IT308: Operating Systems

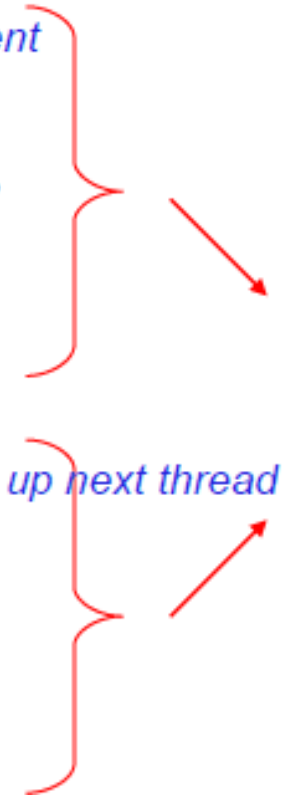
Fine-grained locking

Recall: Semaphores

- P() or wait() or down()
  - From Dutch "proeberen", meaning "test"
  - Atomic action:
    - Wait for semaphore value to become > 0, then decrement it
- V() or signal() or up()
  - From Dutch "verhogen", meaning "increment"
  - Atomic action:
    - Increments semaphore value by 1.

Simple Semaphore Implementation

```
struct semaphore {
    int val;
    thread_list waiting;  // List of threads waiting for semaphore
}

P(semaphore Sem):    // Wait until > 0 then decrement
    while (Sem.val <= 0) {
        add this thread to Sem.waiting;
        block(this thread);  // What does this do??
    }
    Sem.val = Sem.val -1;
    return;

V(semaphore Sem):    // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
        remove a thread T from Sem.waiting;
        wakeup(T);
    }
```



P() and V() must be atomic actions!

Simple Semaphore Implementation

```
struct semaphore {
    int val;
    thread_list waiting;  // List of threads waiting for semaphore
}

P(semaphore Sem):   // Wait until > 0 then decrement
    while (Sem.val <= 0) {
        add this thread to Sem.waiting;
        block(this thread);  // What does this do??
    }
    Sem.val = Sem.val -1;
    return;

V(semaphore Sem):   // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
        remove a thread T from Sem.waiting;
        wakeup(T);
    }
```

Why is this a while loop and not just an if statement?

Reader/Writers

- Single shared object
- Want to allow any number of threads to read simultaneously
- But, only one thread should be able to write to the object at a time
  - And, not interfere with any readers . . .

Readers/Writers

- readers share:
  - semaphore wrt; // initialized to 1
  - int readcount; // initialized to 0
- writers also share semaphore wrt

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

Reader:

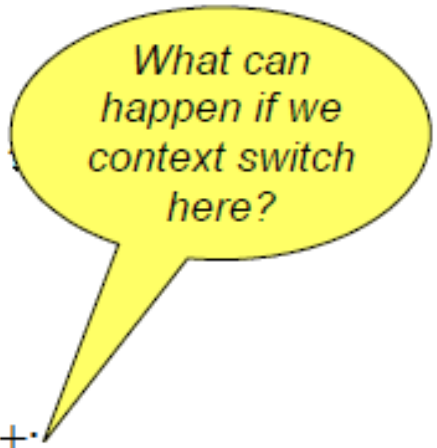
```
while (1) {
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
}
```

- Seems simple, but this code is broken . . .

Readers/Writers

Writer:

```
while (1) {
    P(wrt);
     // writing
    V(wrt);
}
```



Reader:

```
while (1) {
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    // reading
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
}
```

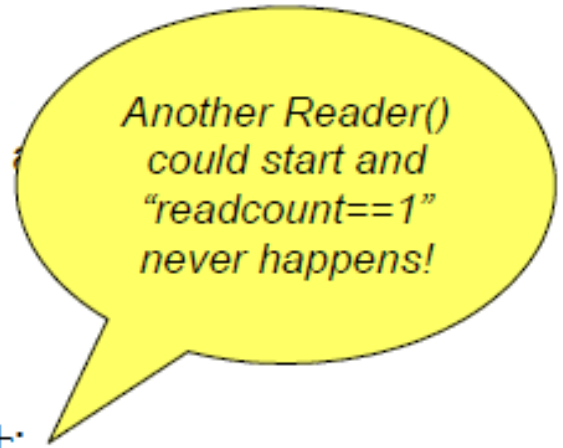
Readers/Writers

Writer:

```
while (1) {
    P(wrt);
     // writing
    V(wrt);
}
```

Reader:

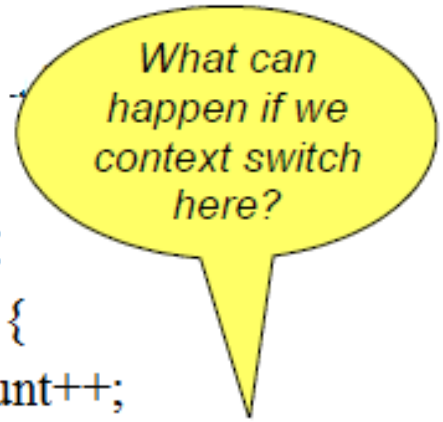
```
while (1) {
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    // reading
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
}
```



Readers/Writers

Writer:

```
while (1) {
    P(wrt);
     // writing
    V(wrt);
}
```



Reader:

```
while (1) {
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    // reading
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
}
```

Readers/Writers

A Writer() could start, P the semaphore first, then subsequent Reader() threads would be able to get past the semaphore (since "readcount != 1")

Writer:

```
while (1) {
    P(wrt);
     // writing
    V(wrt);
}
```

Reader:

```
while (1) {
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    // reading
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
}
```

Readers/Writers fixed

- Problem: Multiple Readers are accessing `readcount`
  - Solution: Make "increment, test, P" and "decrement, test, V" both atomic - using a mutex

Writer:

```
while (1) {
   P(wrt);
   // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   P(mutex);
   readcount++;
   if (readcount == 1) {
      P(wrt);
   }
   V(mutex);
   // reading
   P(mutex);
   readcount--;
   if (readcount == 0) {
      V(wrt);
   }
   V(mutex);
}
```

Readers/Writers fixed

Writer:

```
while (1) {
    P(wrt);
    // writing
    V(wrt);
}
```

Reader:

```
while (1) {
    P(mutex);
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    V(mutex);
    // reading
    P(mutex);
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
    V(mutex);
}
```

What if a Writer() is active, the first Reader() stalls on P(wrt), and additional Readers() try to enter?

Readers/Writers fixed

Writer:

```
while (1) {
    P(wrt);
    // writing
    V(wrt);
}
```

Reader:

```
while (1) {
    P(mutex);
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    V(mutex);
    // reading
    P(mutex);
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
    V(mutex);
}
```

The subsequent Reader() threads stall on P(mutex) !

A sorted linked list

```
struct Node {
   int value;
   Node* next;
};
```

```
struct List {
  Node* head;
};
```

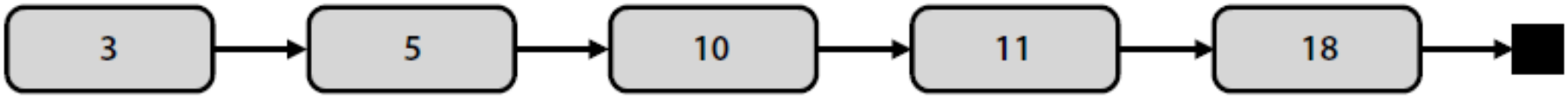
What can go wrong if multiple threads operate on the linked list simultaneously?

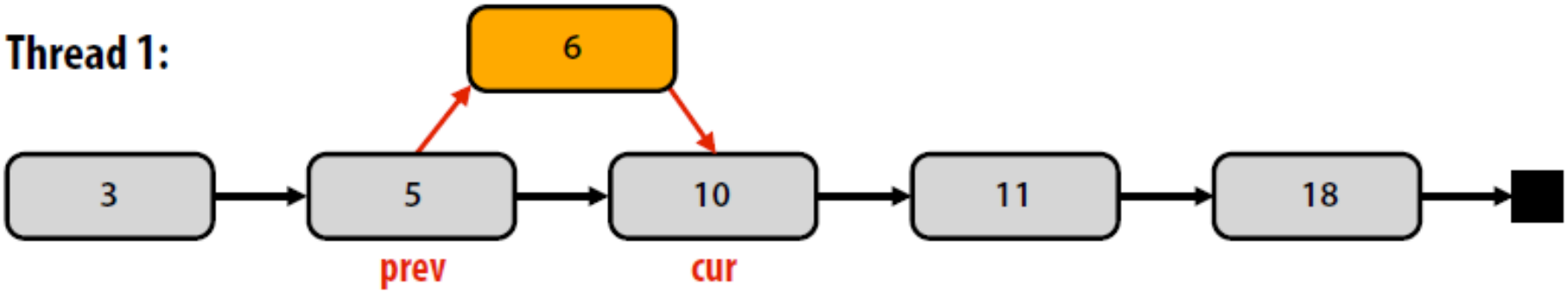
```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }

   n->next = cur;
   prev->next = n;
}
```

```
void delete(List* list, int value) {

   // assume case of deleting first element is
   // handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       return;
     }

     prev = cur;
     cur = cur->next;
   }
}
```

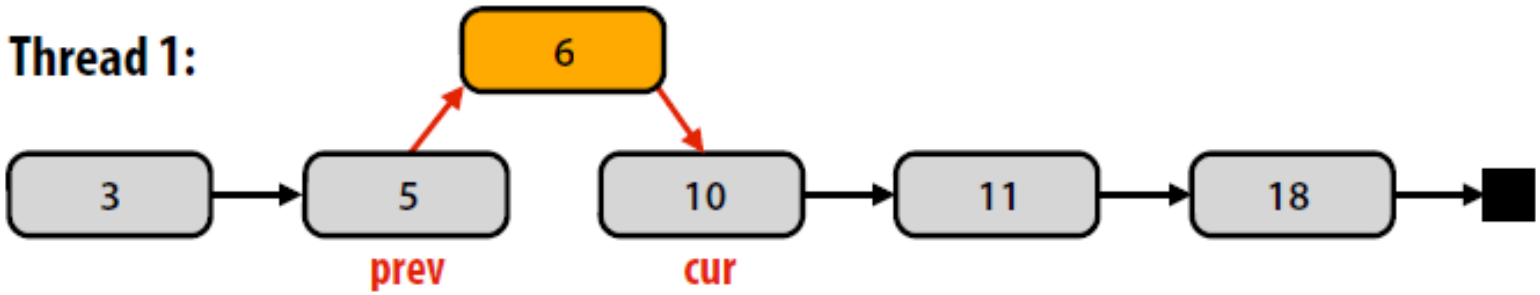
Example: concurrent insertion

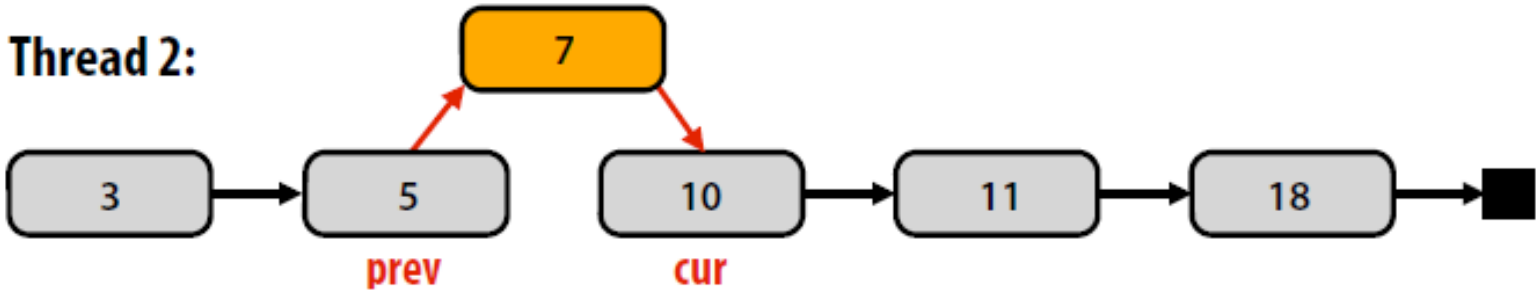
- Thread 1 attempts to insert 6
- Thread 2 attempts to insert 7





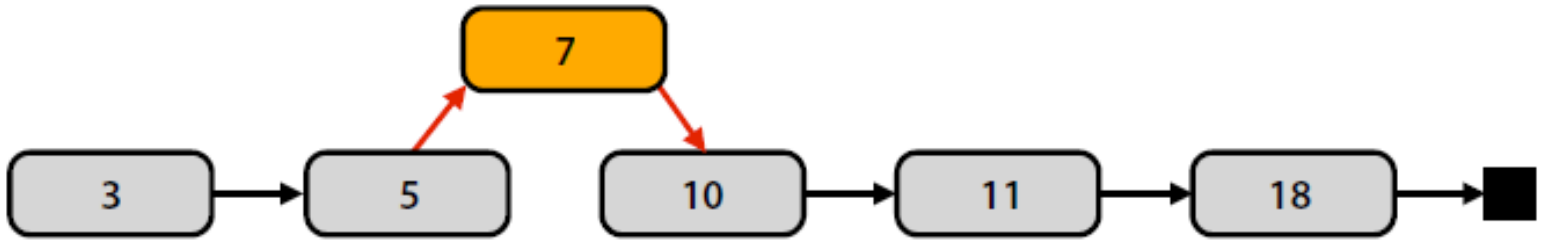
Example: concurrent insertion





Thread 1 and thread 2 both compute same prev and cur.
Result: one of the insertions gets lost!

