

Concurrency Control of Ordered Linked List



胡卉芪
华东师范大学
数据科学与工程学院
hqhu@dase.ecnu.edu.cn

讨论一个有序链表的并发 控制相关技术

多核处理编程的艺术-第9章

模型：有序链表

- 假定有序链表不允许重复元素
- 方法
 - `add(x)` 增加一个元素
 - `remove(x)` 删除一个元素
 - `contains(x)` 查找一个元素
- 每个节点包括
 - `Key`
 - 一个指向后面节点的指针 `next`

基本操作

Add()

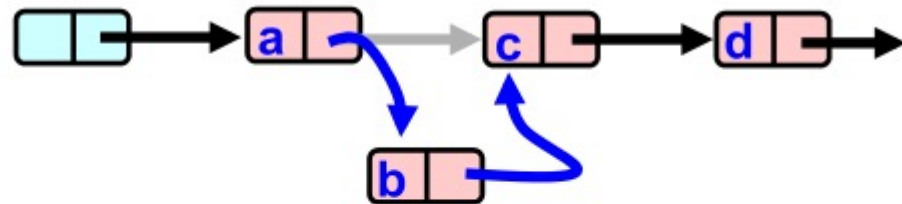
```
node.next = curr;
```

```
pred.next = node
```

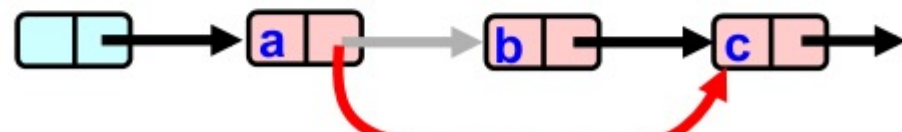
Remove()

```
pred.next = curr.next
```

add()



remove()

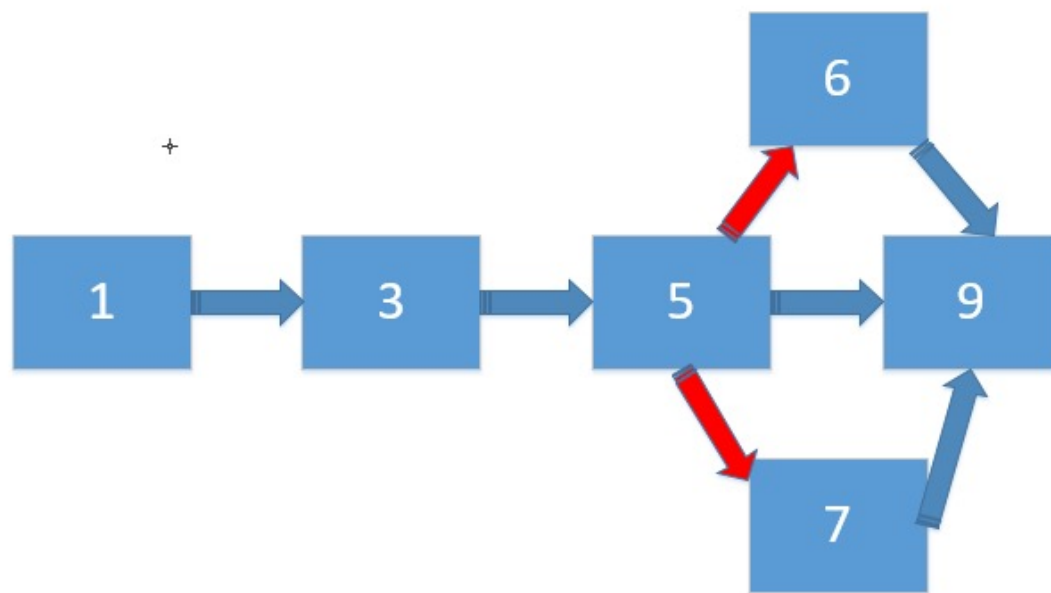


并发控制上的异常

- add和add
- remove和remove
- add和remove
- contains和add
- contains和remove

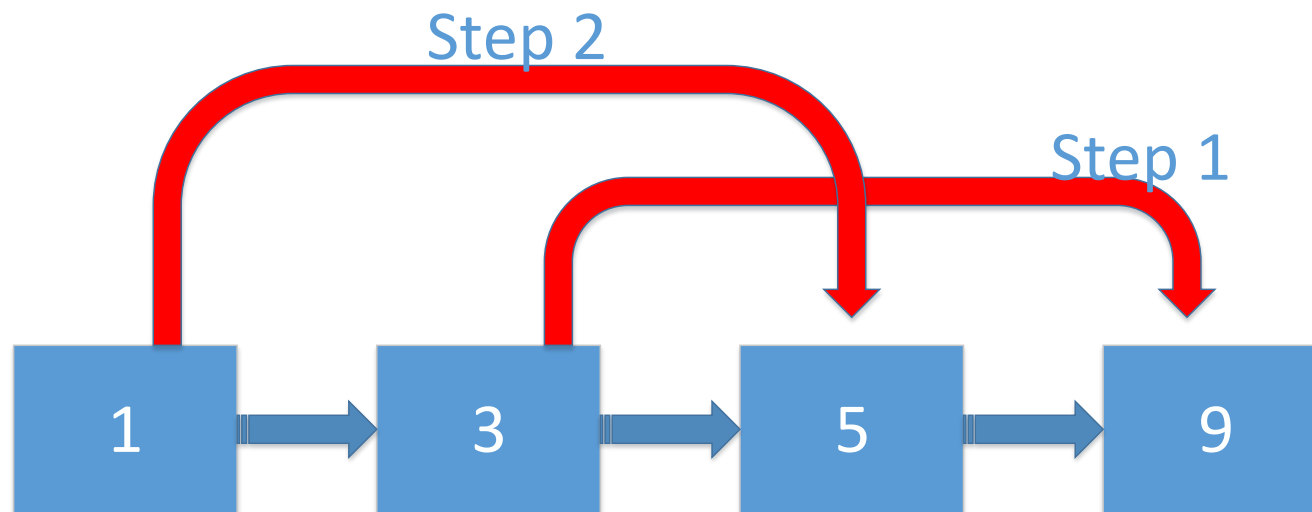
并发控制的异常

- add和add
 - 有丢失更新的问题
 - 同时插入6和7



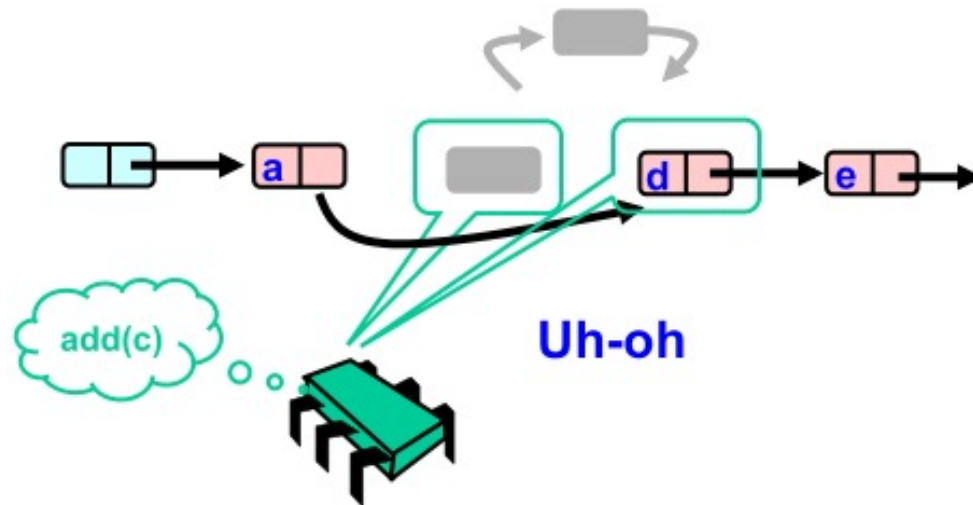
并发控制的异常

- remove和remove
– 假删除问题
– 同时删除3, 5, 5未真正删除



并发控制上的异常

- add和remove
 - 插入的后节点被删除
 - 插入的前节点被删除
 - 删除b, 插入c



并发控制上的异常

- contains和add
- contains和remove
 - 拿到一个删除节点的前驱指针
 - 拿到一个删除节点的指针

并发控制的方法

- 粗粒度锁
- 细粒度锁 (hand over hand locking 或lock coupling)
- 乐观锁
- 懒惰锁
- 无锁编程(原子操作)

