# Multicore Computing
# Lecture13 – Lock-Free List

SUNG KYUN KWAN
UNIVERSITY

남 범 석

bnam@skku.edu

# Desynchronization

- Problems with Locking
  - Deadlock
  - Priority Inversion
    - Low-priority processes hold a lock required by a higher priority process
  - Convoying
    - All the other processes slow to the speed of the slowest one
  - Async-signal safety
    - Suppose a thread receives a signal while holding a user level lock in the memory allocator
  - Kill-tolerance
    - threads are killed/crash while holding locks
  - Pre-emption tolerance
    - pre-empted while holding a lock
  - Overall performance

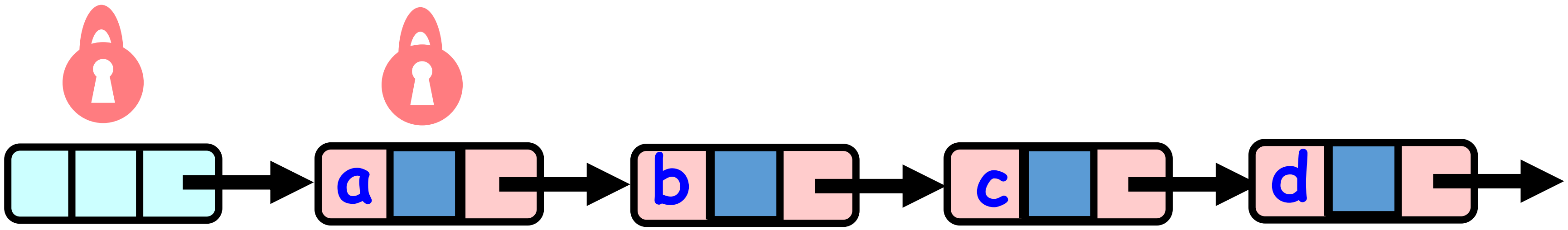
# For Highly-Concurrent Applications

- Designing generalized lock-free algorithms is hard
- Design highly concurrent data structures instead
  - Buffer, list, stack, queue, map, B-tree, etc
- Four patterns
  1. Fine-grained Synchronization
  2. Optimistic Synchronization
  3. Lazy Synchronization (Wait-Free Search)
  4. Lock-Free Synchronization

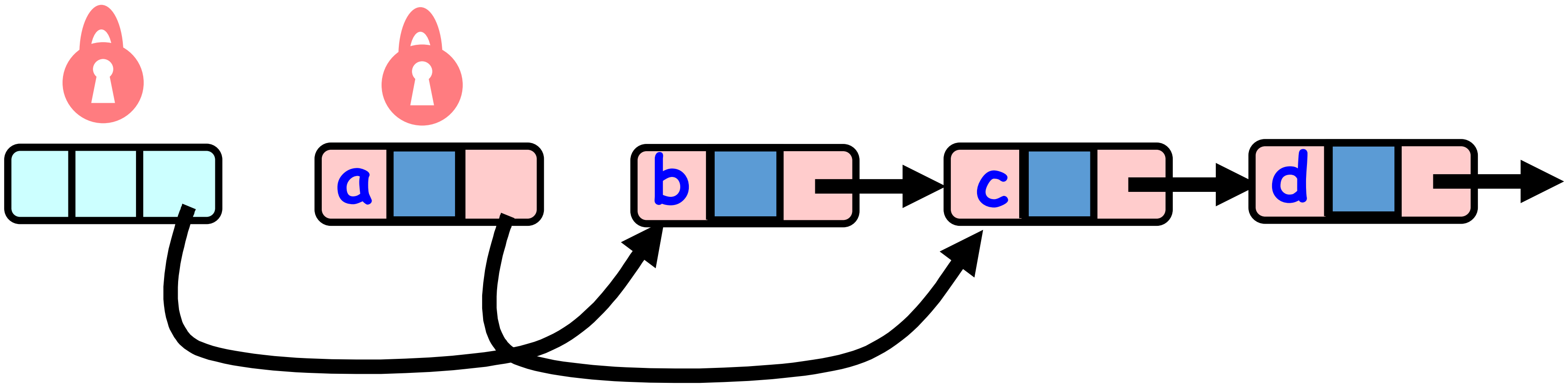
# 1. Fine-Grained Synchronization

- Instead of using a single global lock …
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component …
  - At the same time