Result: (assuming thread 1 updates `prev->next` before thread 2)



Solution 1: global locking

```
struct Node {
   int value;
   Node* next;
};
```

```
struct List {
  Node* head;
  Lock  lock;
};
```

Per-list lock

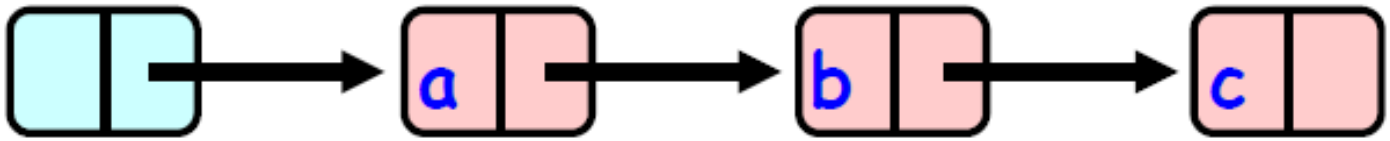
```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   lock(list->lock);

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }
   n->next = cur;
   prev->next = n;
   unlock(list->lock);
}
```

```
void delete(List* list, int value) {

   lock(list->lock);

   // assume case of deleting first element is
   // handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       unlock(list->lock);
       return;
     }

     prev = cur;
     cur = cur->next;
   }
   unlock(list->lock);
}
```

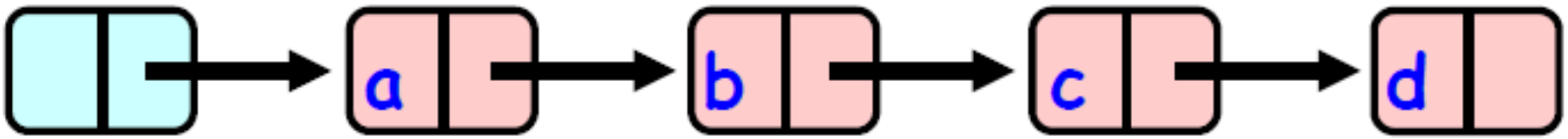
Single global lock

- It is relatively simple to implement correct mutual exclusion for data structure operations (we just did it!)
- Disadvantages:
  - Operations on the data structure are serialized
  - May limit application performance