Lock Coupling

粗粒度锁

- 对add(), remove(), contains()三个操作
 - 访问链表时加锁，操作完成后释放锁
 - 最安全
 - 效率最差

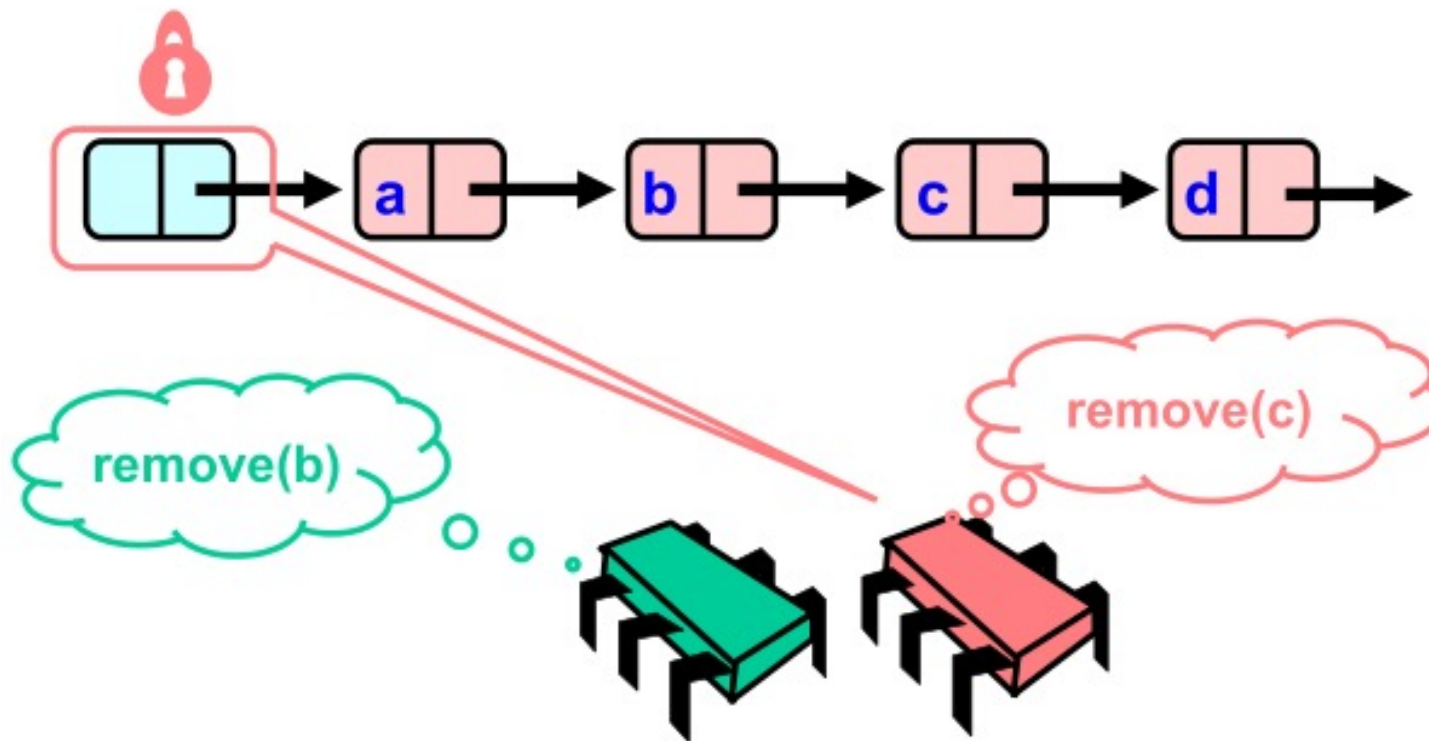
Hand over hand Locking

- 链表向前推进时先获取锁时，先获取curr锁，然后再释放prev锁
- 数据操作时，同时锁住前驱和后继

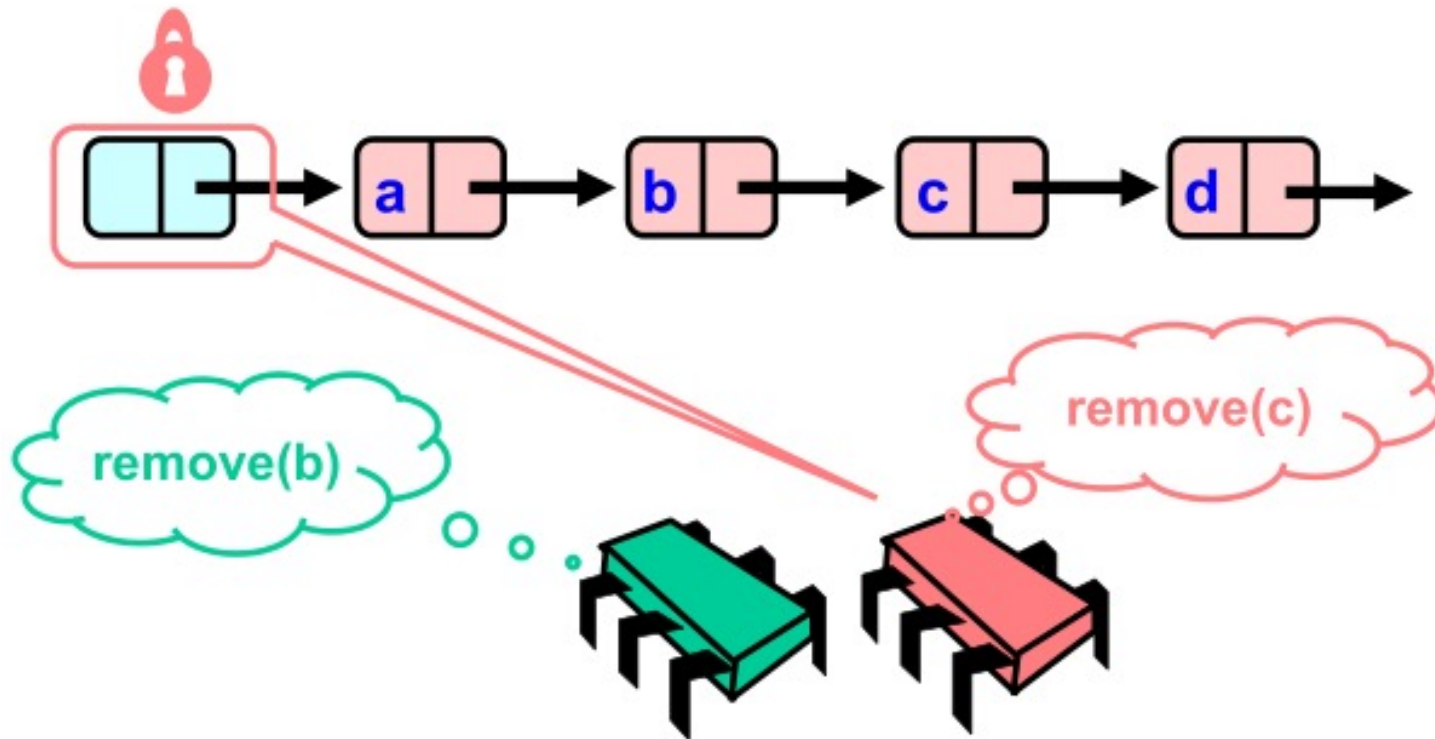
Lock Coupling的一些代码

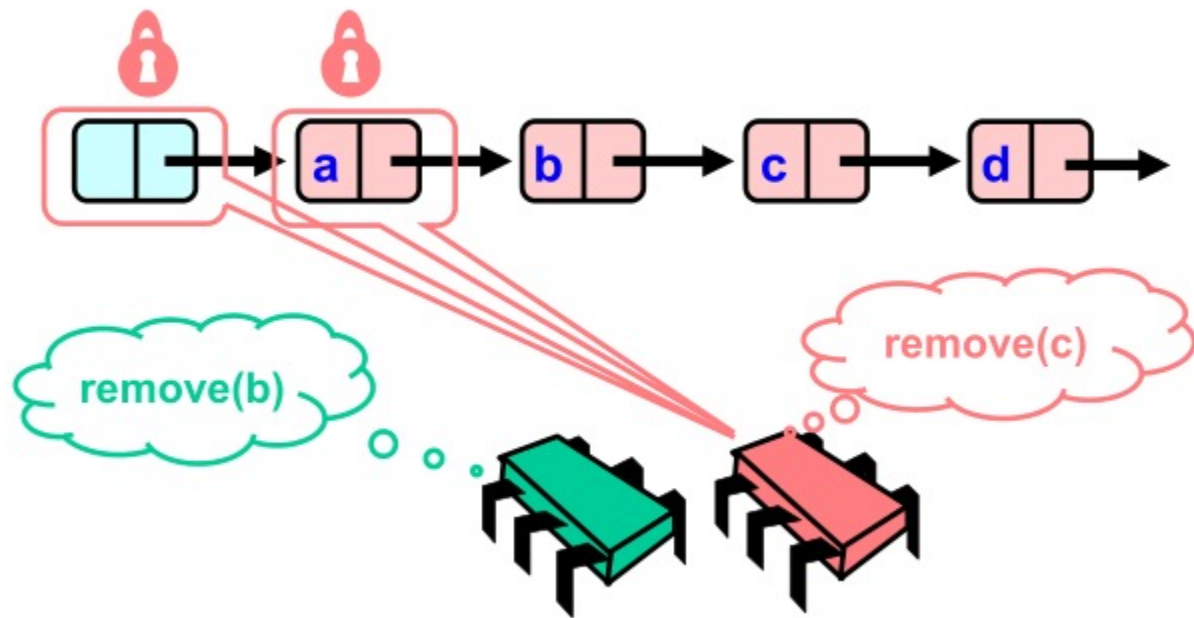
```
head.lock();
pred = head;
try{
    curr = pred.next;
    curr.lock();
    try{
        while(curr.key < key){
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
        }
        if(curr.key == key){
            return false;
        }
        NodeWithLock<T> node = new NodeWithLock<T>(item);
        node.next = curr;
        pred.next = node;
        return true;
    }finally{
        curr.unlock();
    }
}finally{
    pred.unlock();
}
```