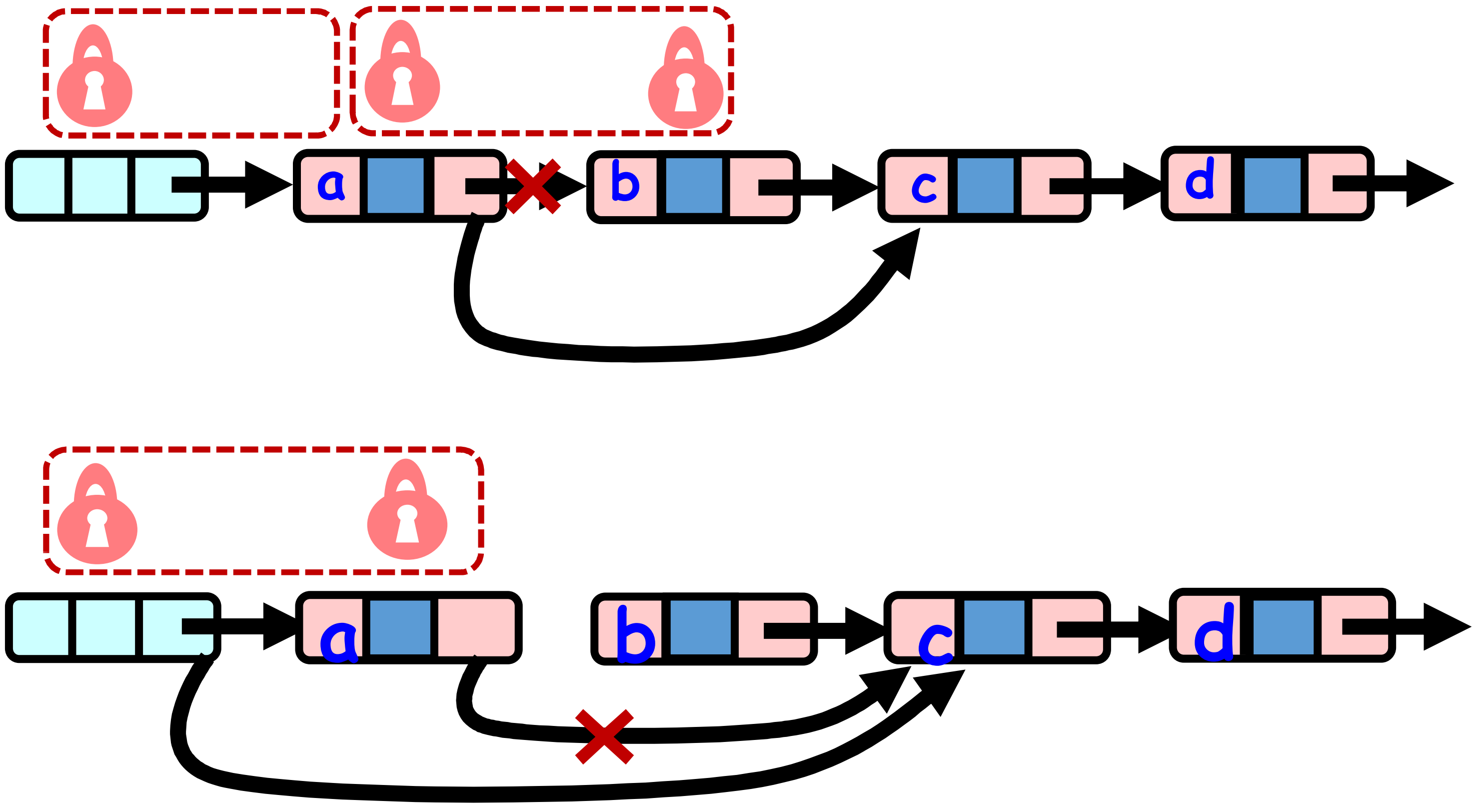
- Need two locks for linked list
  - Suppose thread A is about to remove *a* and another thread B is about to remove *b*.
  - Thread A locks head and thread B locks a.
  - Thread A then sets head to b while B sets a's next to c.
  - We fail to remove node b.