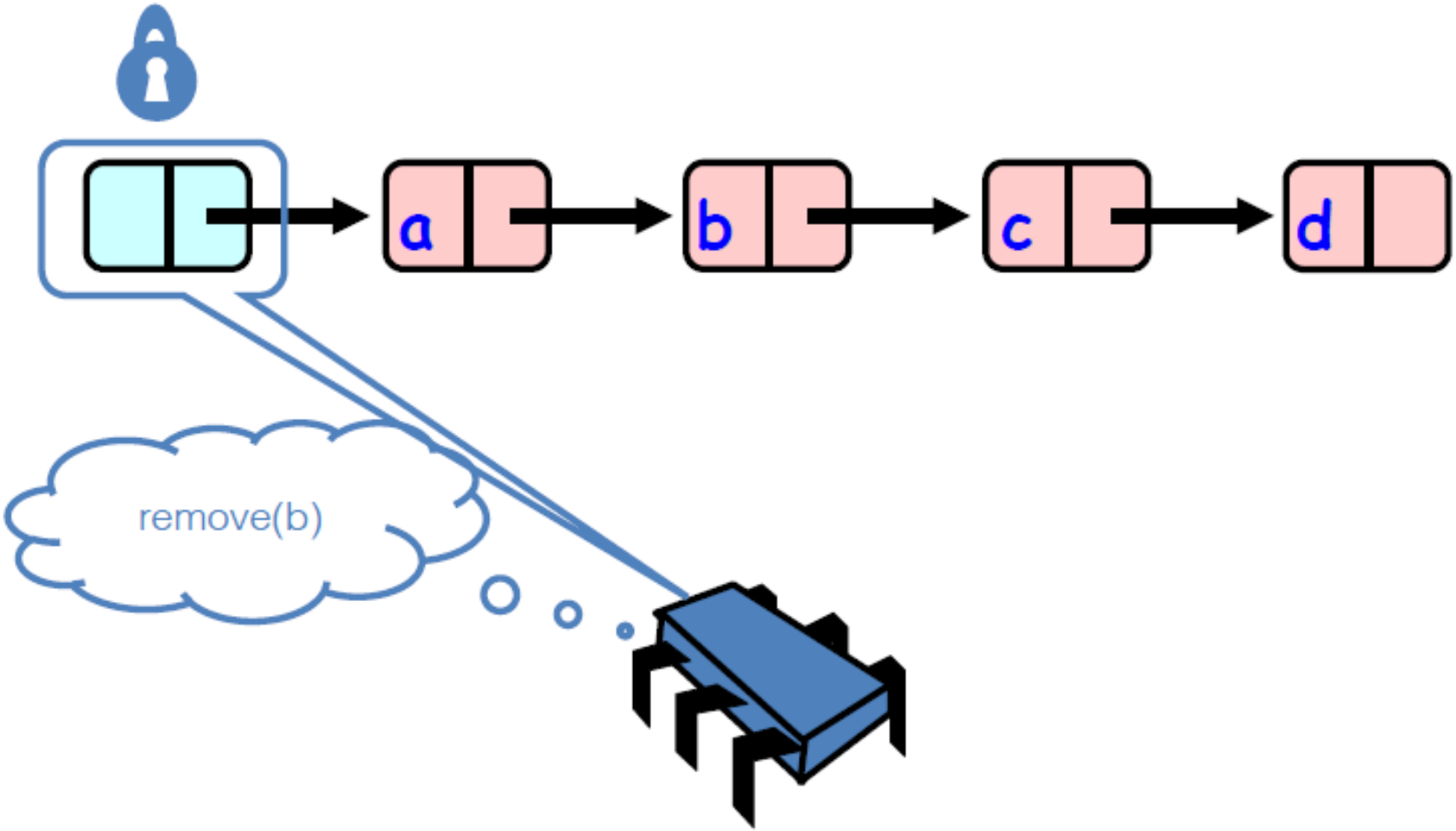
Hand-over-hand locking



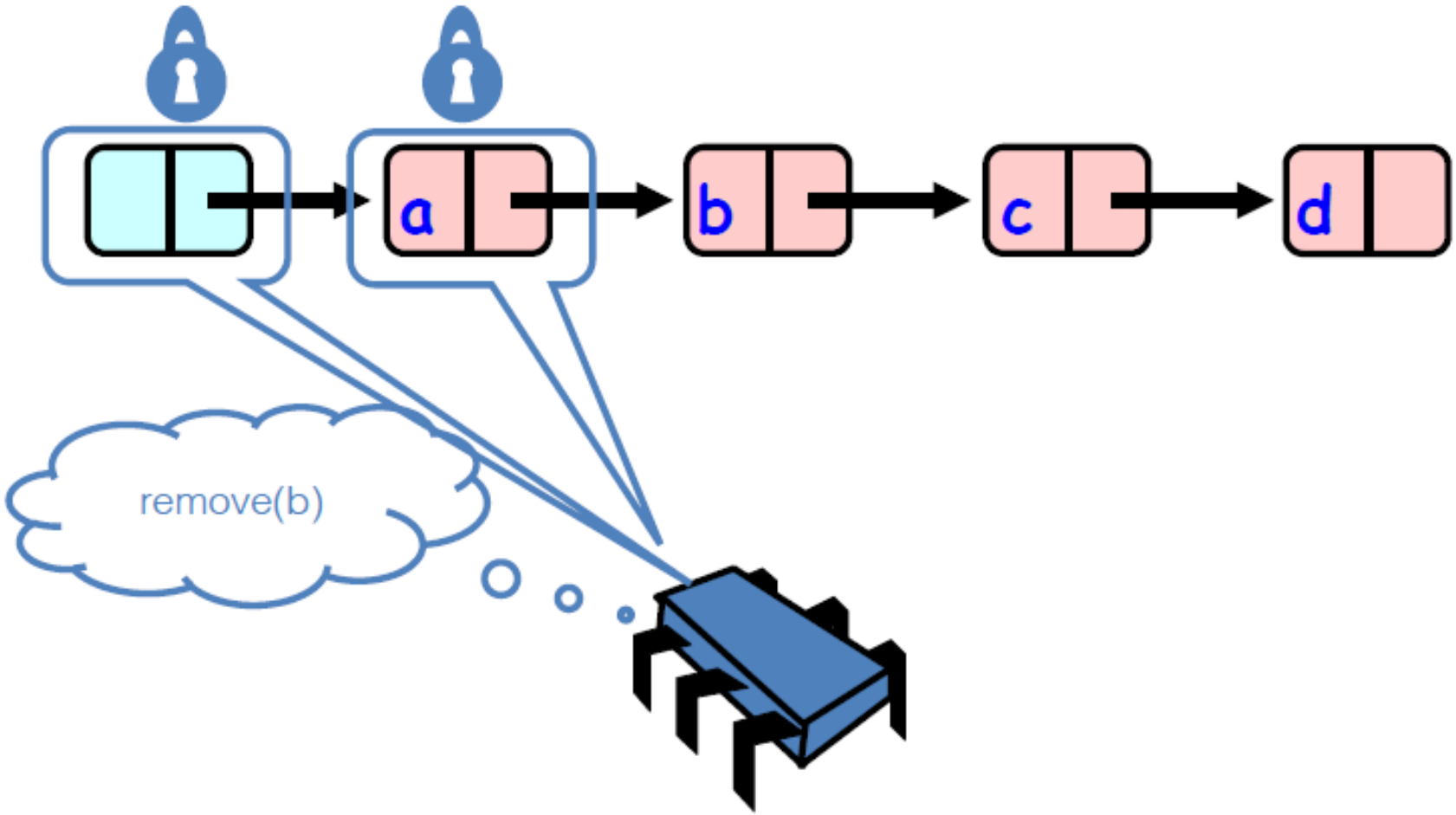
Removing a node



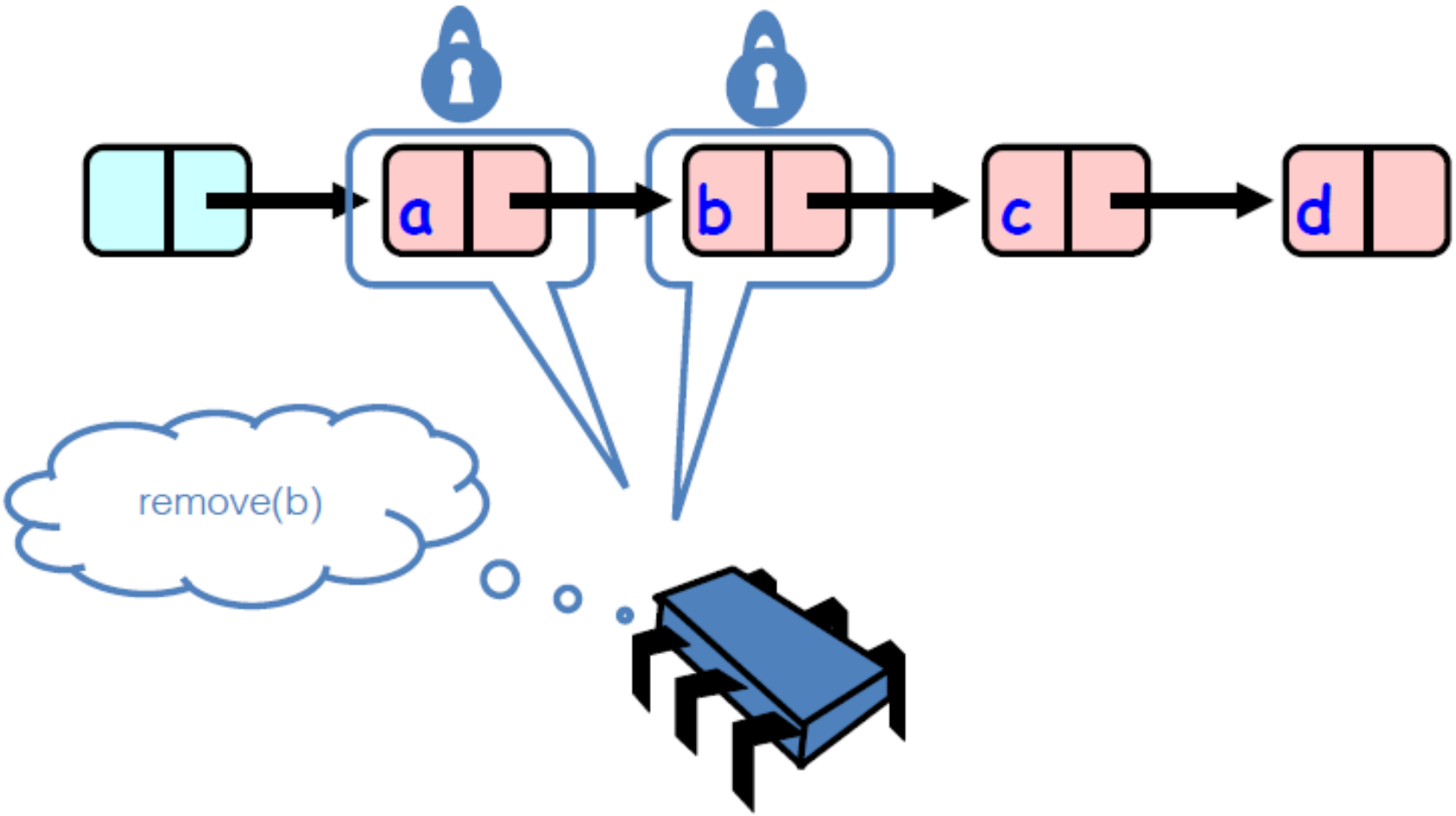
Removing a node



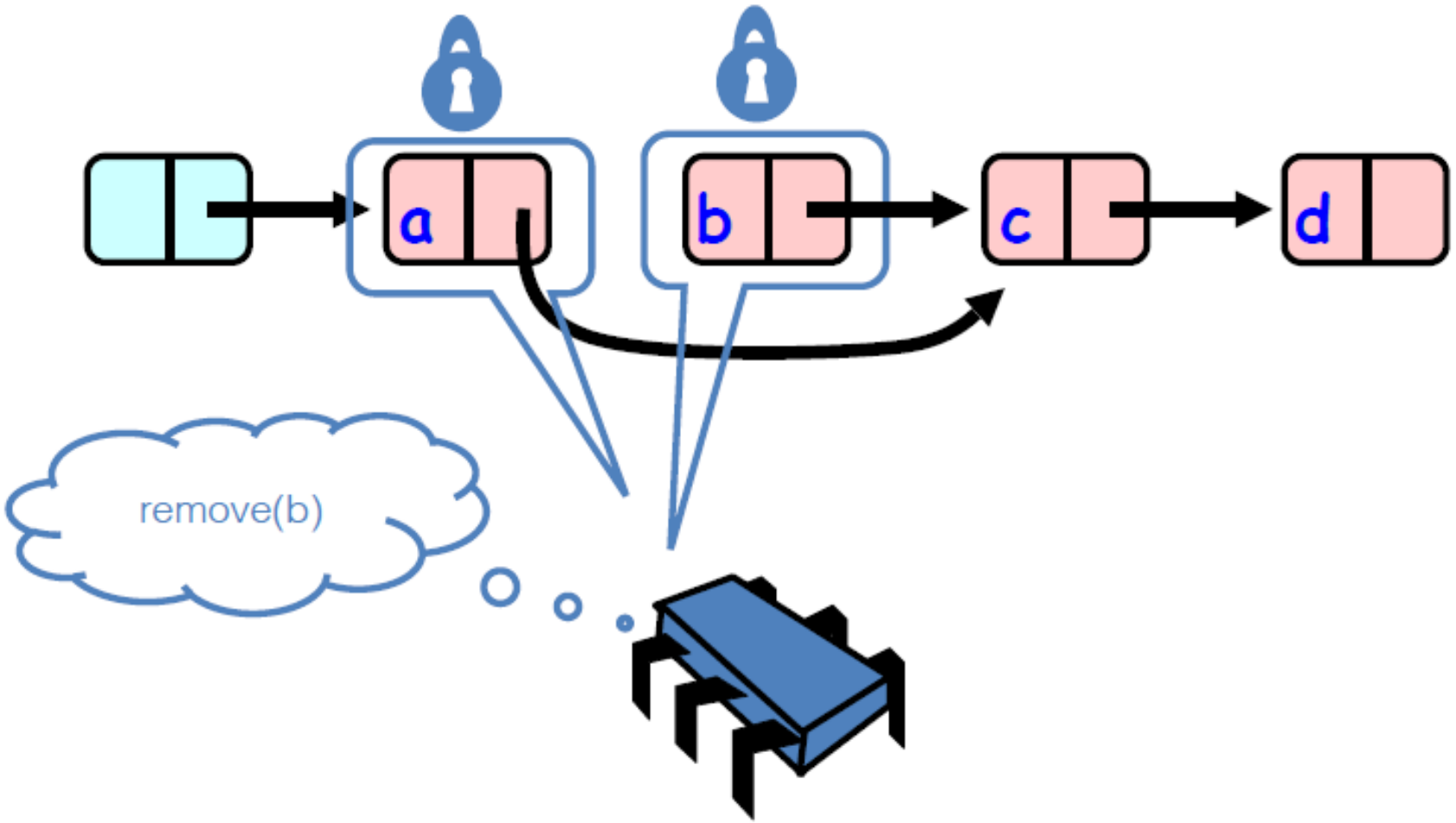
Removing a node



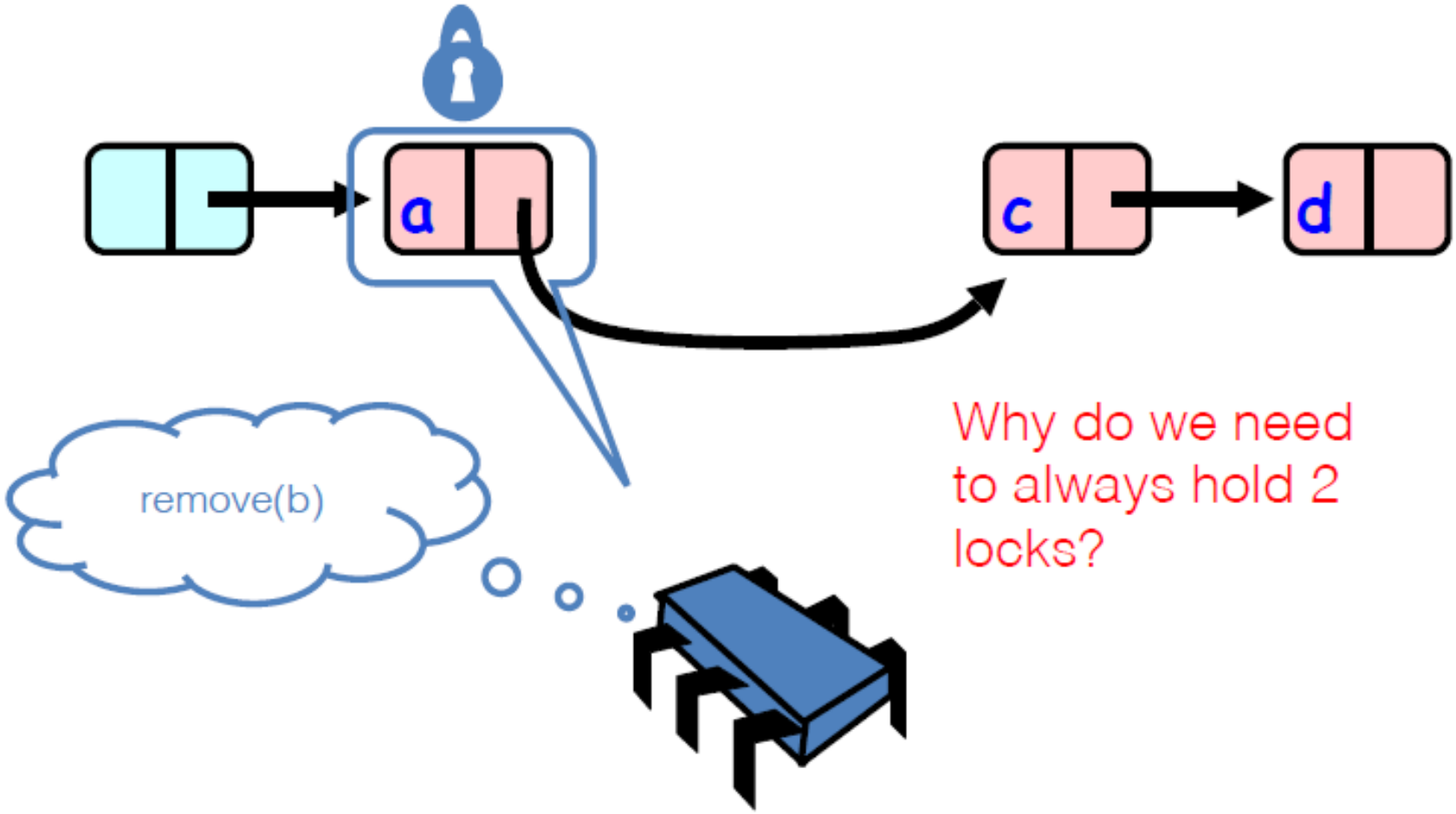
Removing a node