Lock Coupling 举例

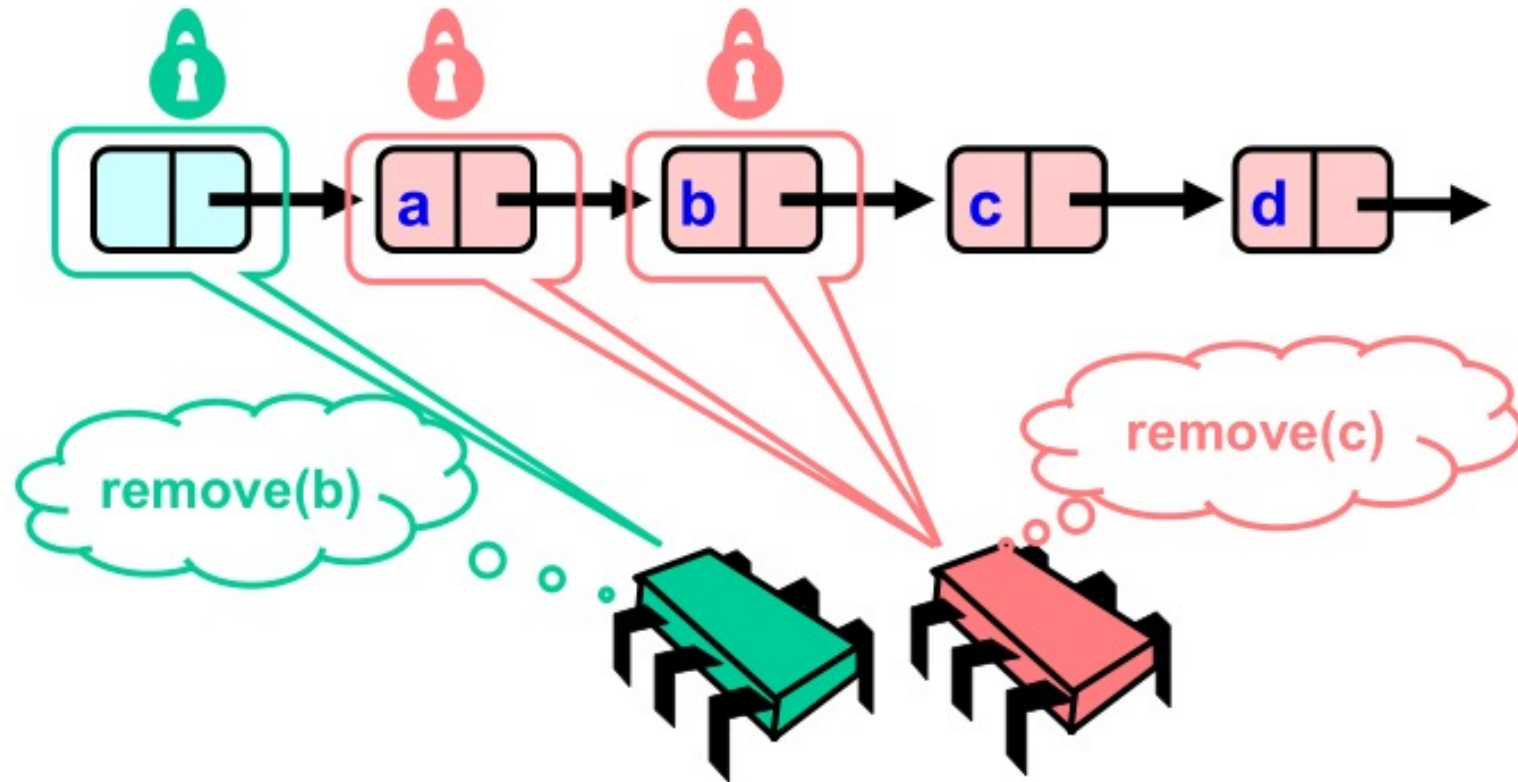


Lock Coupling 举例

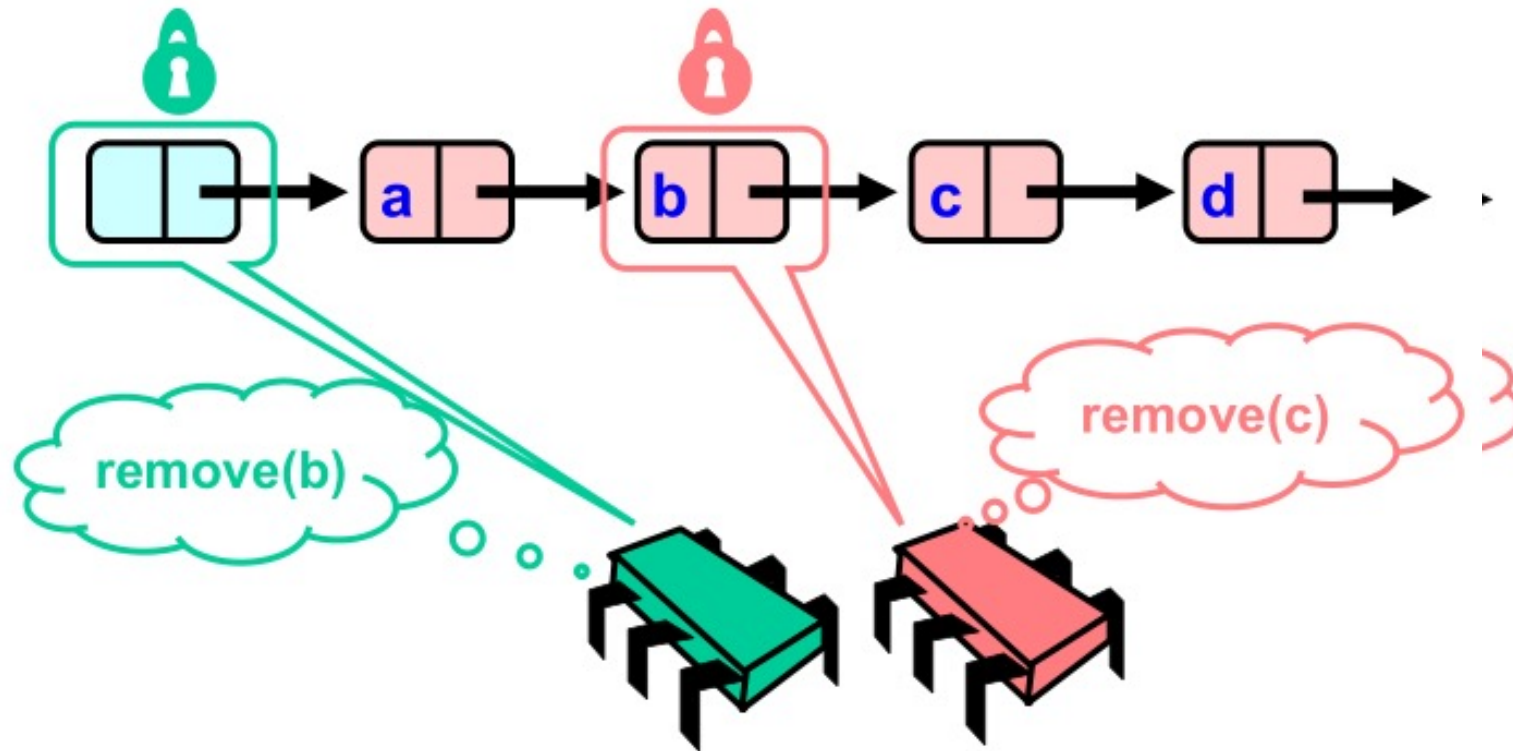


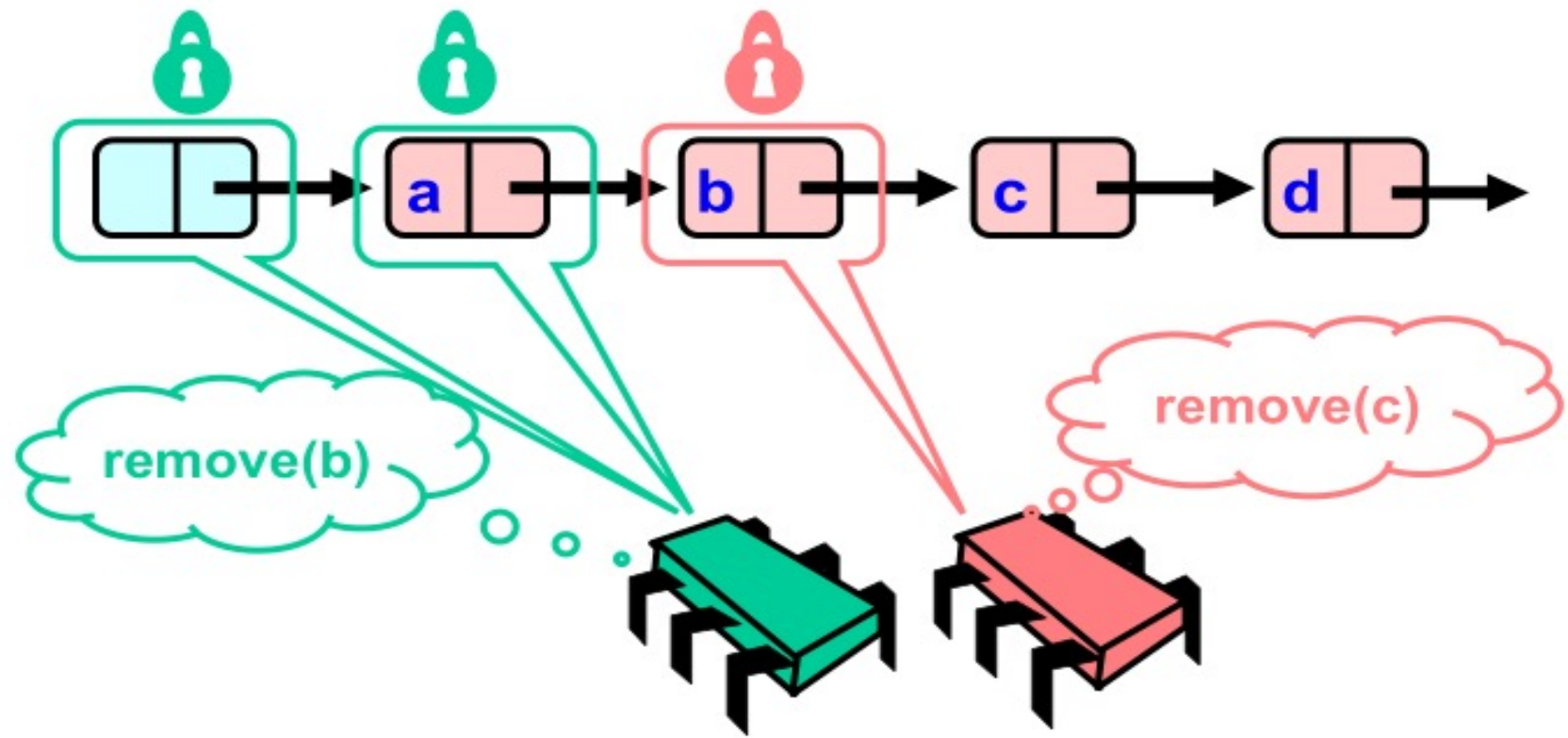


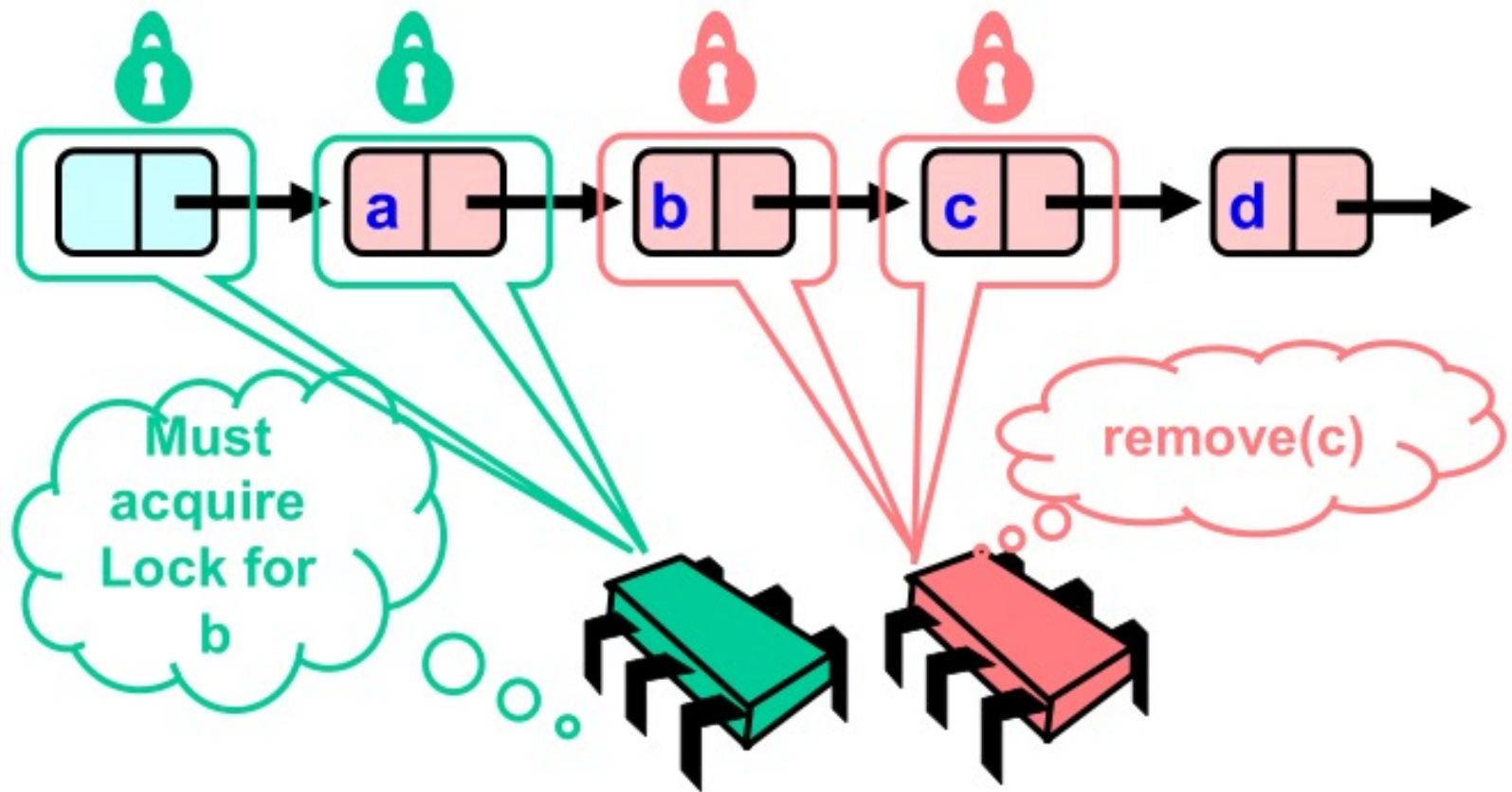
Lock Coupling 举例

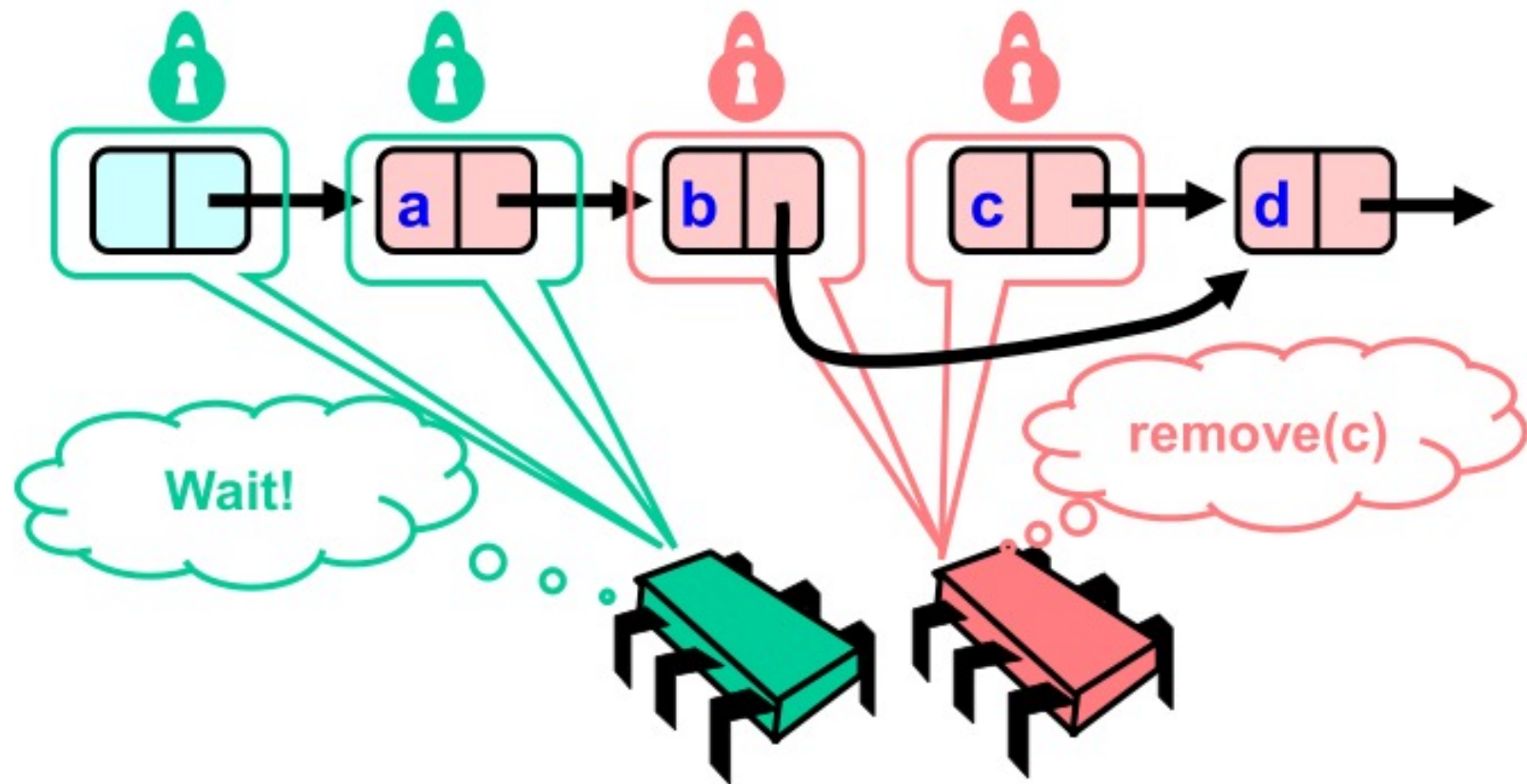


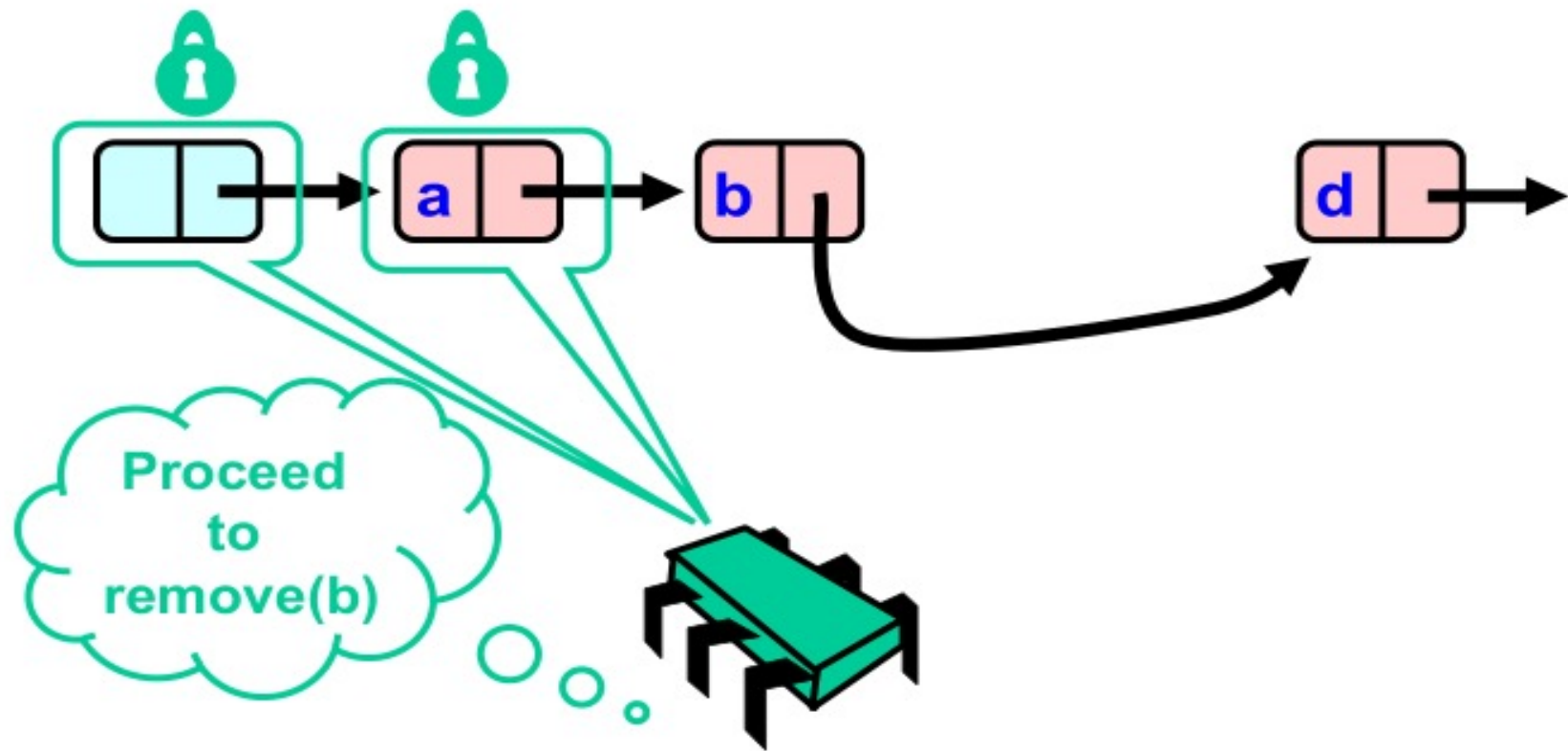