- Need two locks for linked list
  - Suppose thread A is about to remove *a* and another thread B is about to remove *b*.
  - Thread A locks head and thread B locks a.
  - Thread A then sets head to b while B sets a's next to c.
  - We fail to remove node b.



- Need two locks for linked list
  - **Hand-over-hand locking** ensures that concurrent removals conflict.
  - Thread A and B are serialized.



# 2. Optimistic Synchronization

- Lock-free traversal. If you find it, lock and check …
  - OK: we are done
  - If wrong: start over
- Evaluation
  - Usually cheaper than locking, but
  - Validation is expensive, i.e., we traverse the list twice.

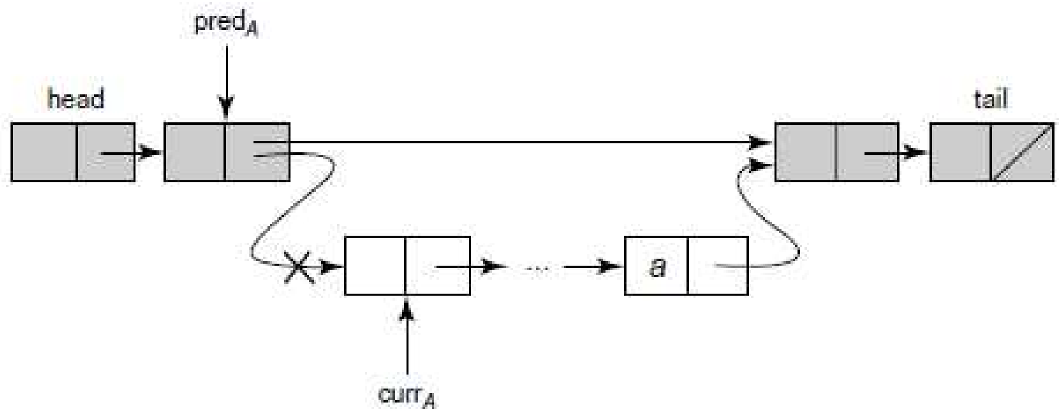
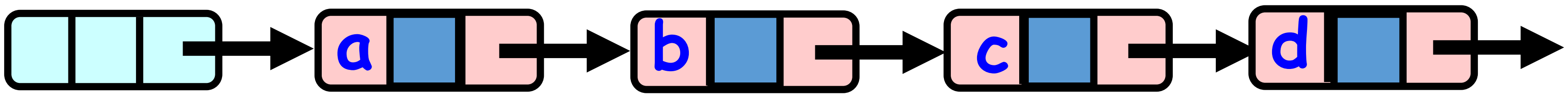