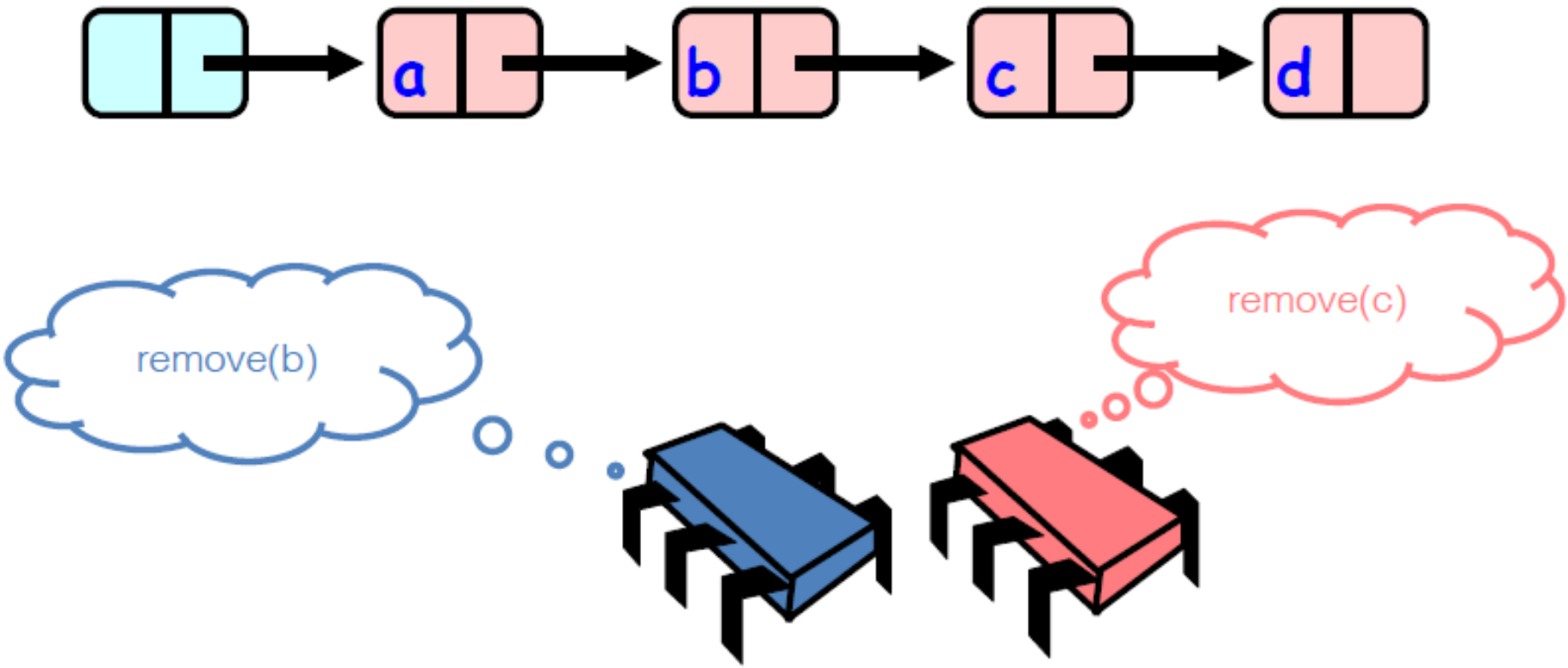
Removing a node



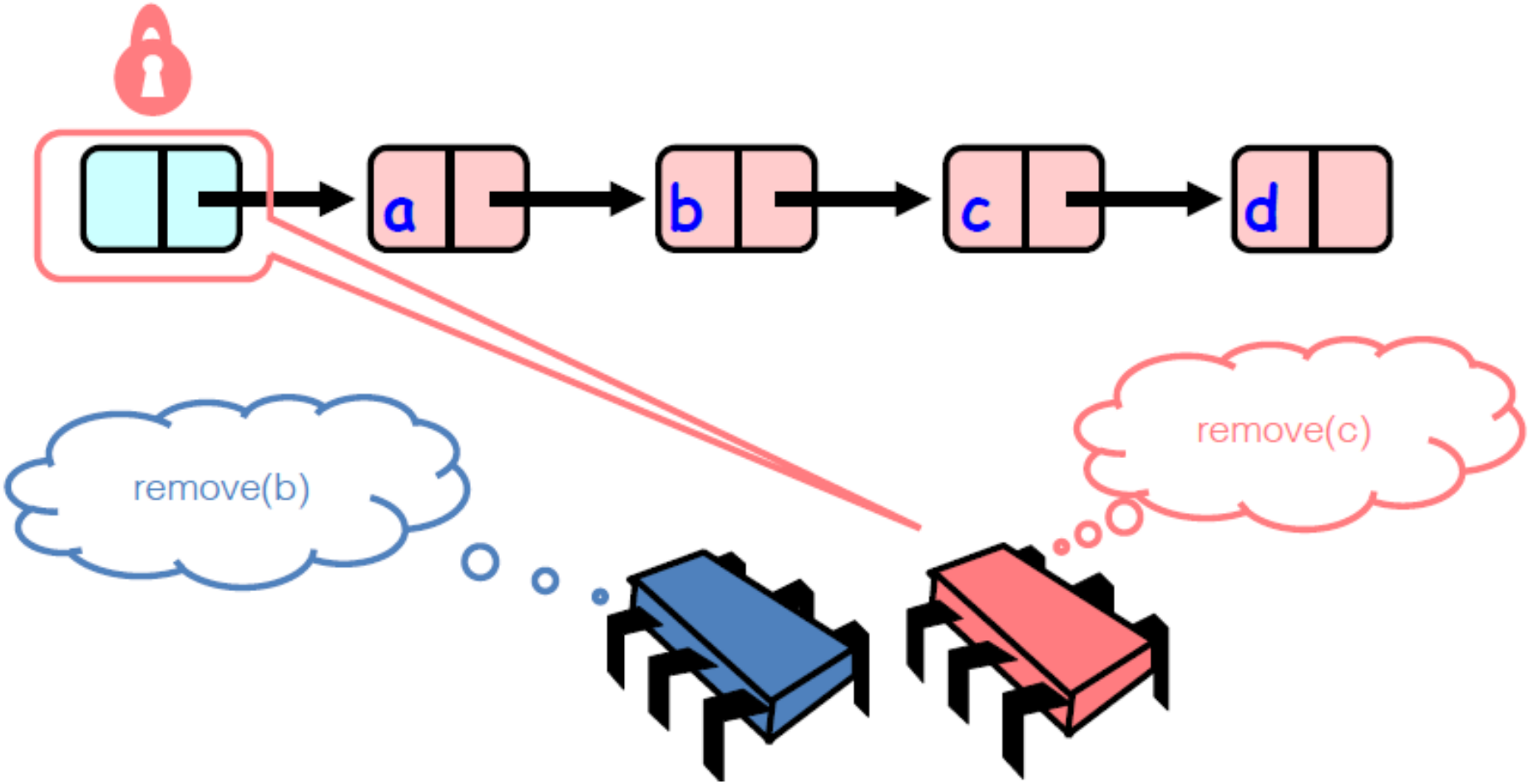
Removing a node



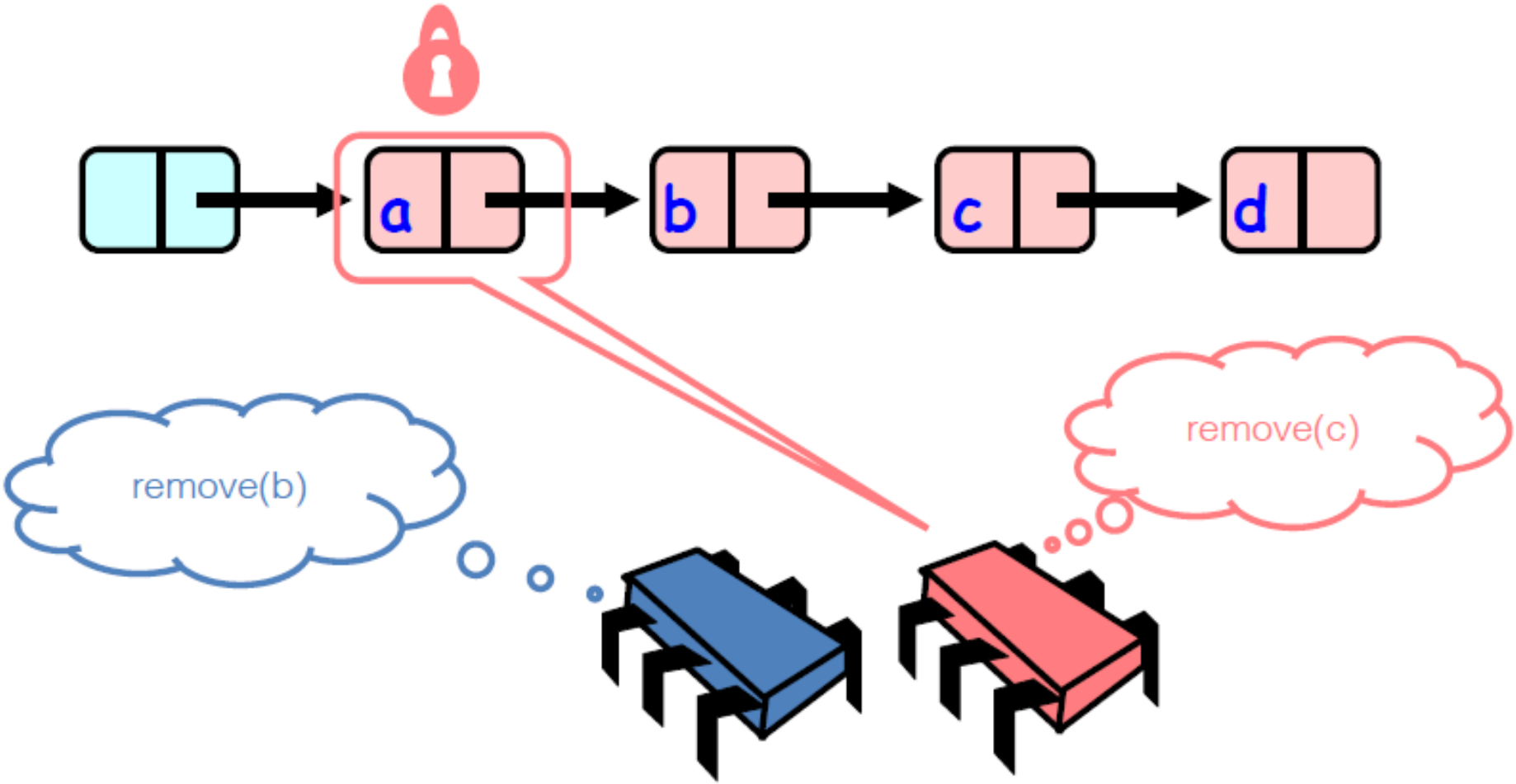
Concurrent Removes



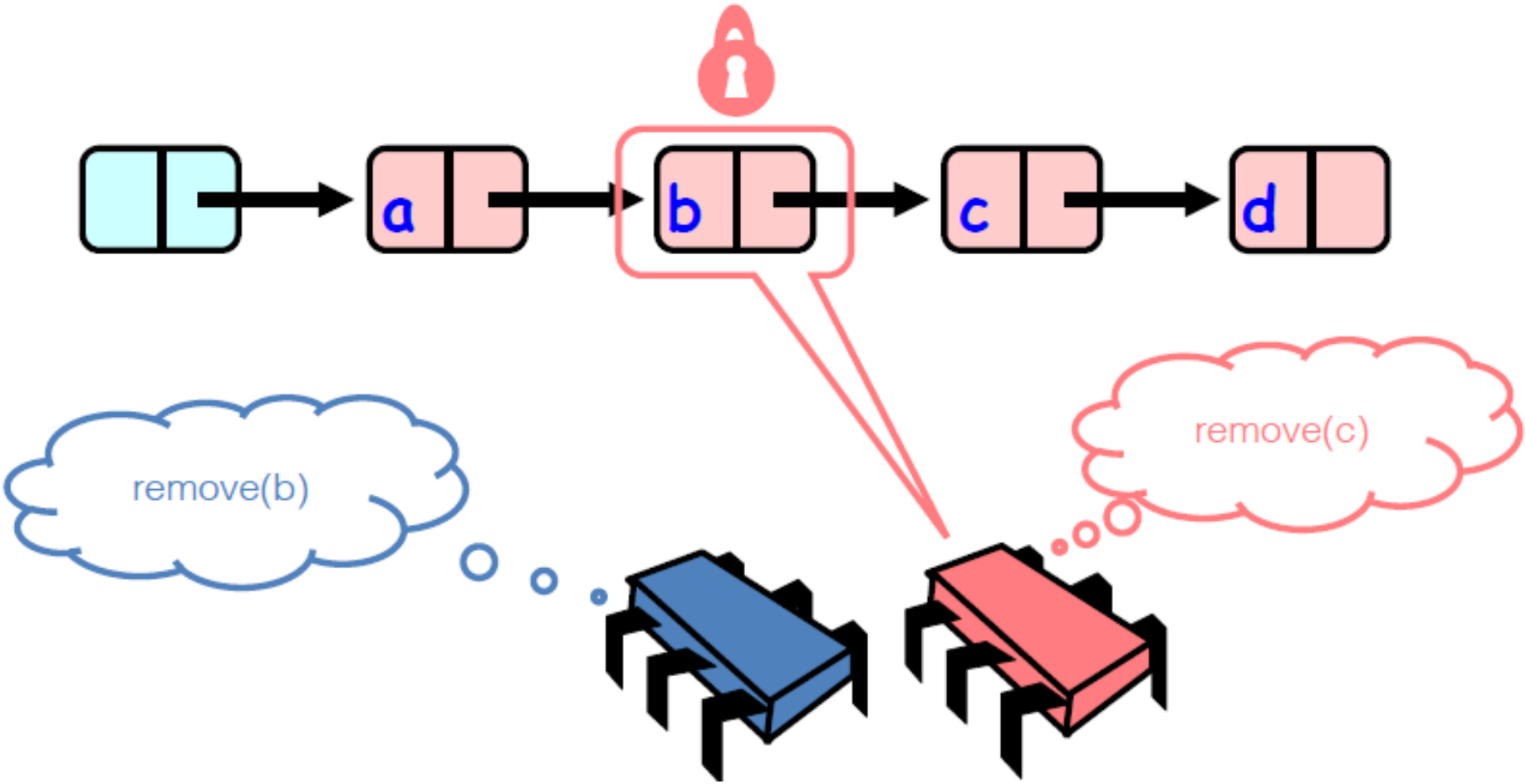
Concurrent Removes



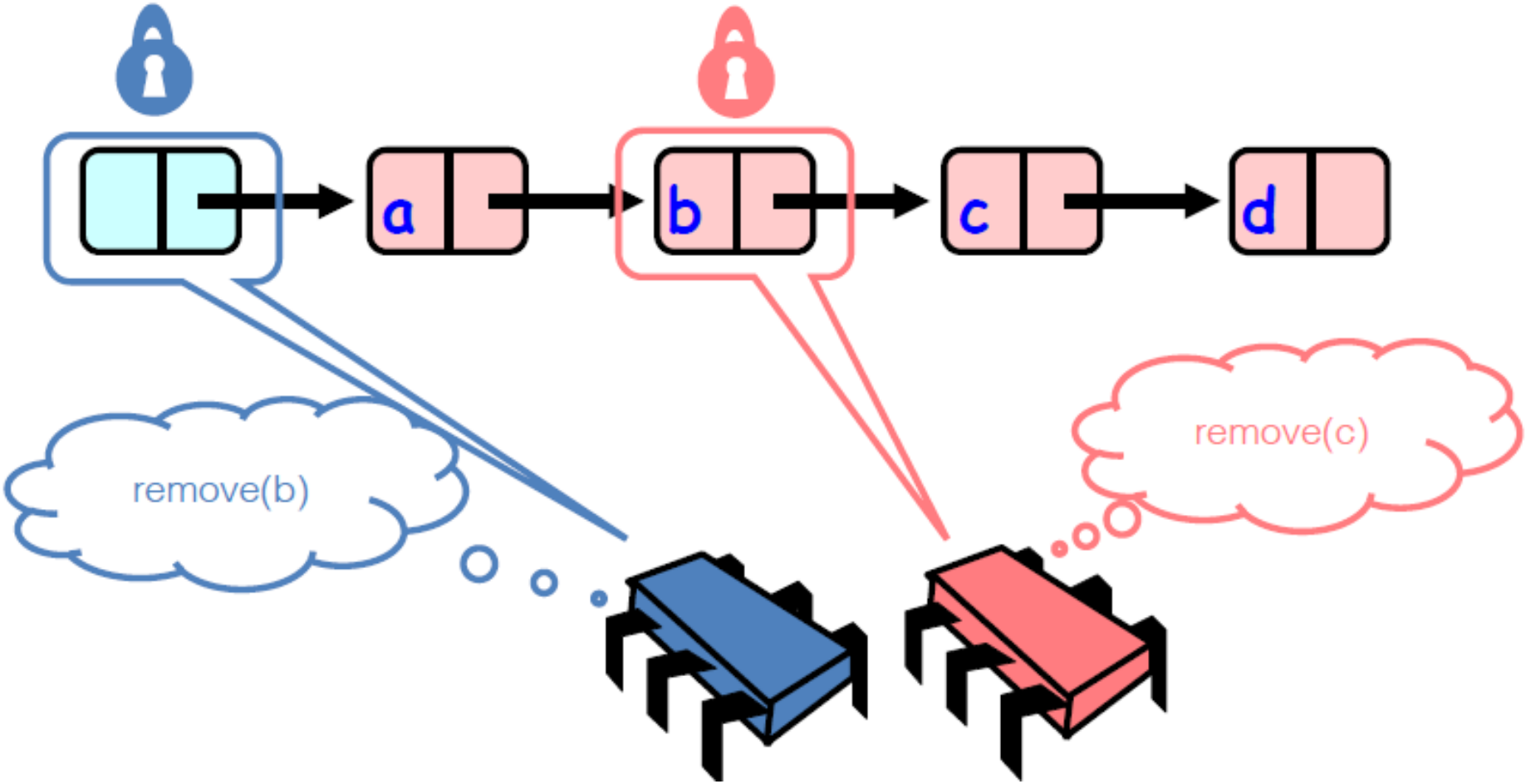
Concurrent Removes