Lock Coupling 举例

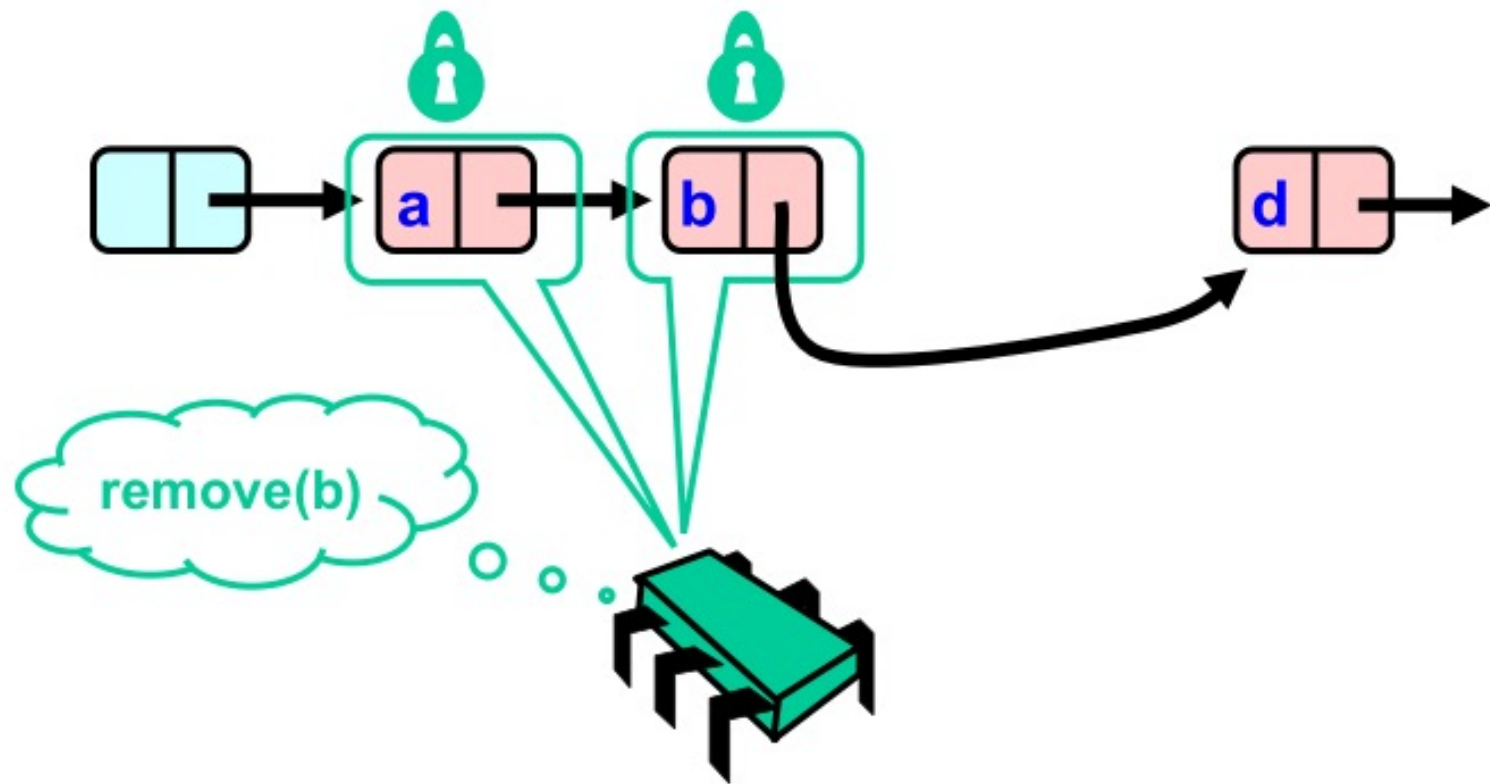




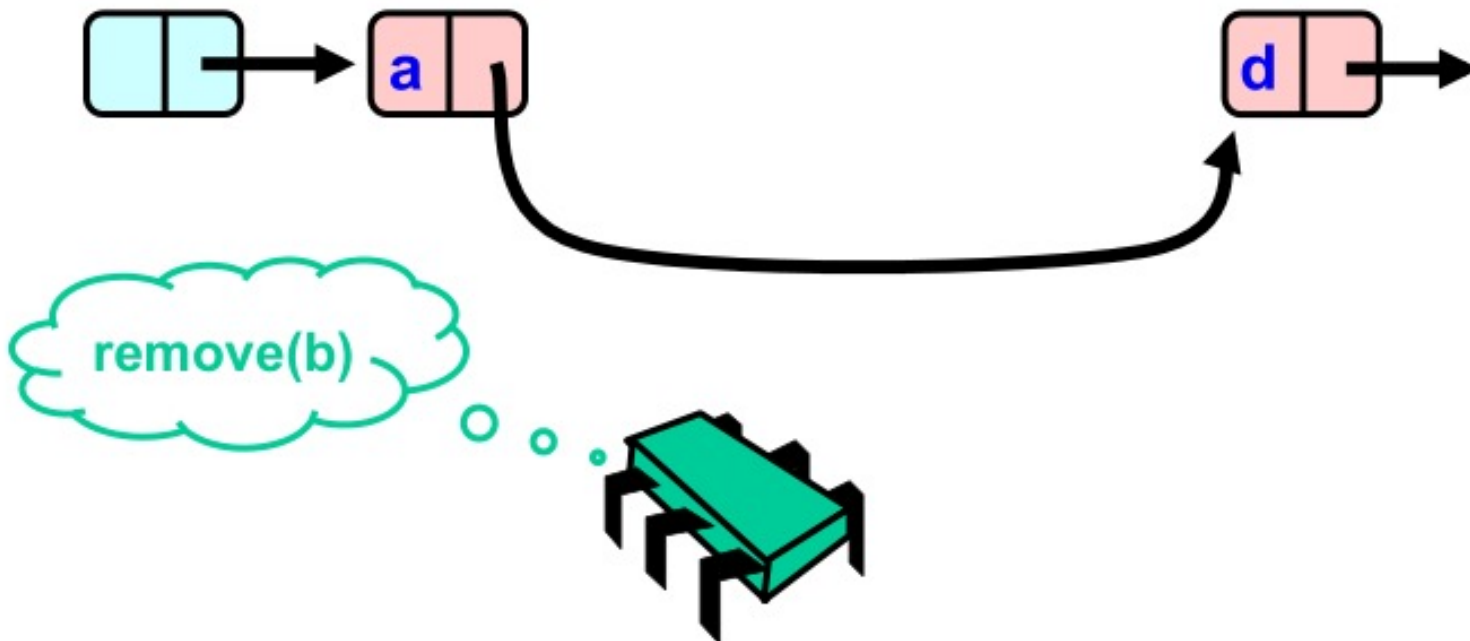








避免了假删除问题



Lock Coupling

- Remove()
 - 锁住删除节点和其前驱节点，然后再删除
- Add()
 - 锁住插入节点的前驱和后续节点，然后插入

如果不做Lock Coupling

- Remove()
 - 如果不锁删除节点？
 - 如果不锁前驱节点？
 - 无法避免假删除问题