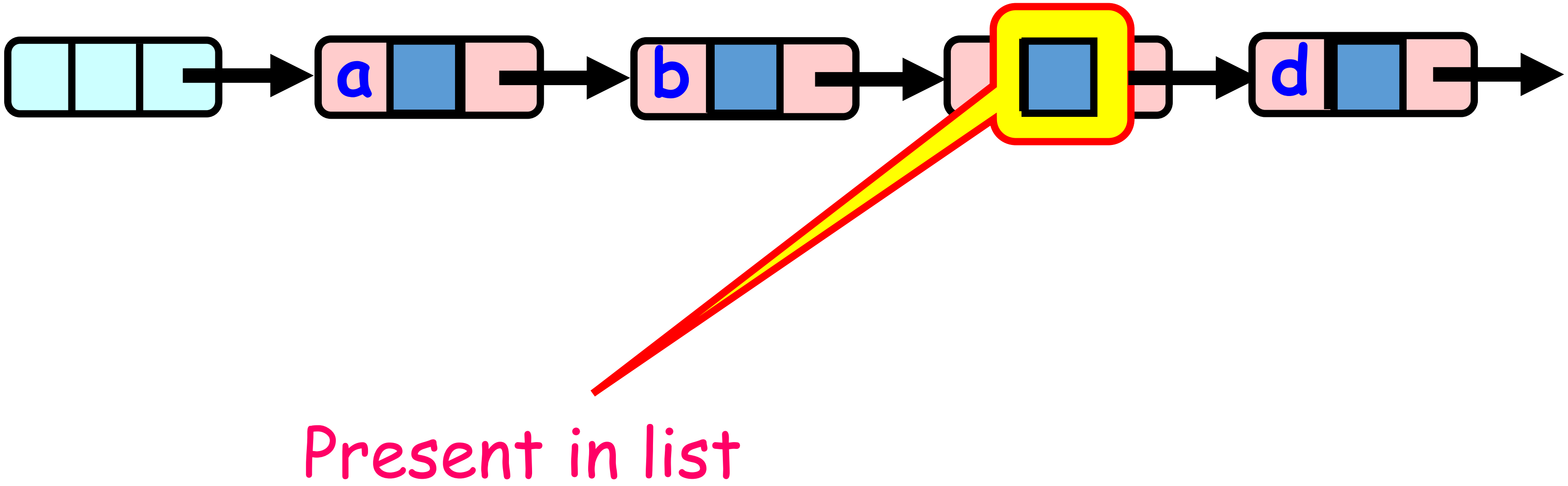

**Figure 9.15** The `OptimisticList` class: why validation is needed. Thread $A$ is attempting to remove a node $a$. While traversing the list, $\texttt{curr}_A$ and all nodes between $\texttt{curr}_A$ and $a$ (including $a$) might be removed (denoted by a lighter node color). In such a case, thread $A$ would proceed to the point where $\texttt{curr}_A$ points to $a$, and, without validation, would successfully remove $a$, even though it is no longer in the list. Validation is required to determine that $a$ is no longer reachable from `head`.

- Lock-free search, but insertion/removal locks
- But postpone hard work
  - Validation does not traverse the entire list
- Removing nodes causes trouble, so let's do it "lazily"
  - remove()
    - Scans once
    - Locks predecessor & current
  - Logical removal
    - Mark current node as deleted
  - Physical removal
    - Do what needs to be done.
    - E.g) redirect predecessor's next