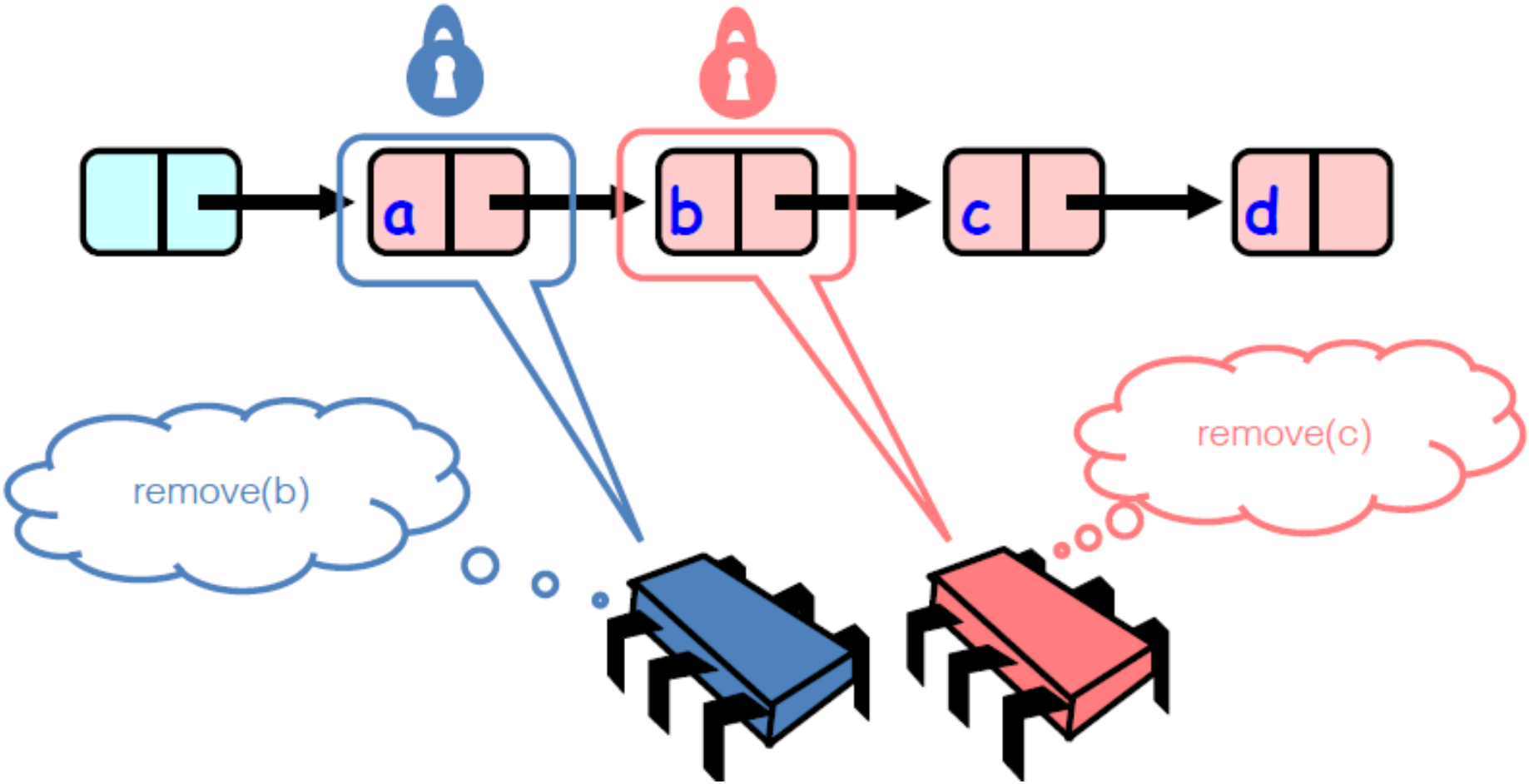
Concurrent Removes



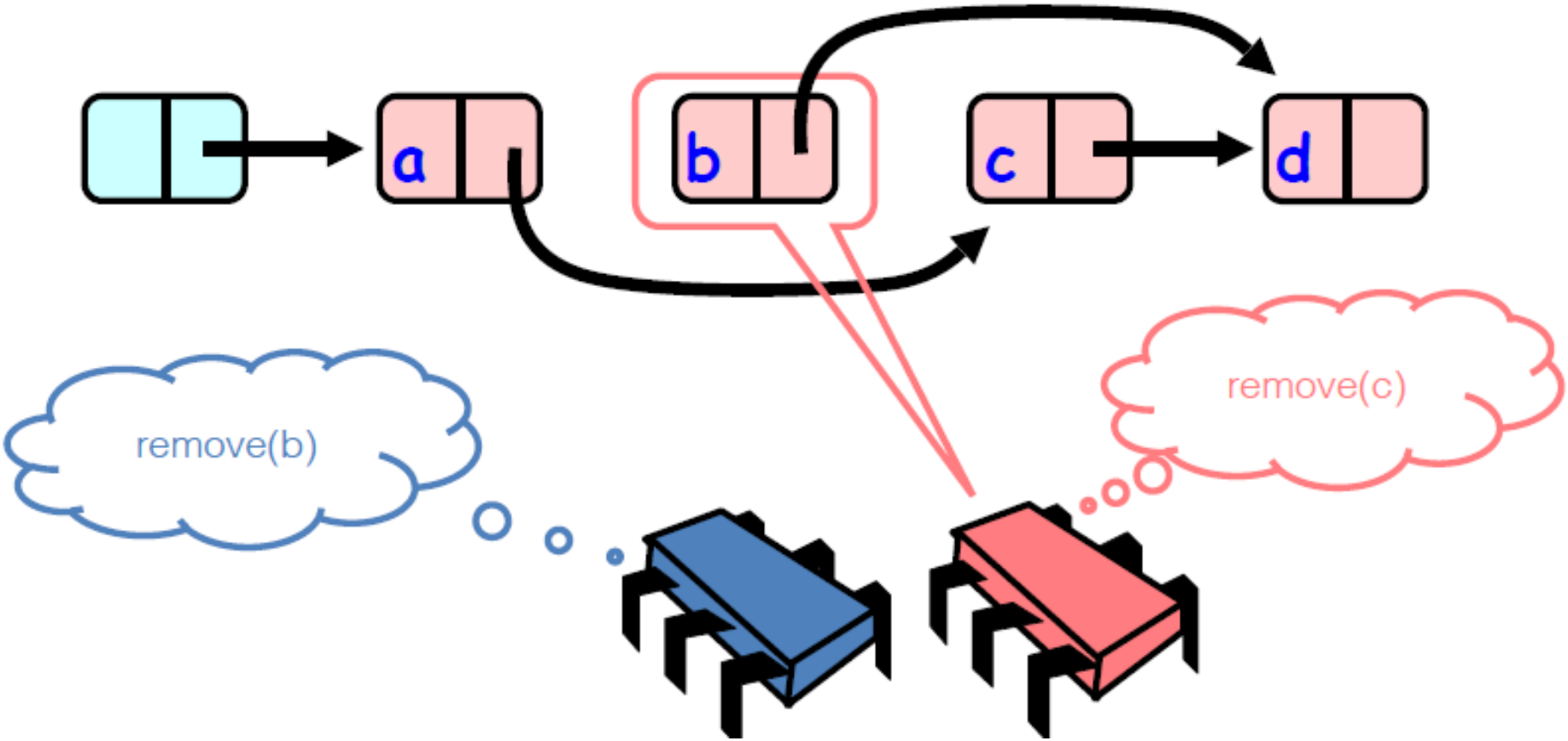
Concurrent Removes



Concurrent Removes

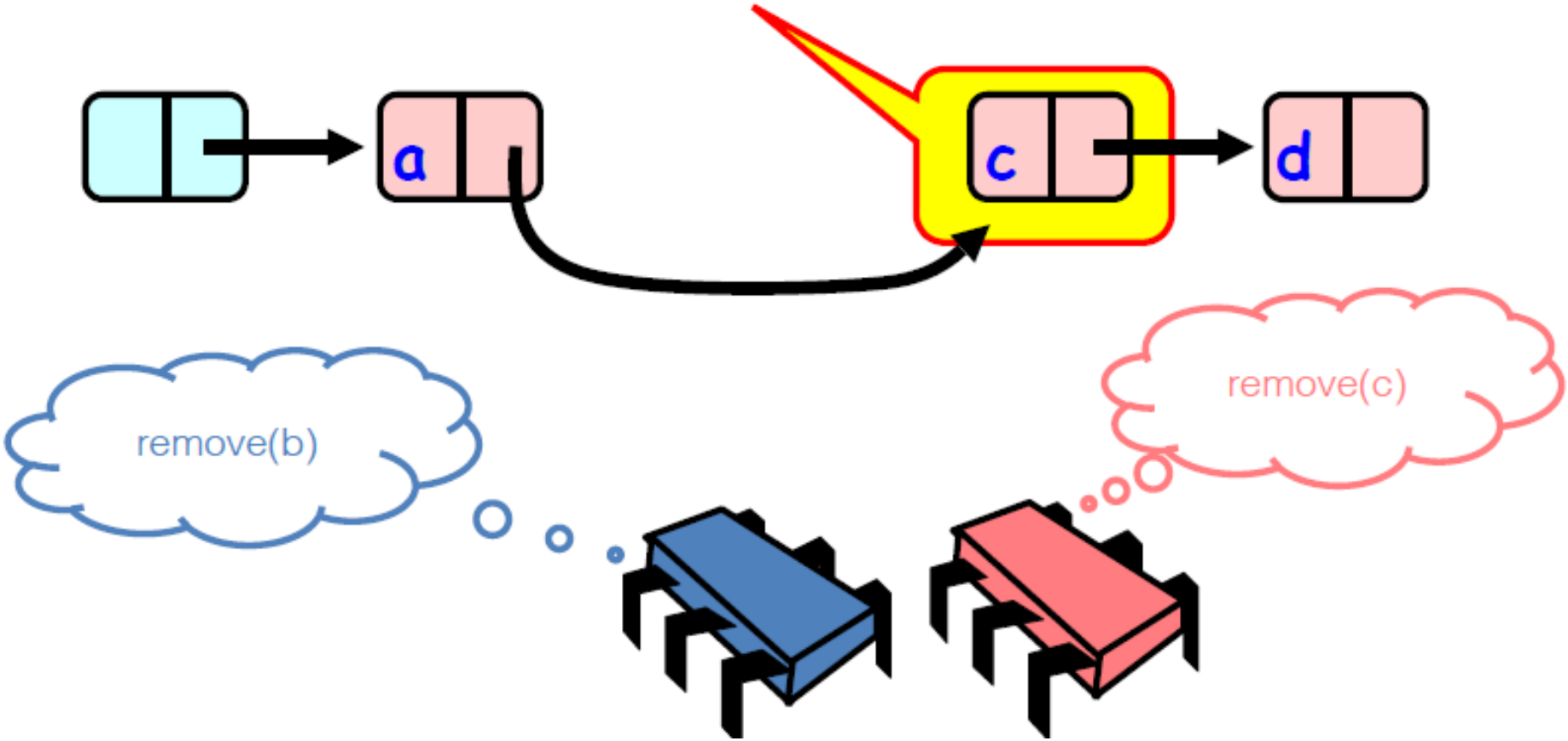


Concurrent Removes

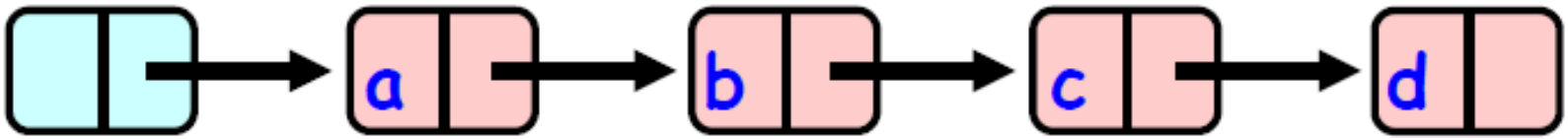


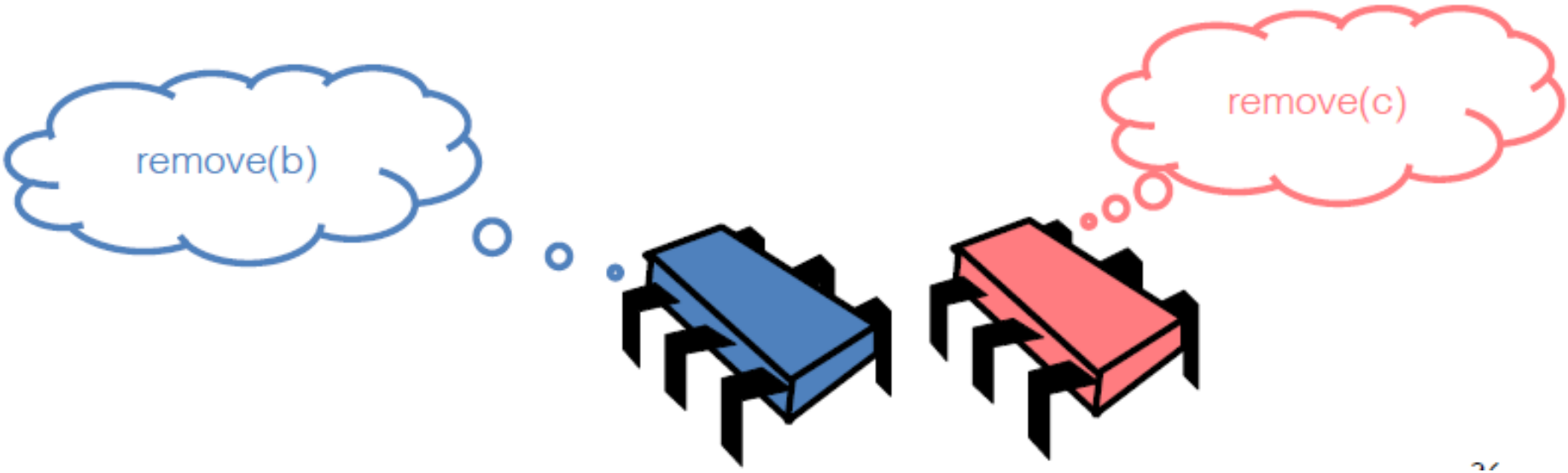