如果不做Lock Coupling

- Add()
 - 如果不锁前驱节点?
 - 仍然无法避免丢失更新
 - 如果不锁后继节点?
 - 请大家思考

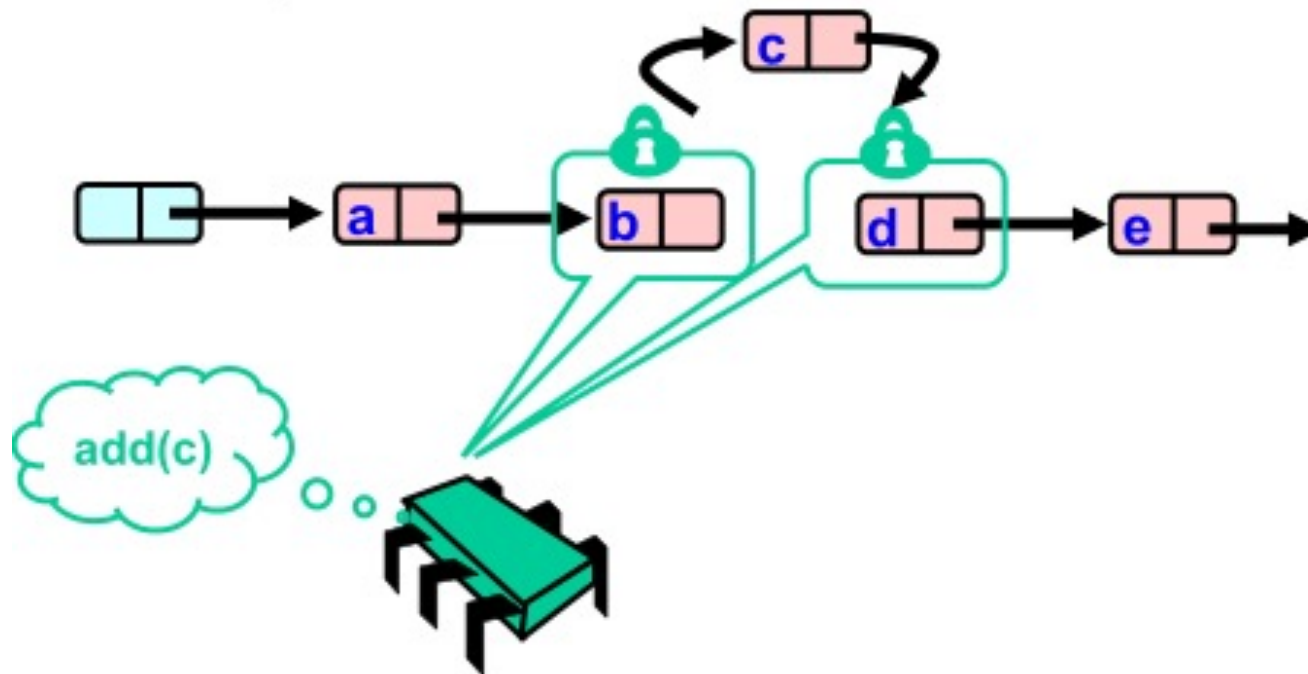
乐观锁

乐观锁

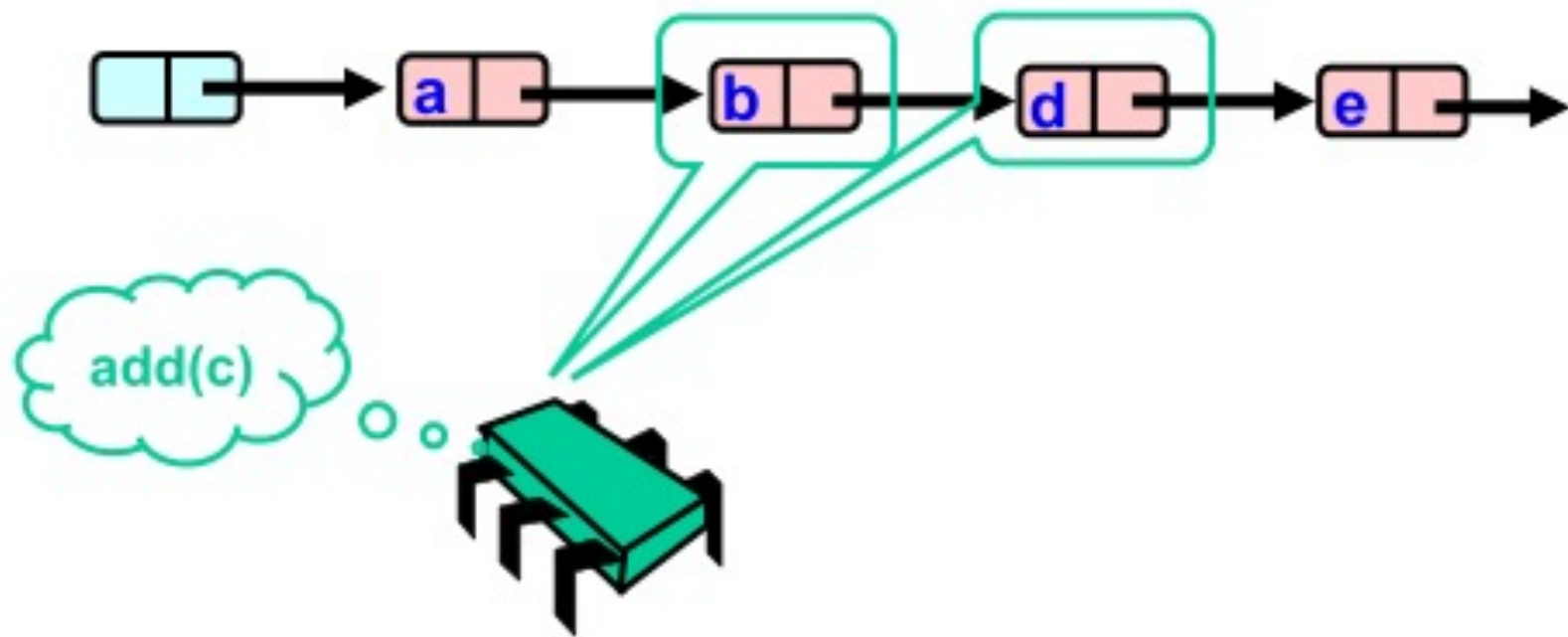
- Lock Coupling需要不停的去获得和释放锁
- 只在需要加锁的时候再加锁
 1. 只有在寻找到要加锁位置的时候才加锁，之前不加锁
 2. 需要加锁时，先加锁，再进行验证是否现场已经被修改
 3. 如果验证失败就需要从头开始重试