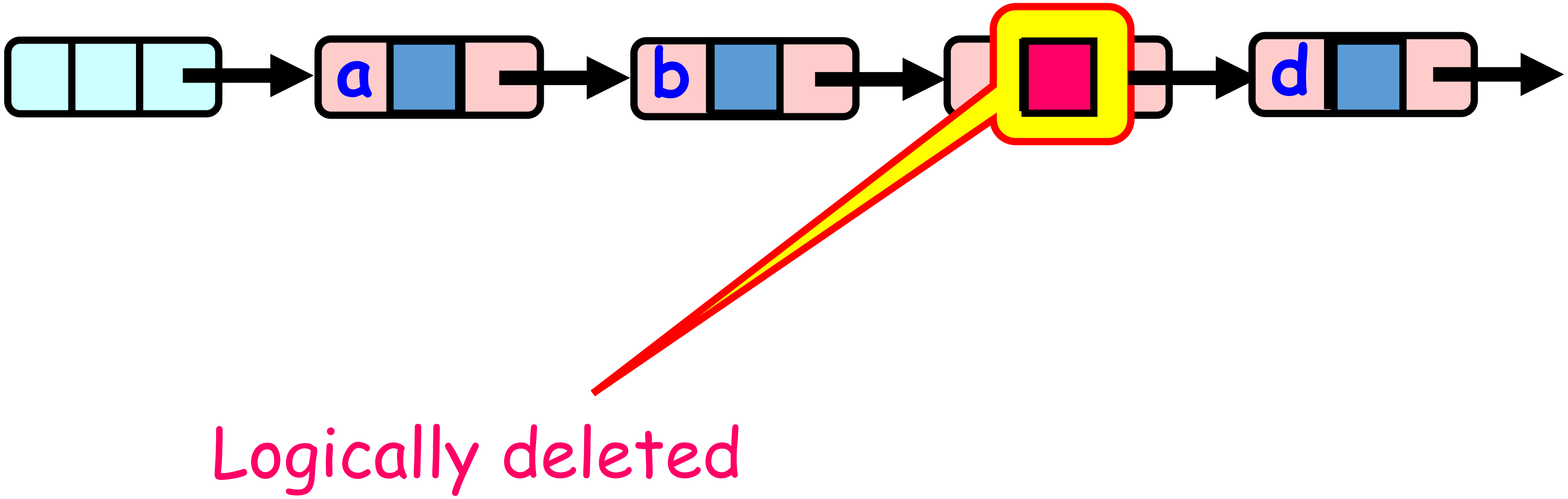
# 3. Lazy Synchronization: Lazy Removal

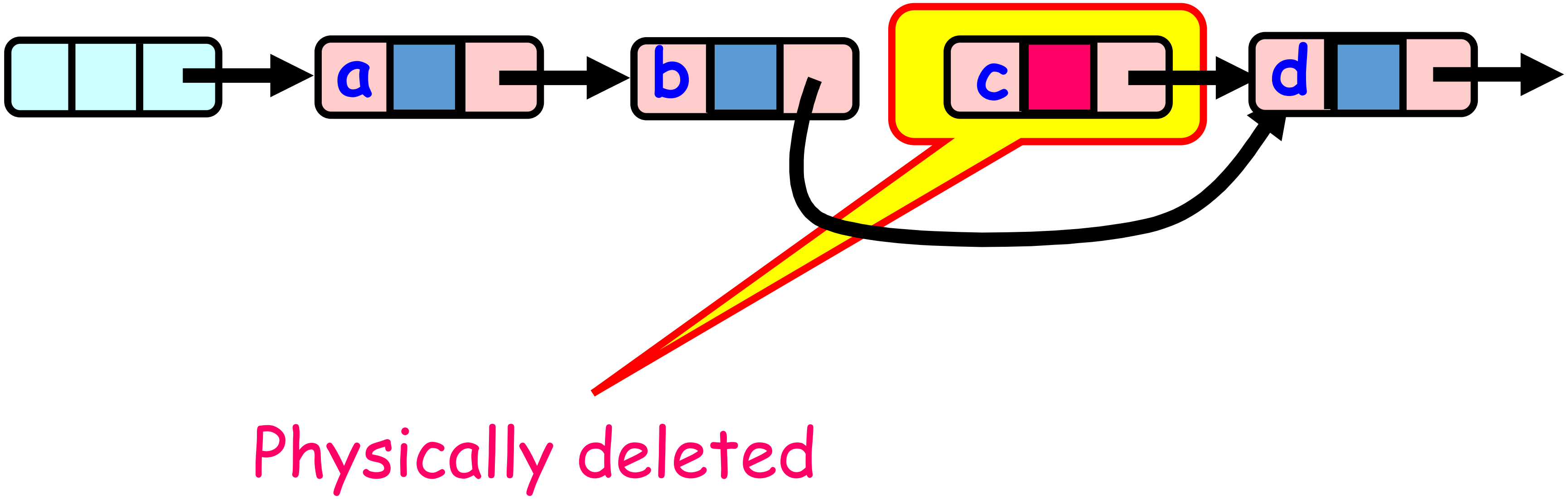


# 3. Lazy Synchronization: Lazy Removal

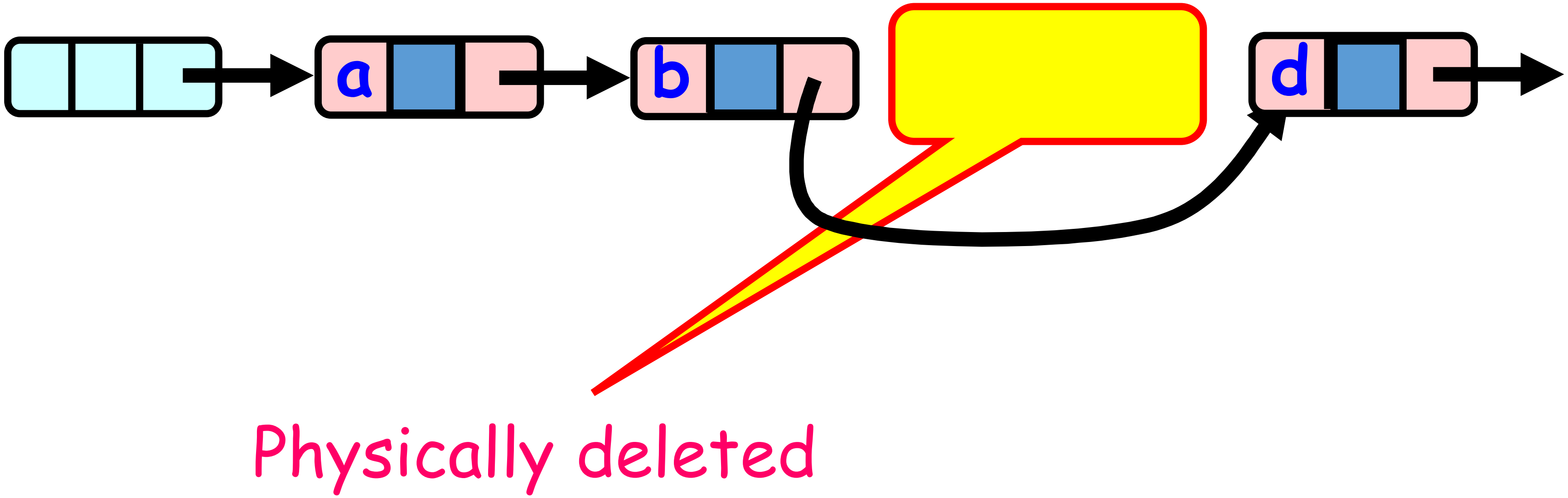


