11         if (pred.next.compareAndSet(curr, node, false, false)) {
12           return true;
13         }
14       }
15     }
16   }
```

wait-free?

non-blocking?

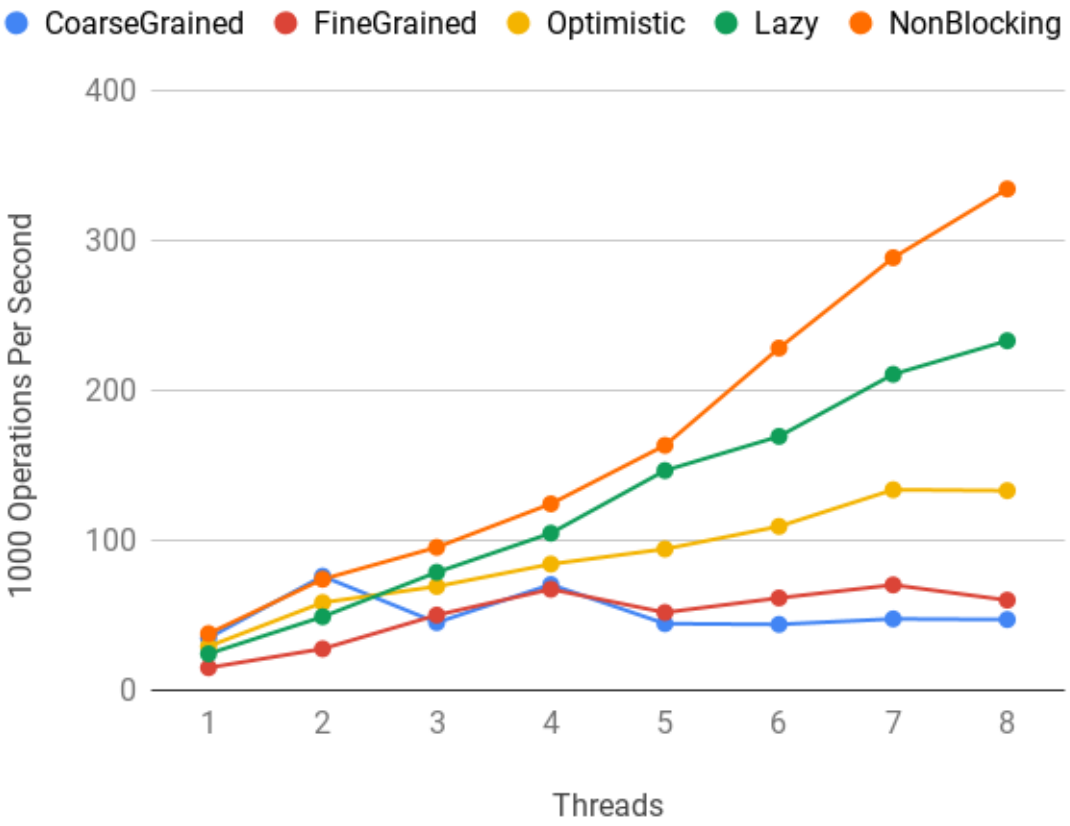
```
17    public boolean remove(T item) {
18      int key = item.hashCode();
19      boolean snip;
20      while (true) {
21       Window window = find(head, key);
22       Node pred = window.pred, curr = window.curr;
23       if (curr.key != key) {
24         return false;
25       } else {
26         Node succ = curr.next.getReference();
27         snip = curr.next.attemptMark(succ, true);
28         if (!snip)
29           continue;
30         pred.next.compareAndSet(curr, succ, false, false);
31         return true;
32       }
33     }
34   }
```

Non-Blocking Synchronization

```
35    public boolean contains(T item) {
36      boolean[] marked = false{};
37      int key = item.hashCode();
38      Node curr = head;
39      while (curr.key < key) {
40       curr = curr.next;
41       Node succ = curr.next.get(marked);
42      }
43      return (curr.key == key && !marked[0])
44    }
```

contains is wait-free

Non-Blocking Synchronization



To Lock or Not to Lock

- Locking v/s Non-blocking
- Beneficial to compromise
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
- Blocking/non-blocking is a property of a method