无锁遍历

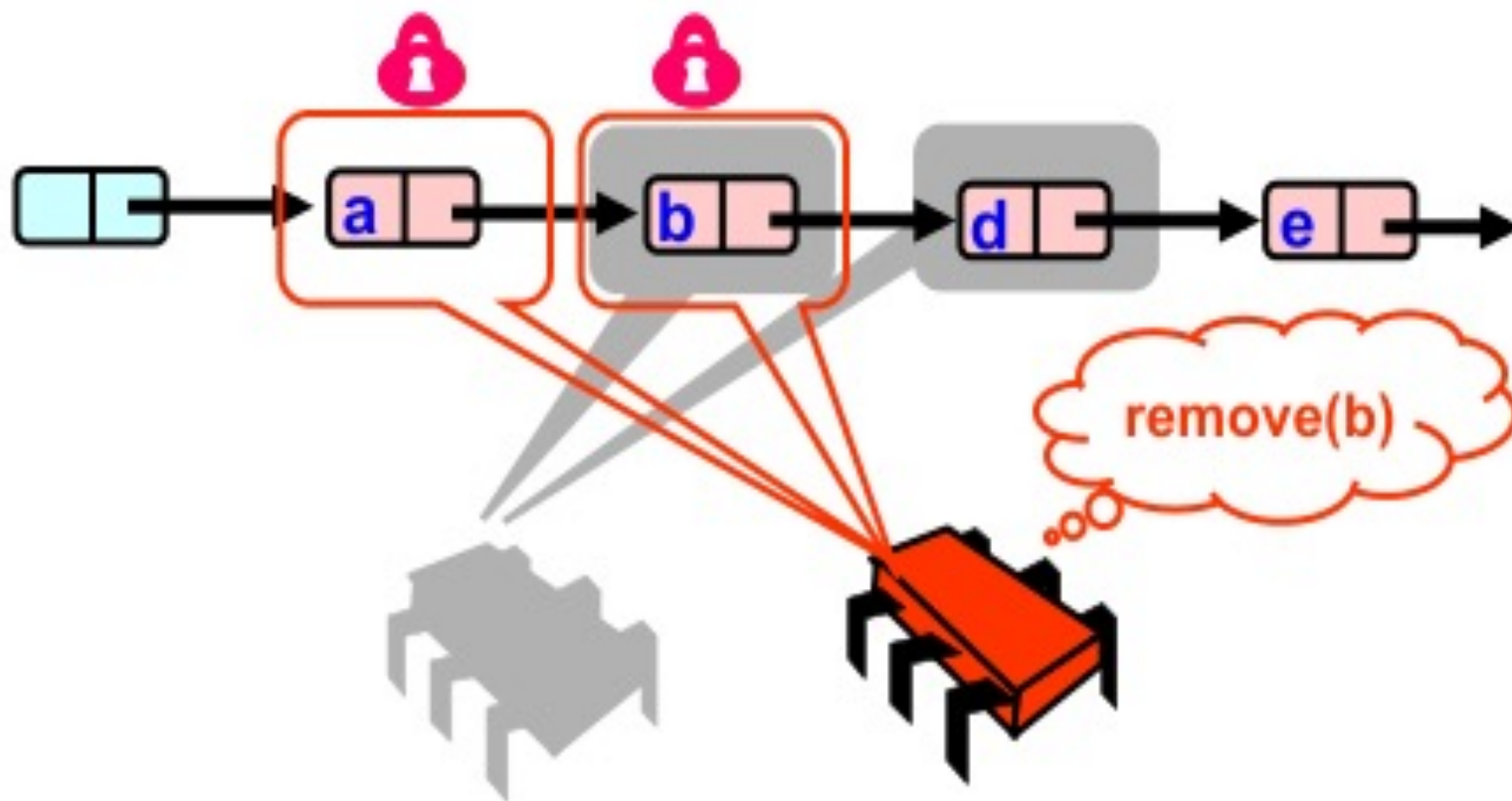
- what could go wrong?