# 3. Lazy Synchronization: Lazy Removal




a
b
c
d
Physically deleted

# 3. Lazy Synchronization: Lazy Removal

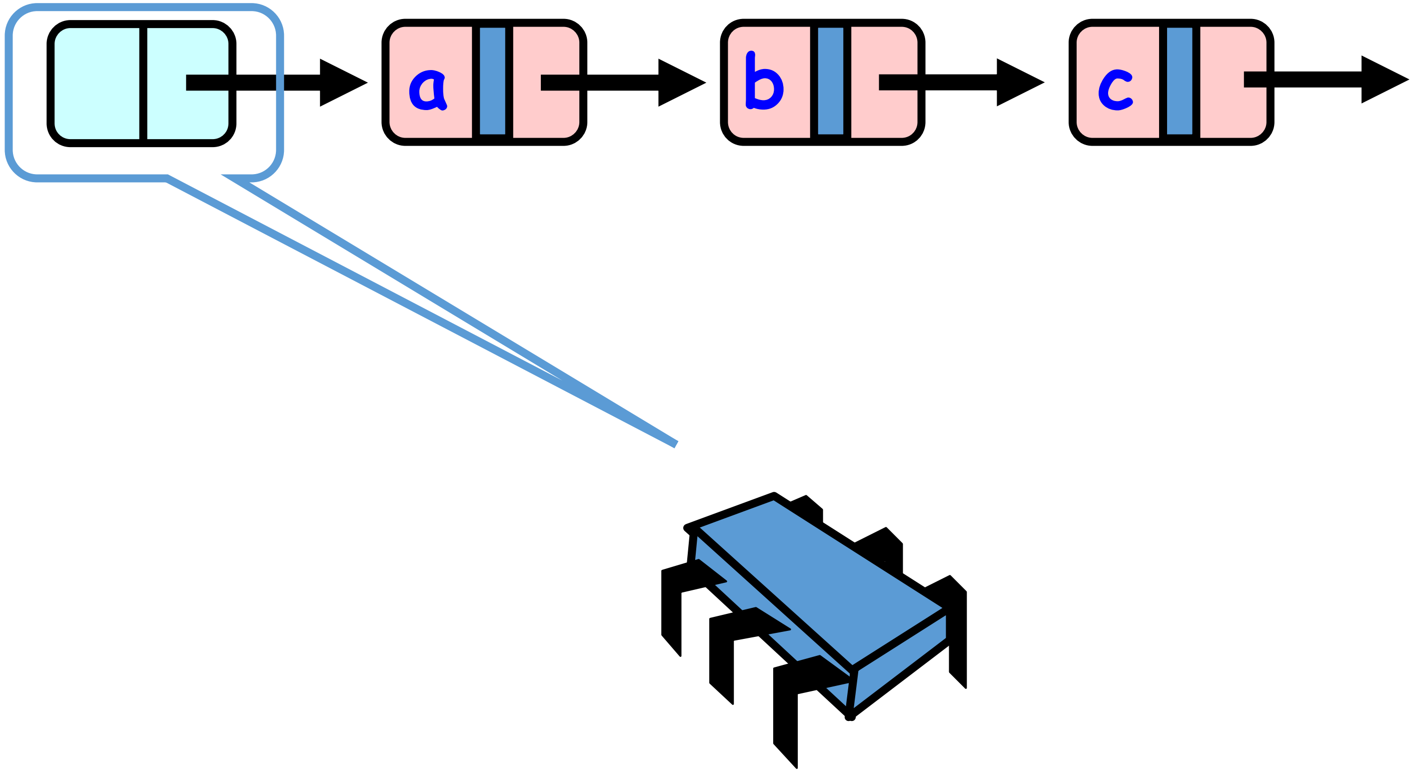