Concurrent Removes

Bad news, c not removed

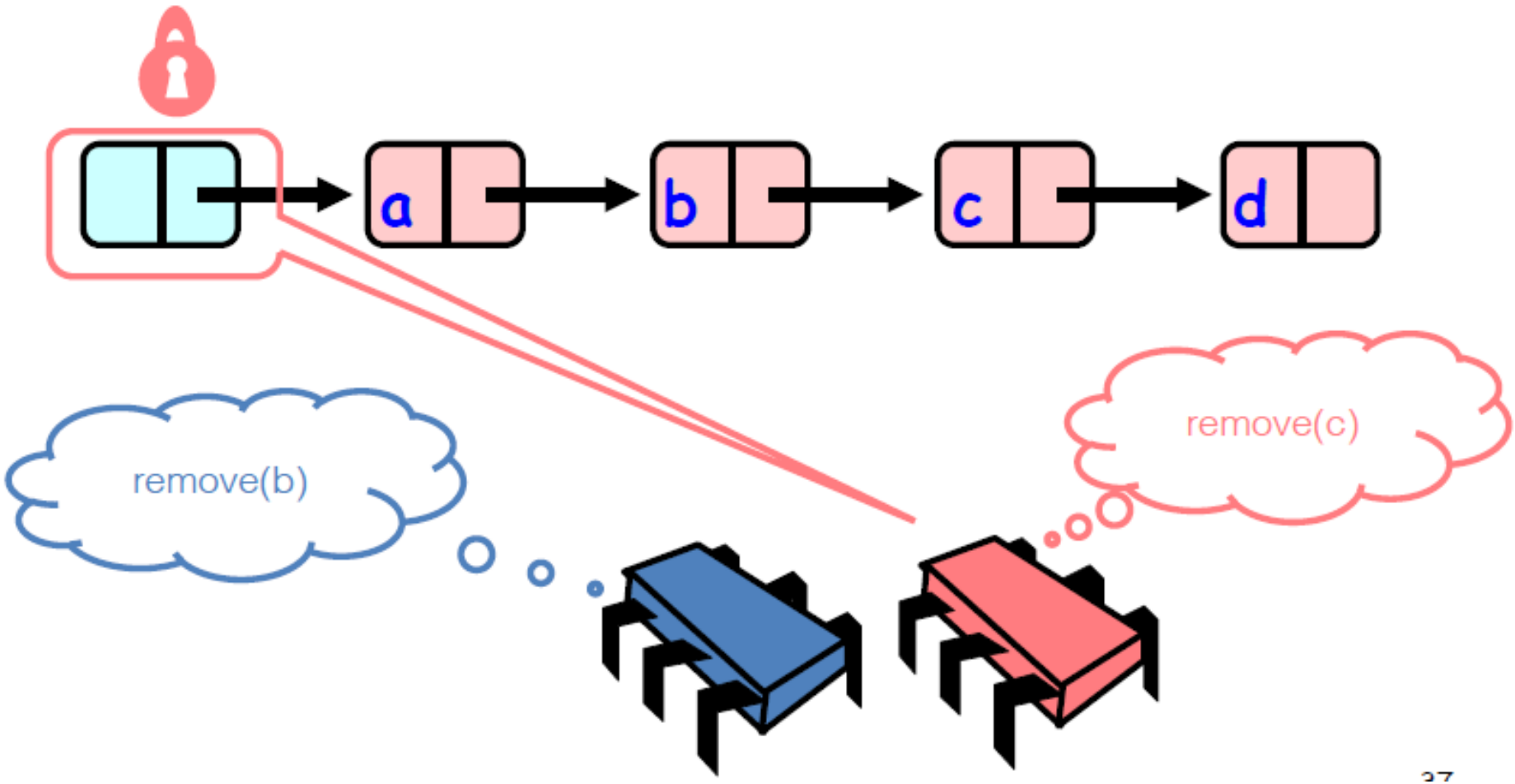


With Two Locks

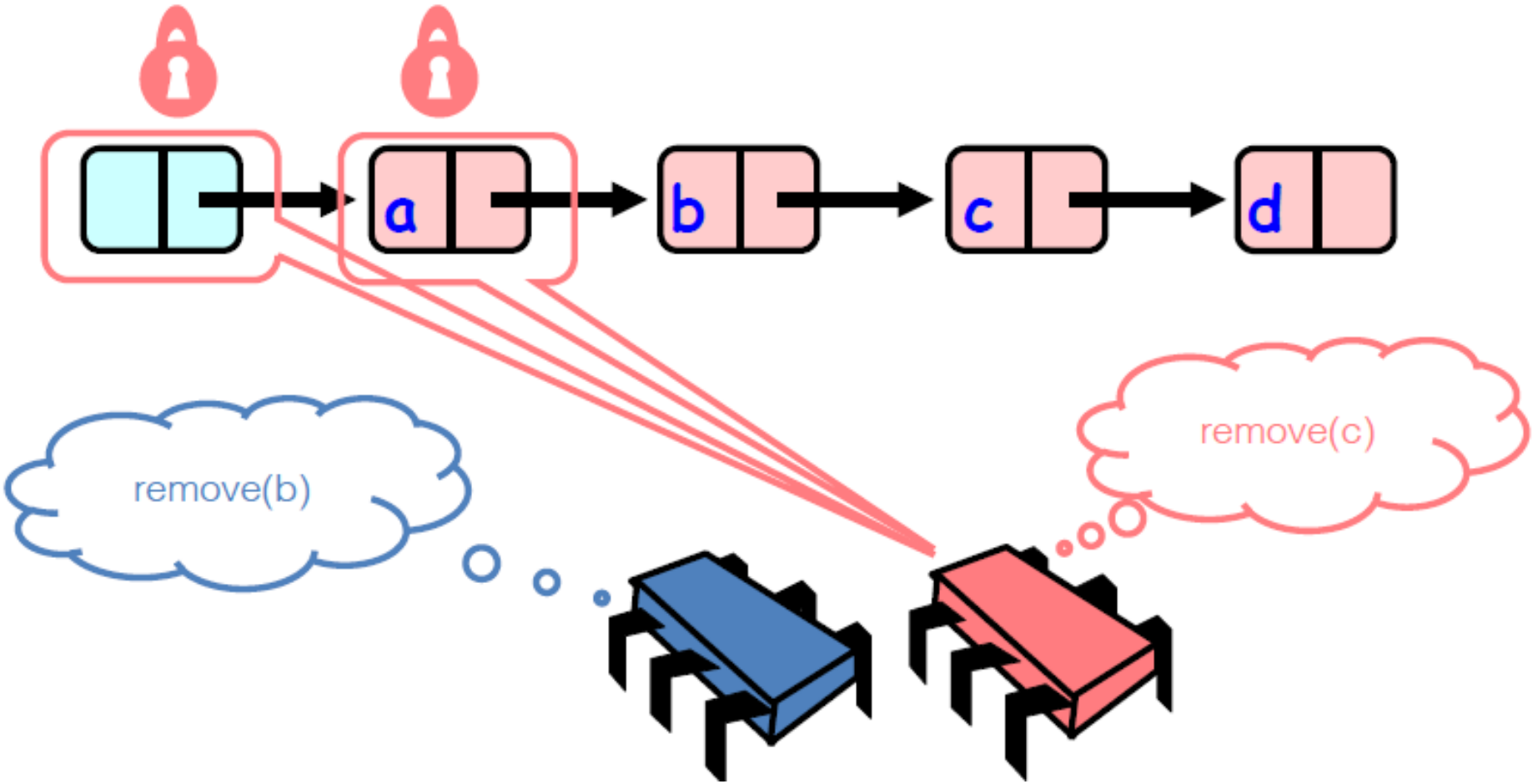




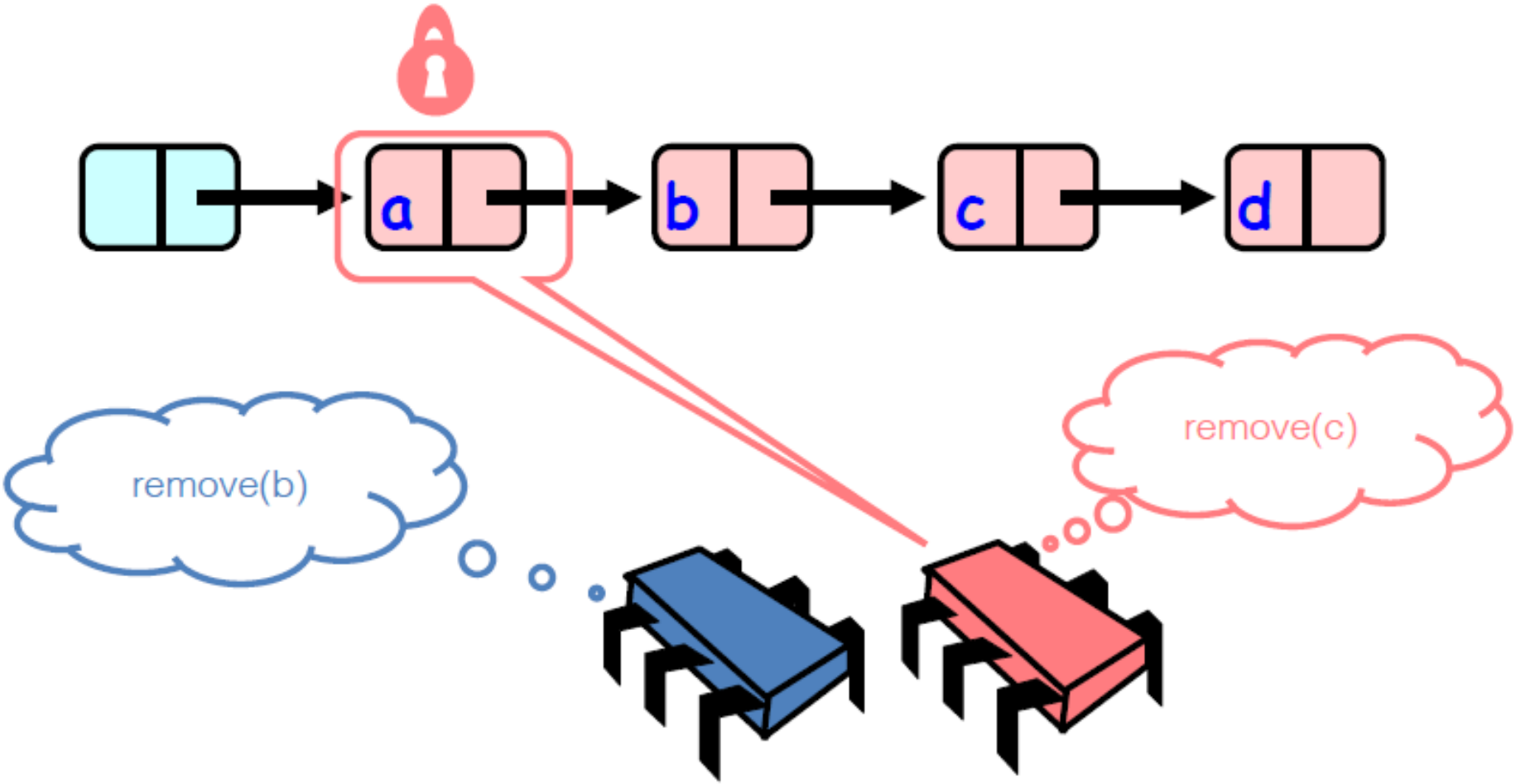
Removing a node



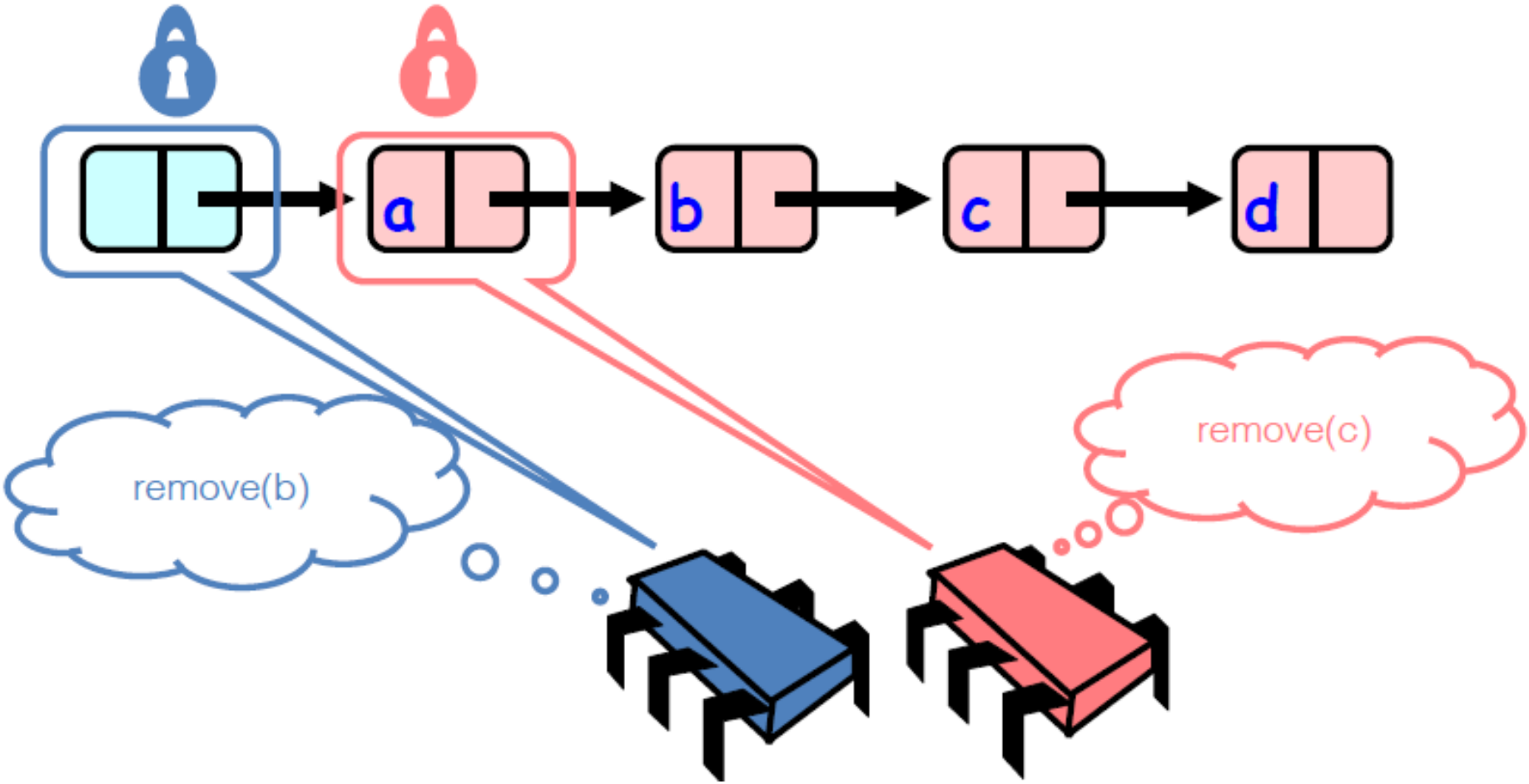
Removing a node



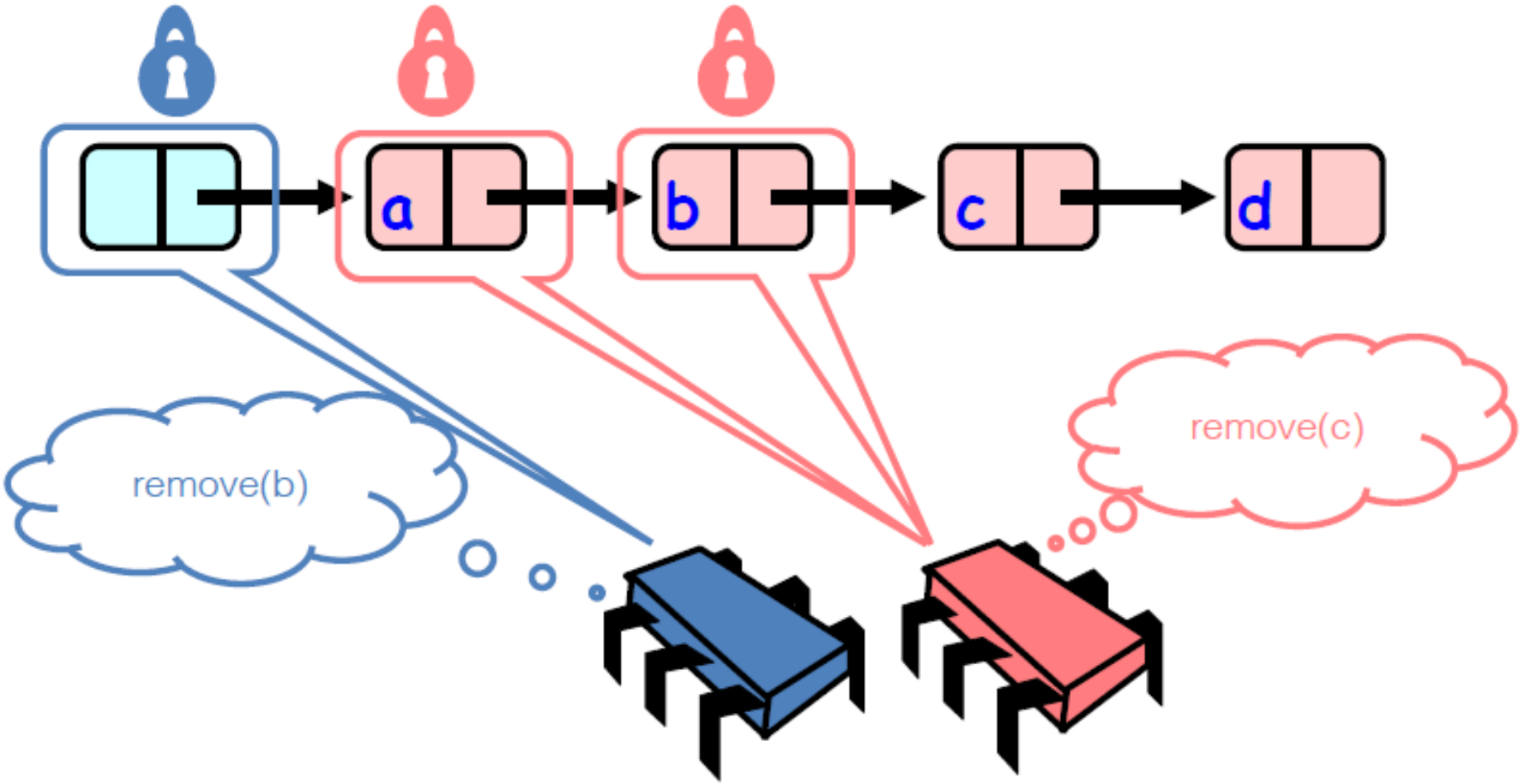