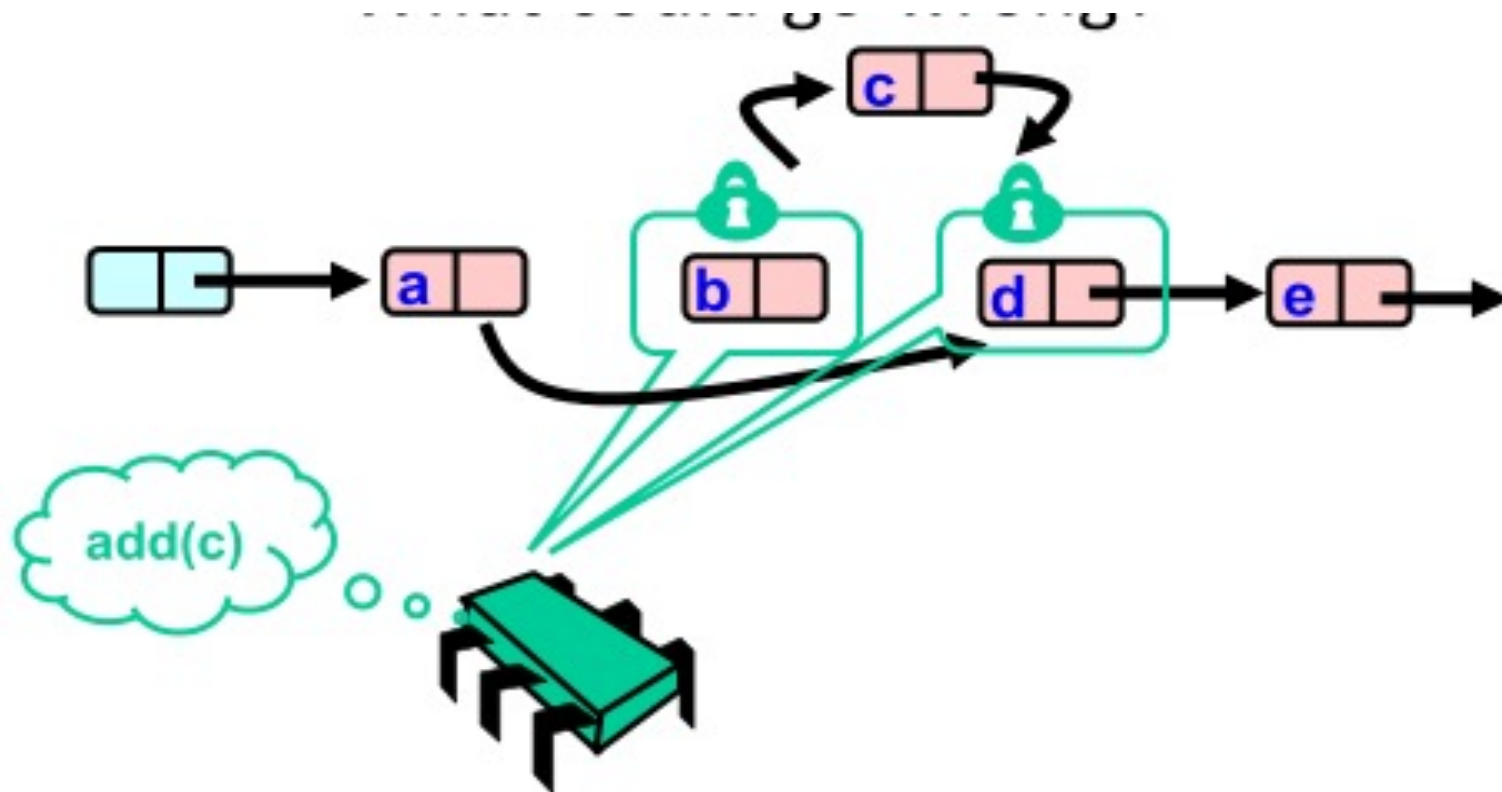
无锁遍历的异常（一）



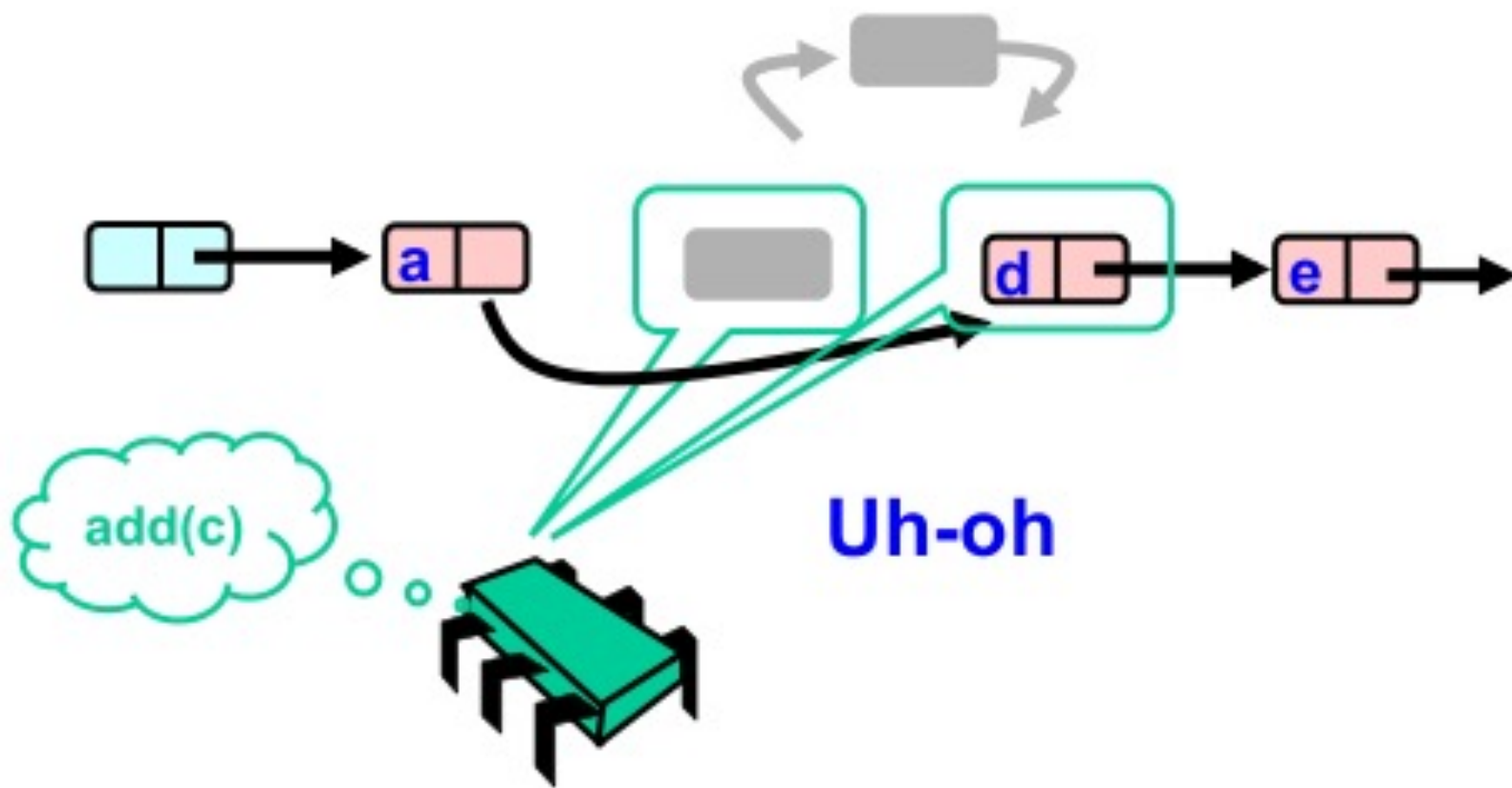
无锁遍历的异常（一）



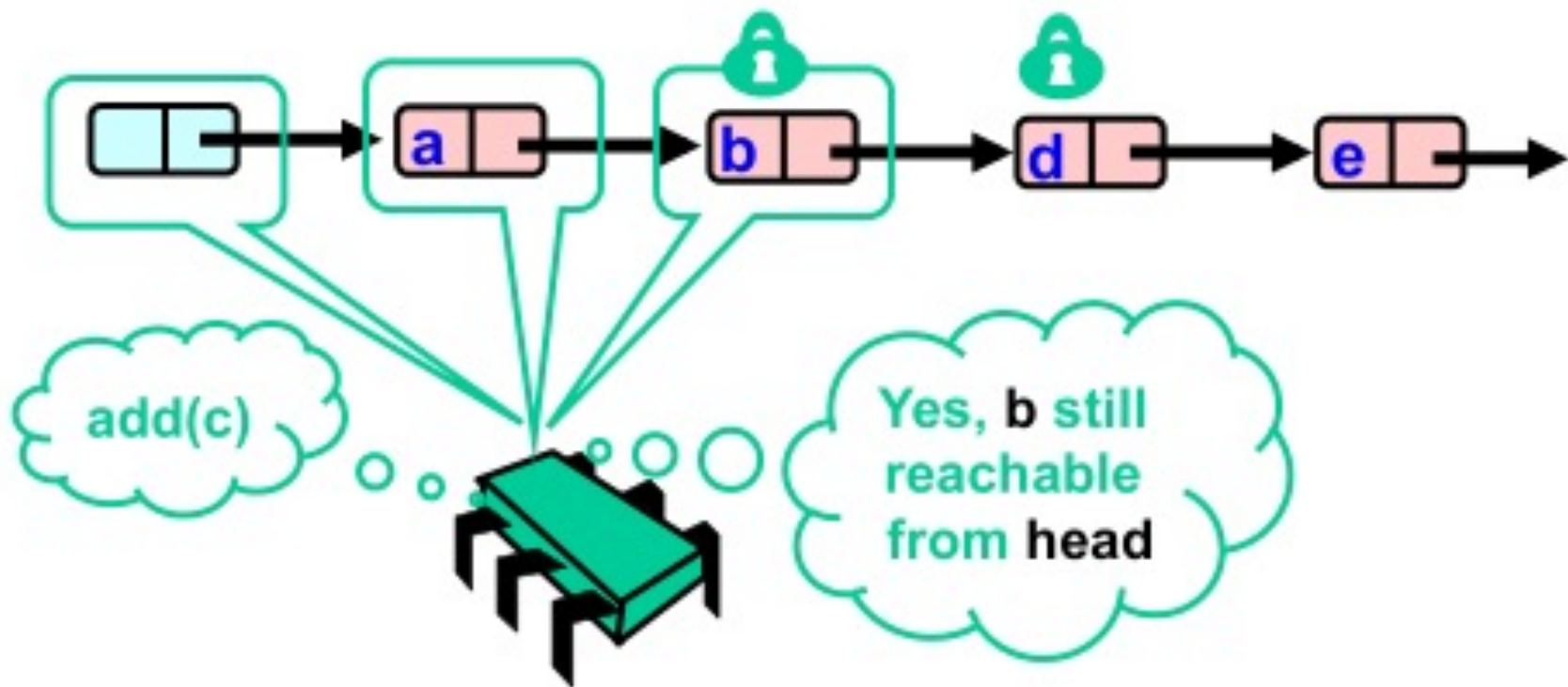
无锁遍历的异常（一）



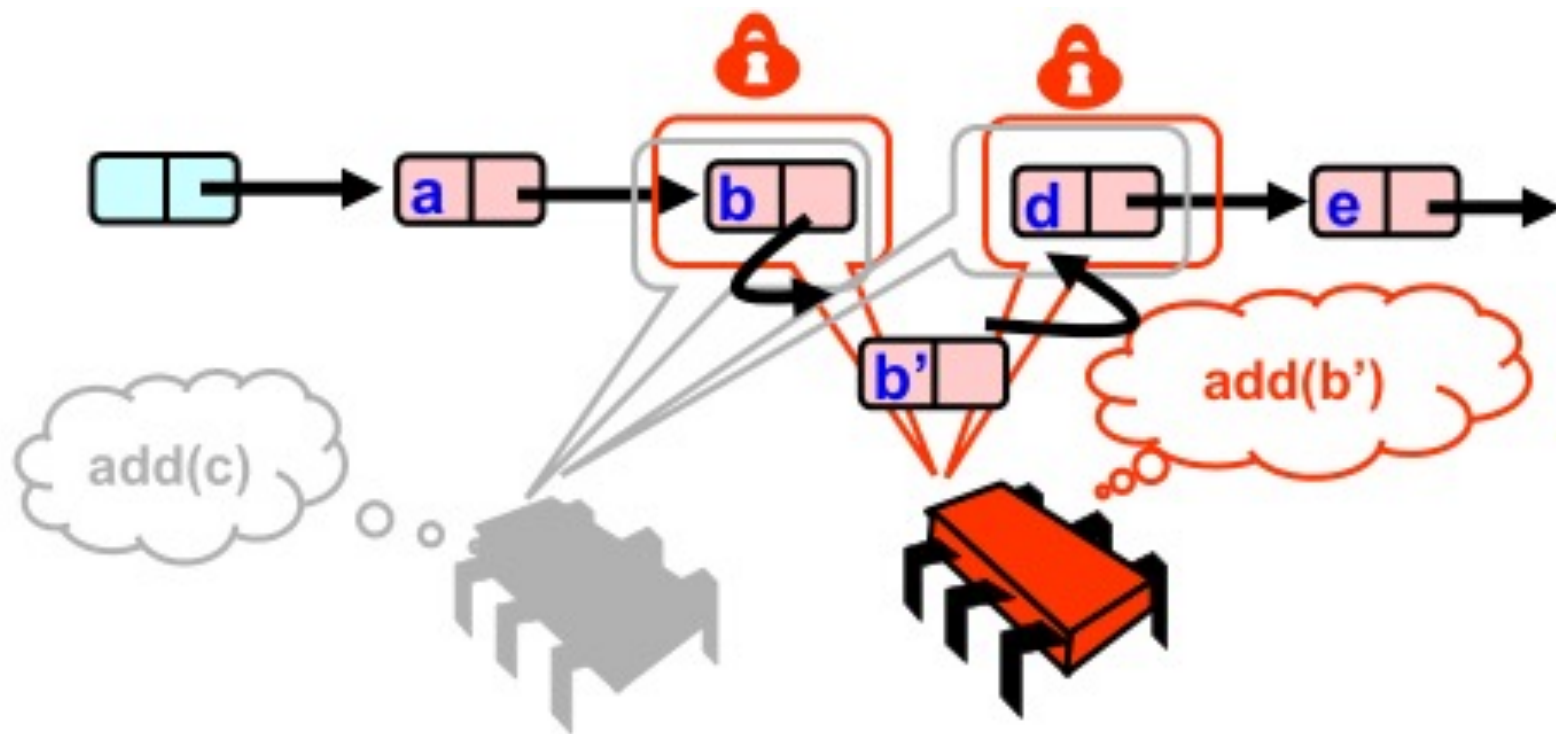
无锁遍历的异常（一）



验证b是否仍然可达

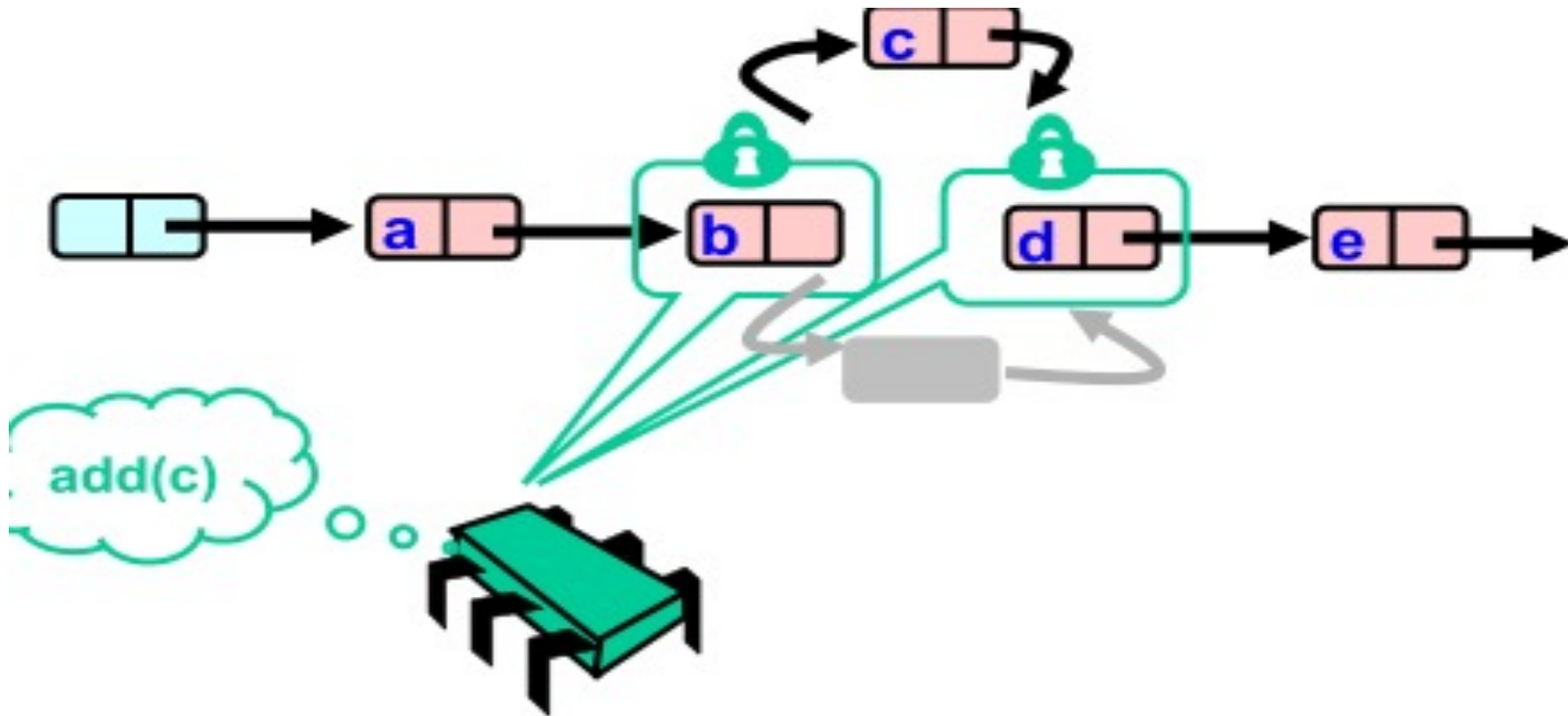


无锁遍历异常(二)