# 3. Lazy Synchronization: Lazy Removal

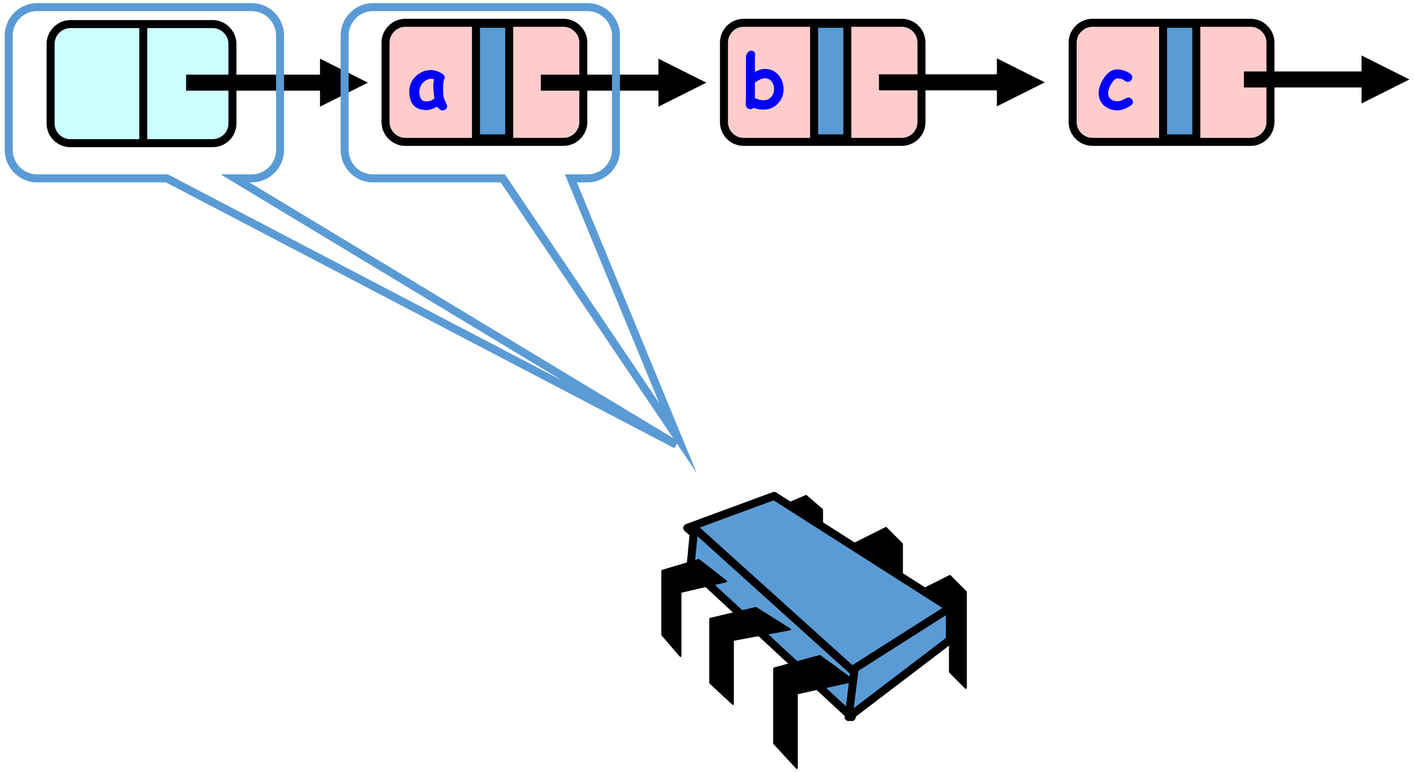
- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
- Write transactions must still lock pred and curr nodes.