Removing a node



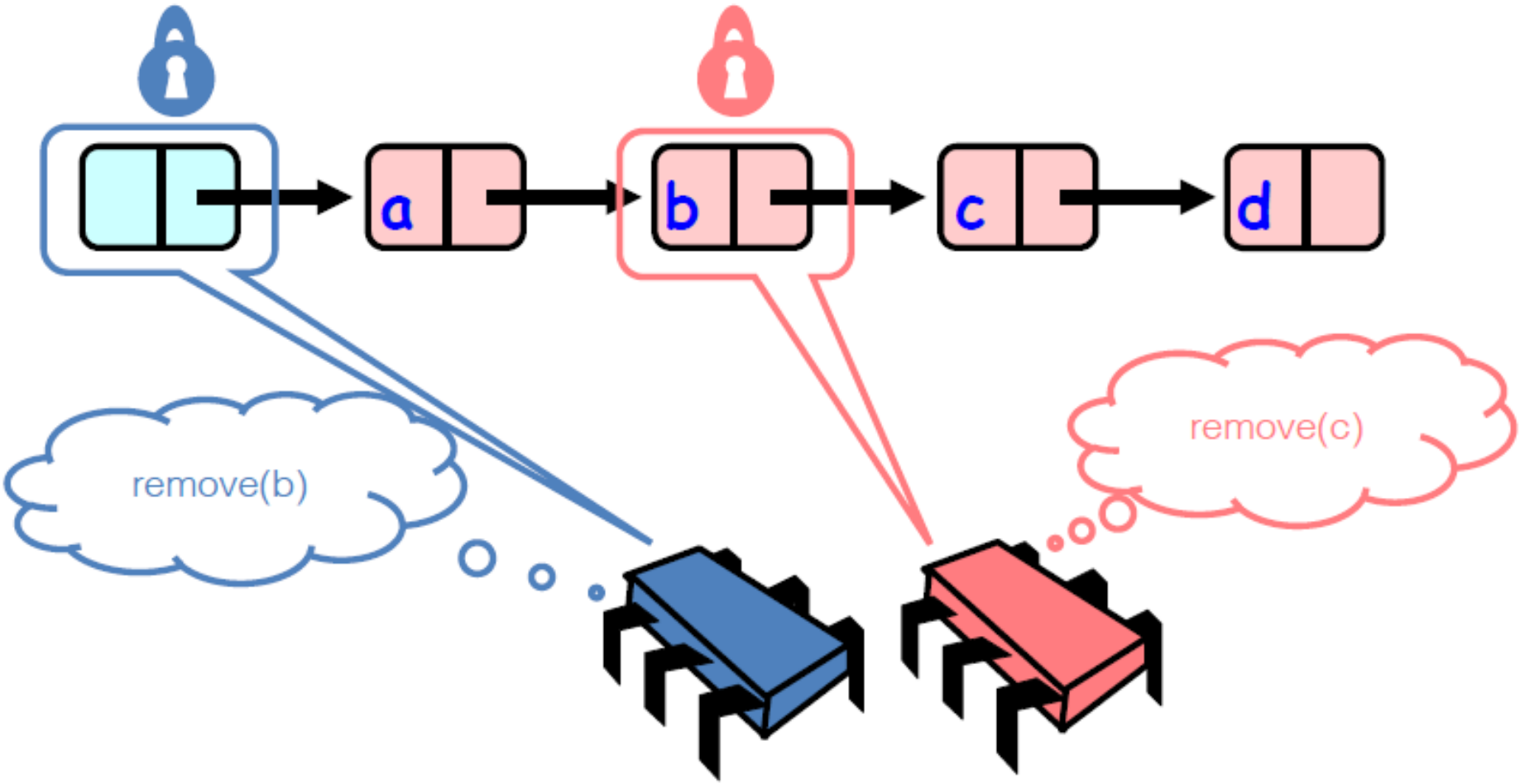
Removing a node



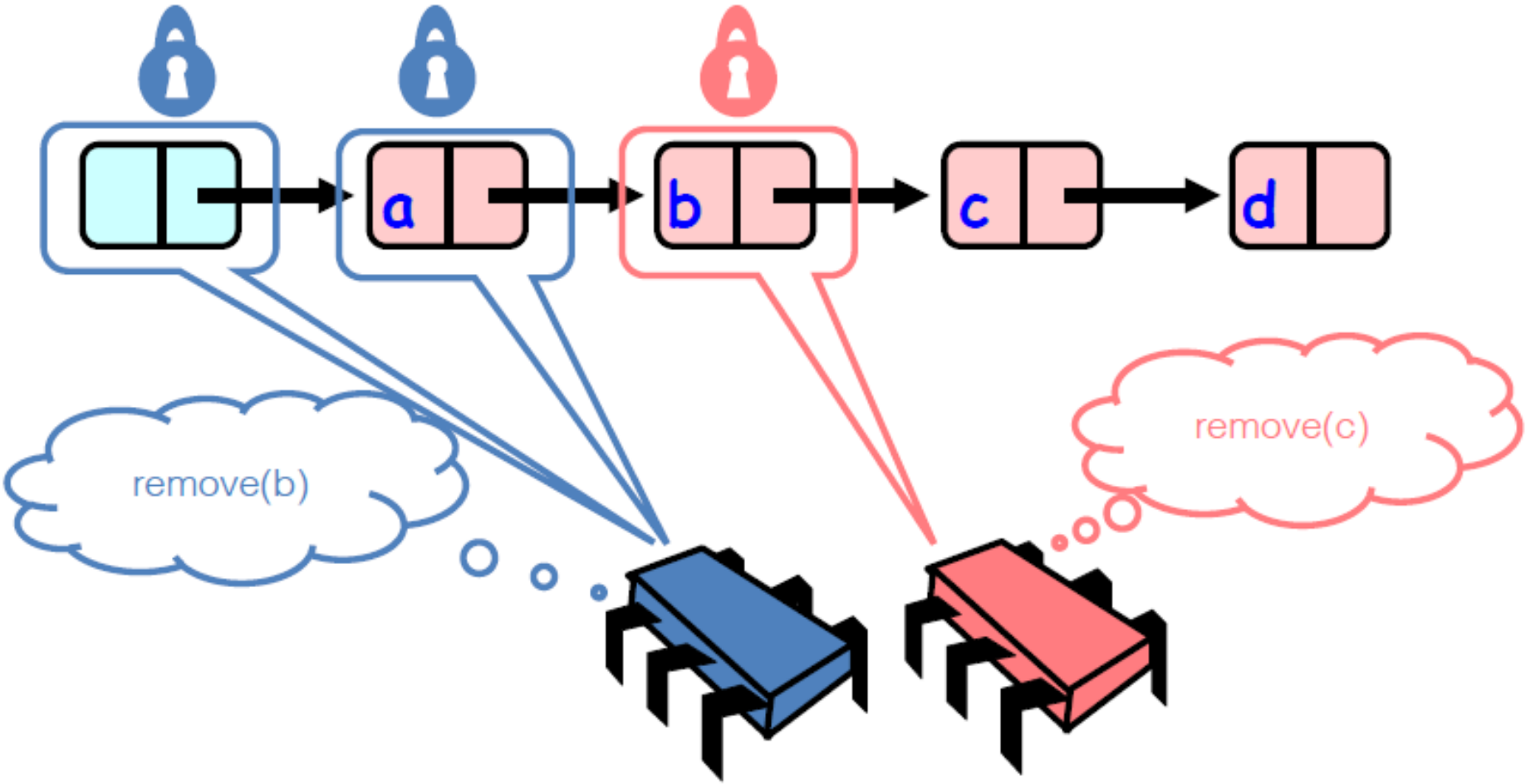
Removing a node



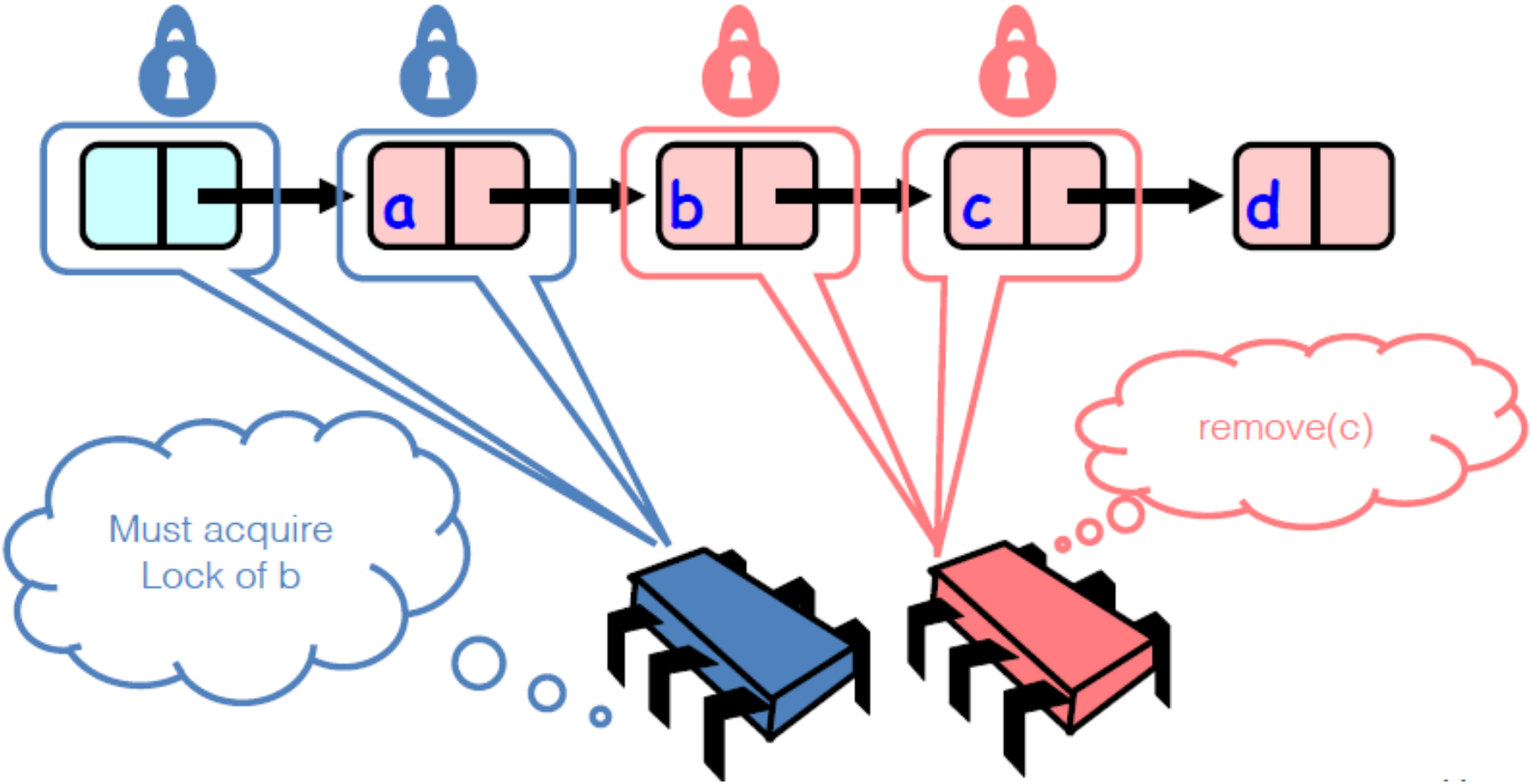
Removing a node



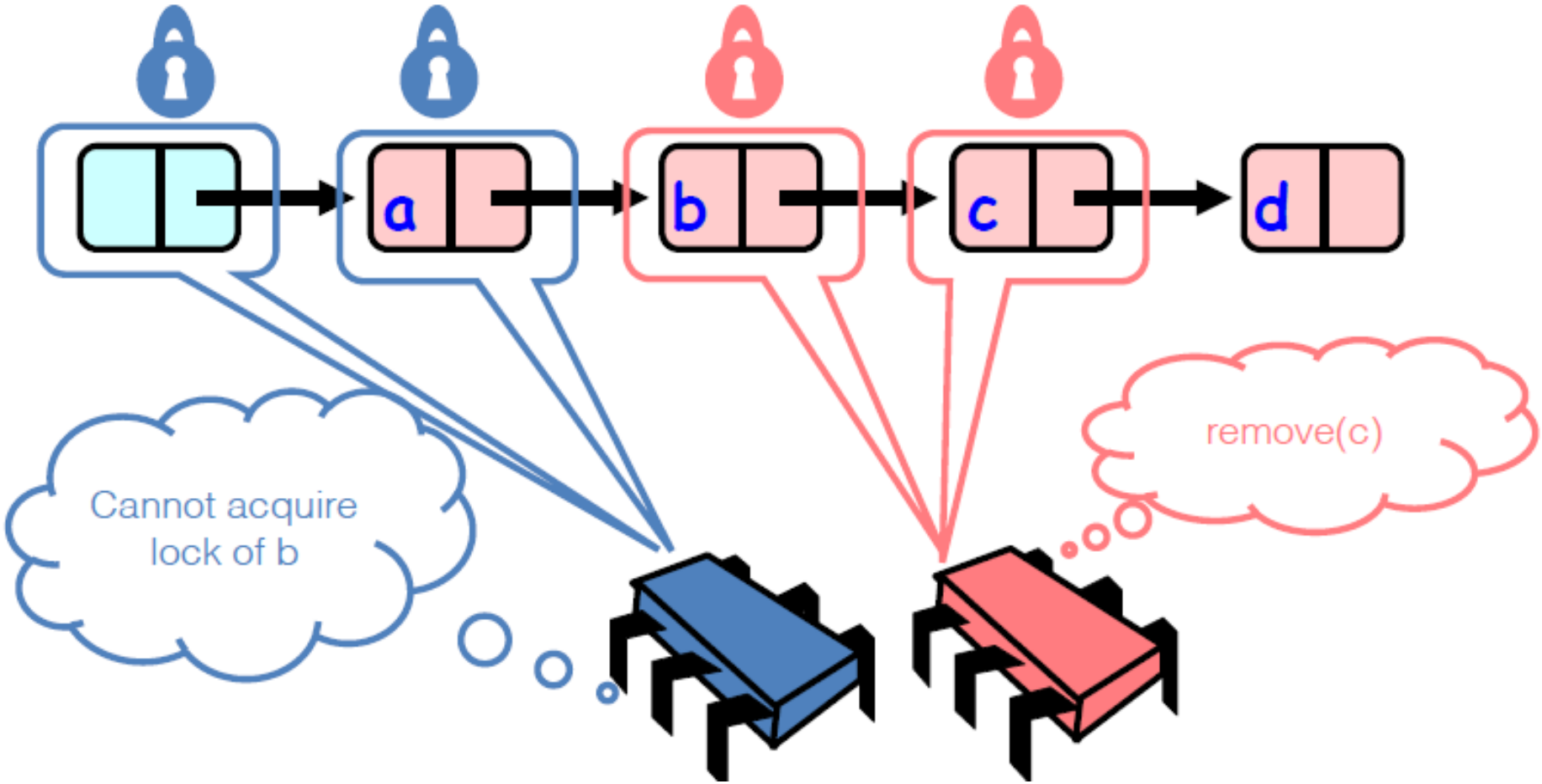
Removing a node



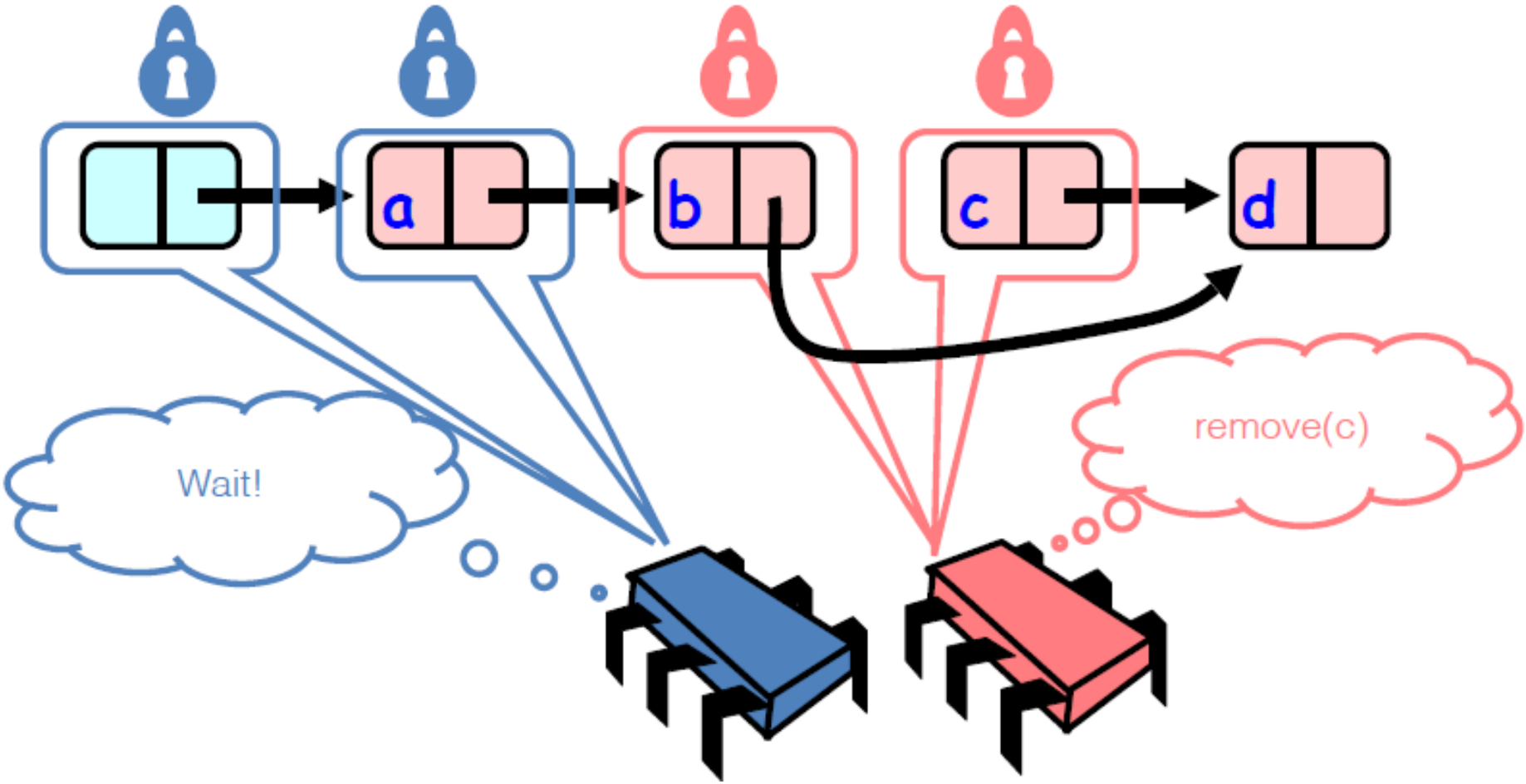