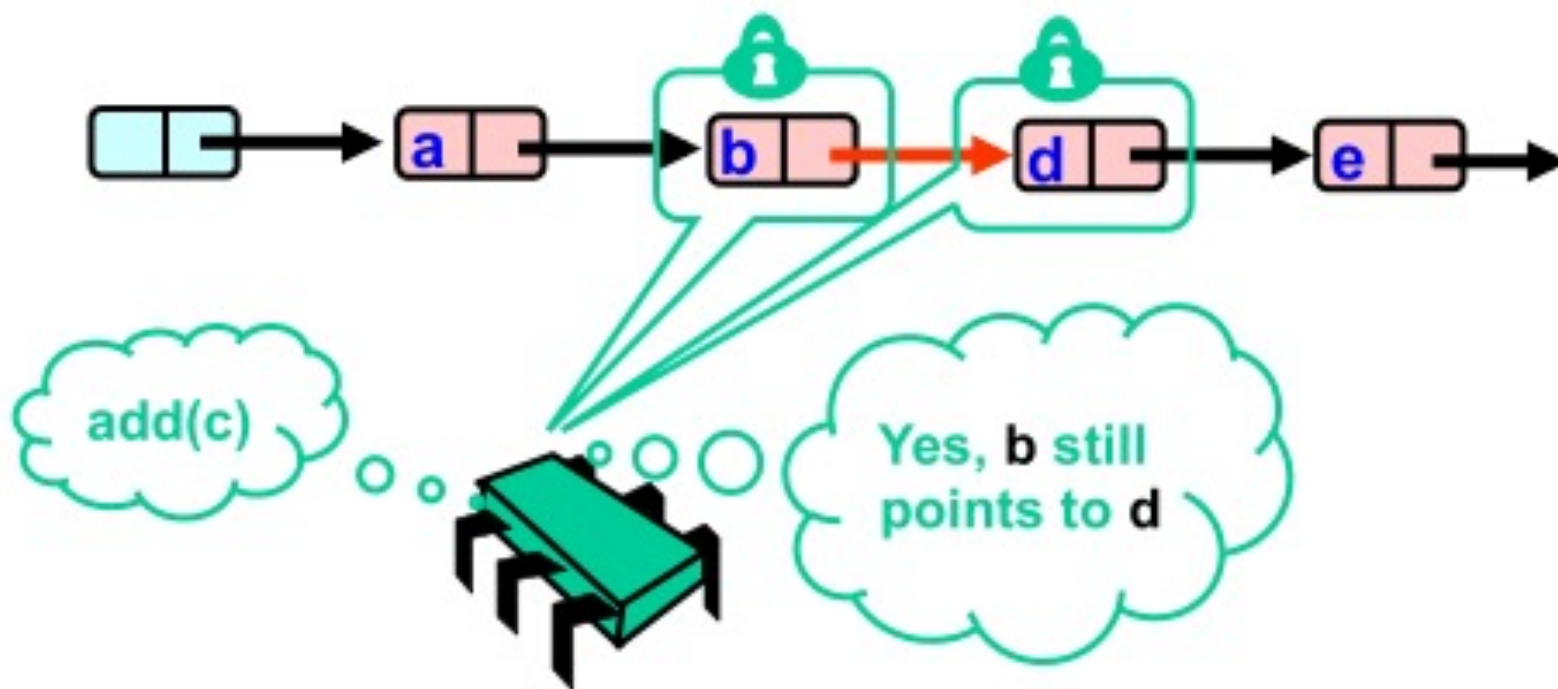


无锁遍历异常(二)

- 丢失更新



验证b和d之间仍然可达



验证代码

```
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;
}
```


Add代码，无锁遍历

```
while(true){
    pred = head;
    curr = pred.next;

    while(curr.key < key){
        pred = curr;
        curr = curr.next;
    }
    pred.lock();
    curr.lock();
    try{
        if(validate(pred, curr)){
            if(curr.key == key){
                return false;
            }
            NodeWithLock<T> node = new NodeWithLock<T>(item);
            node.next = curr;
            pred.next = node;
            return true;
        }
    }finally{
        pred.unlock();
        curr.unlock();
    }
}
```

Contains代码

```
while(true){
    pred = head;
    curr = pred.next;

    while(curr.key < key){
        pred = curr;
        curr = curr.next;
    }
    pred.lock();
    curr.lock();
    try{
        if(validate(pred, curr)){
            return curr.key == key;
        }
    }finally{
        pred.unlock();
        curr.unlock();
    }
}
```

讨论contains是否需要验证？

- contains和remove可能有异常
 - 如果查找元素本身已被删除
 - 可能对一个已不在链表里的节点返回true(和可线性化的点有关，我们认为不允许)
 - A Lazy Concurrent List-Based Set Algorithm
 - 如果查找元素的前驱被删除
 - 链表可能断裂，需要验证

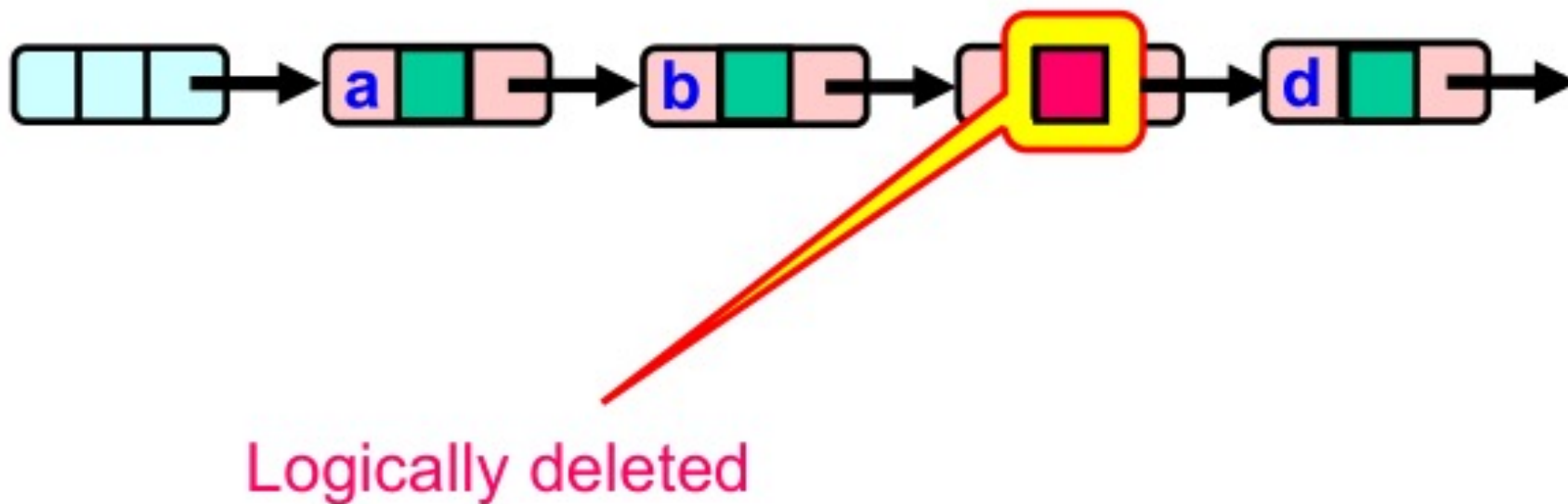
Lazy List

A Lazy Concurrent List-Based Set Algorithm

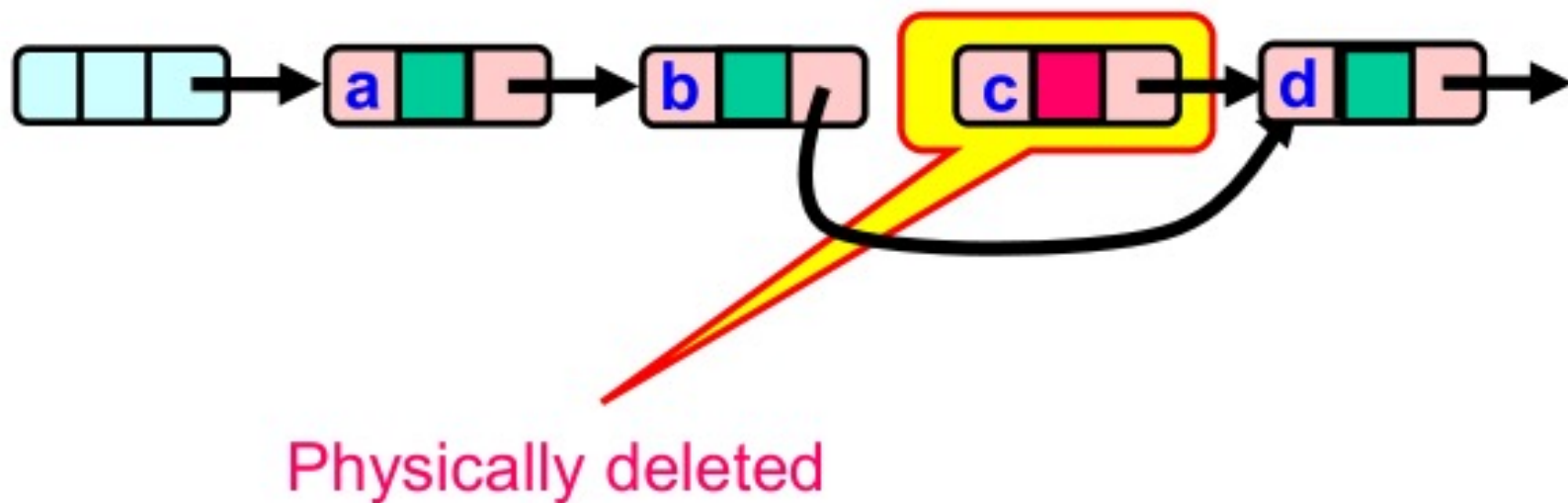
Lazy List

- 乐观锁
 - 仍然需要加锁
 - 可能需要失败重试
- Lazy List
 - 使用标记删除
 - Contains操作无需锁

标记删除



物理删除



验证

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

验证pred和cur的路径是否可达

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  }
```