Removing a node



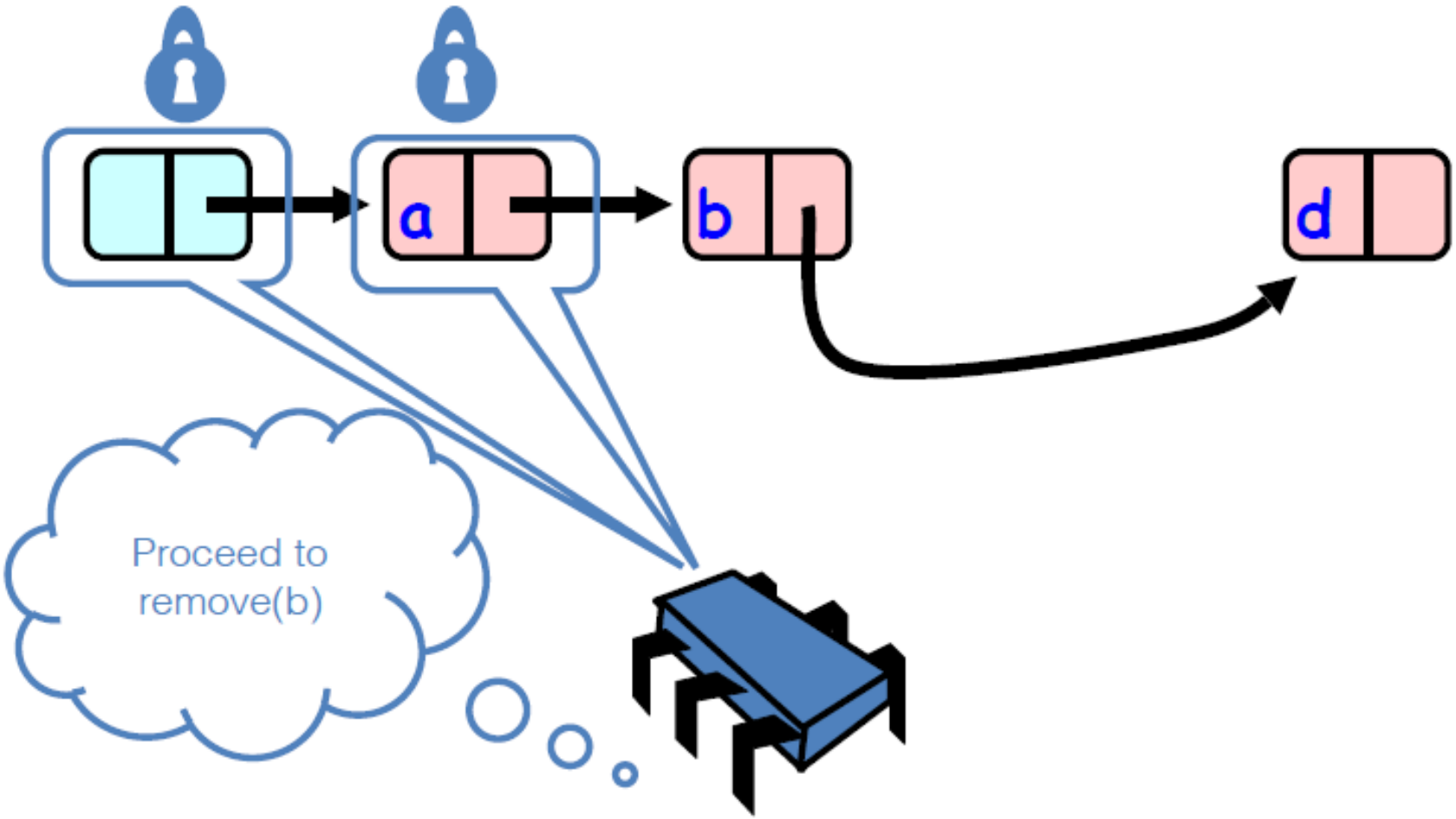
Removing a node



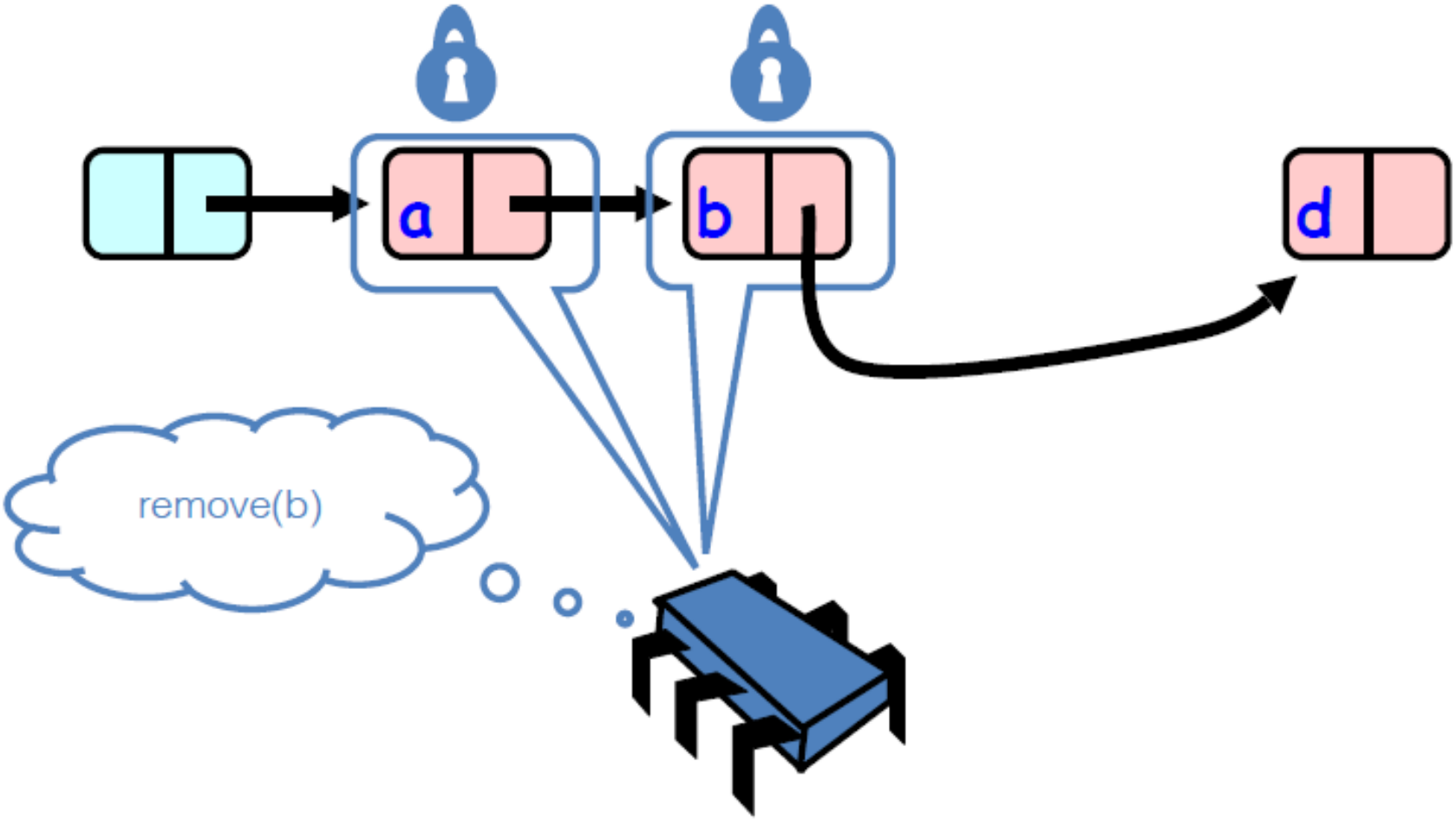
Removing a node



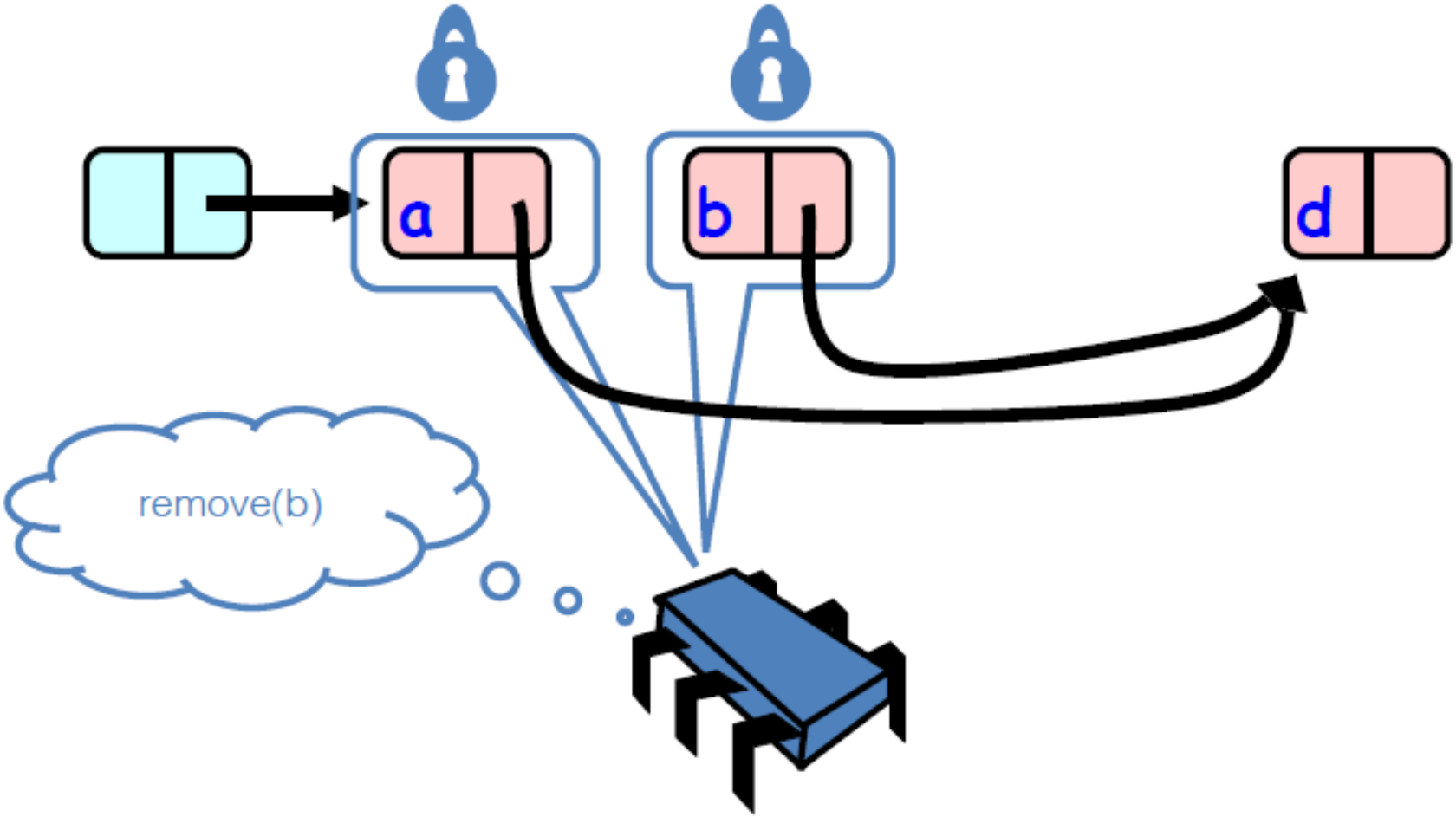
Removing a node



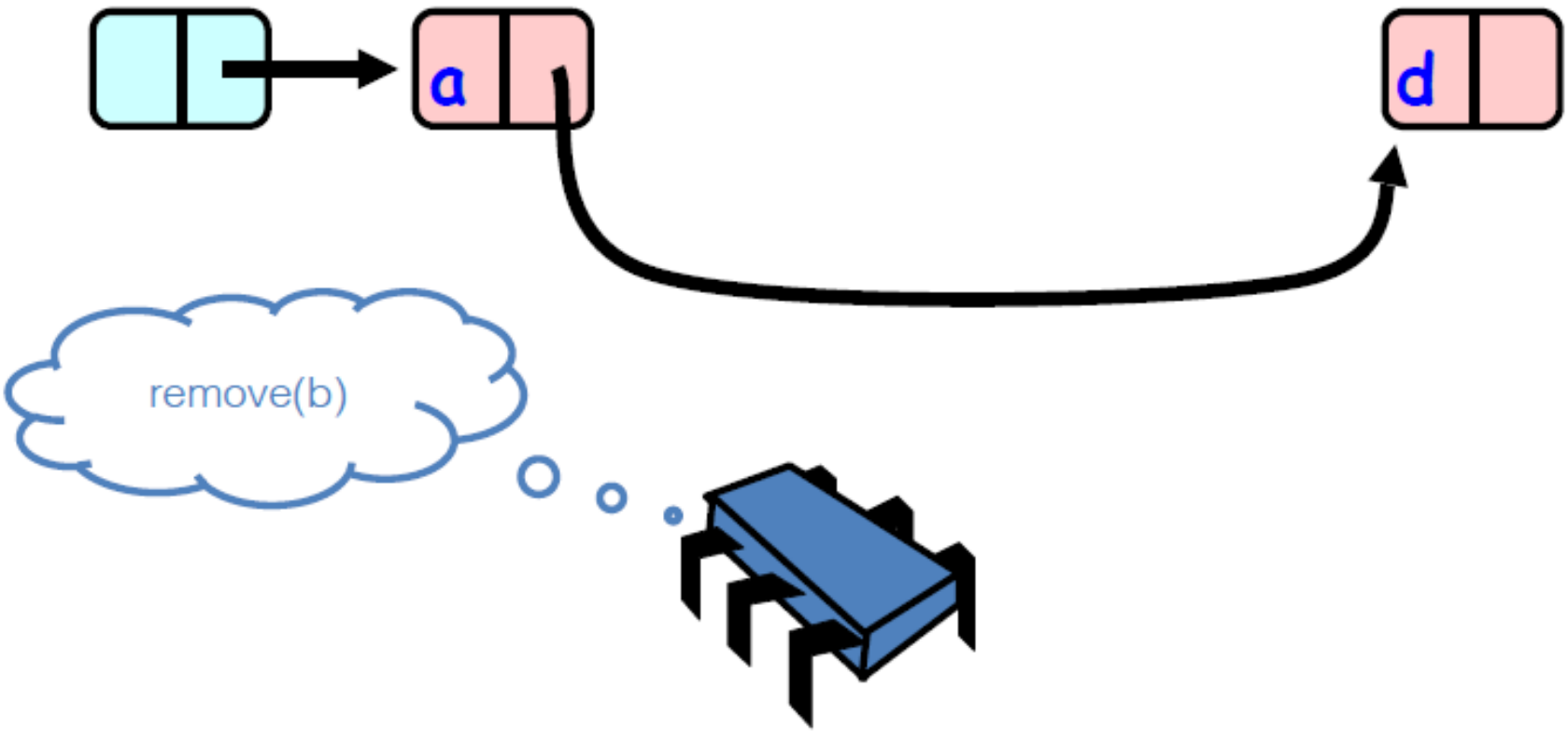
Removing a node



Removing a node



Removing a node



Removing a node