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

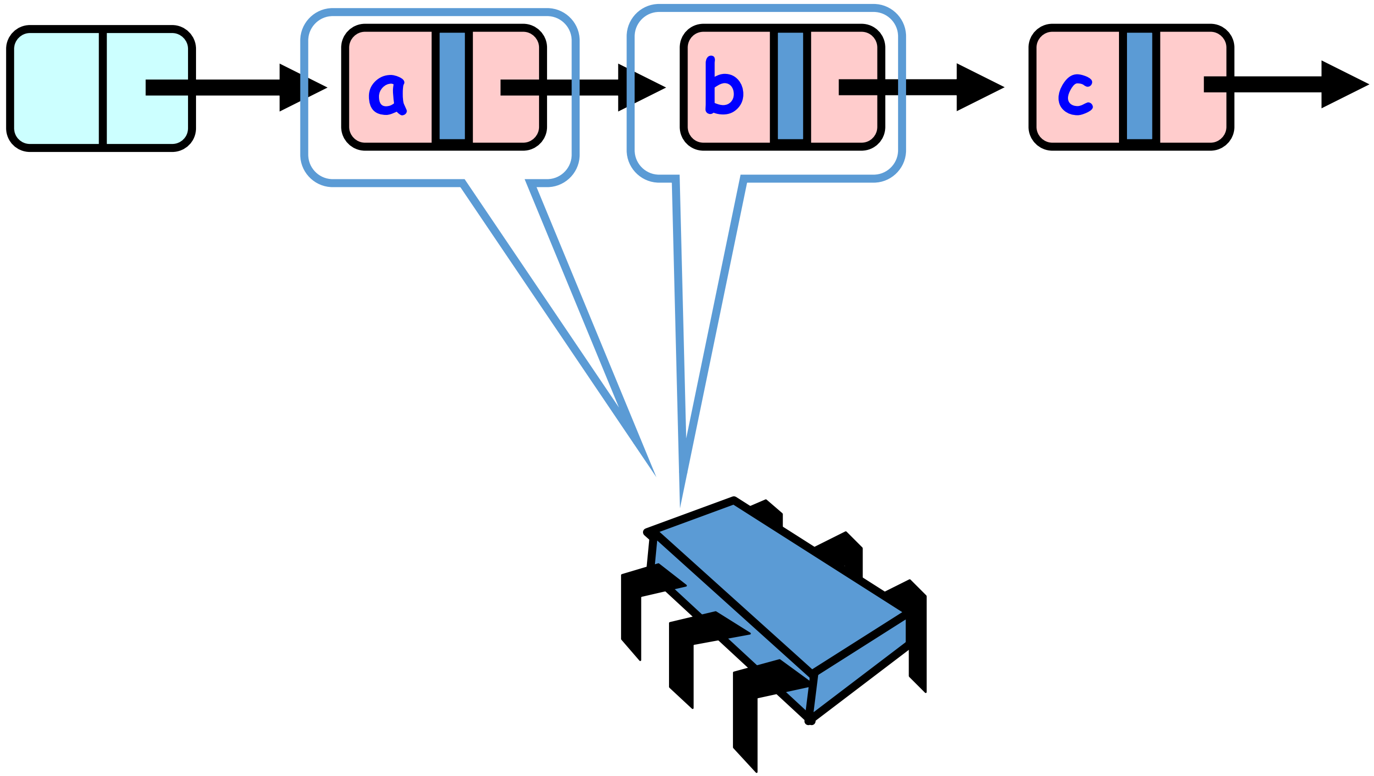
- Thread blue searches for b.



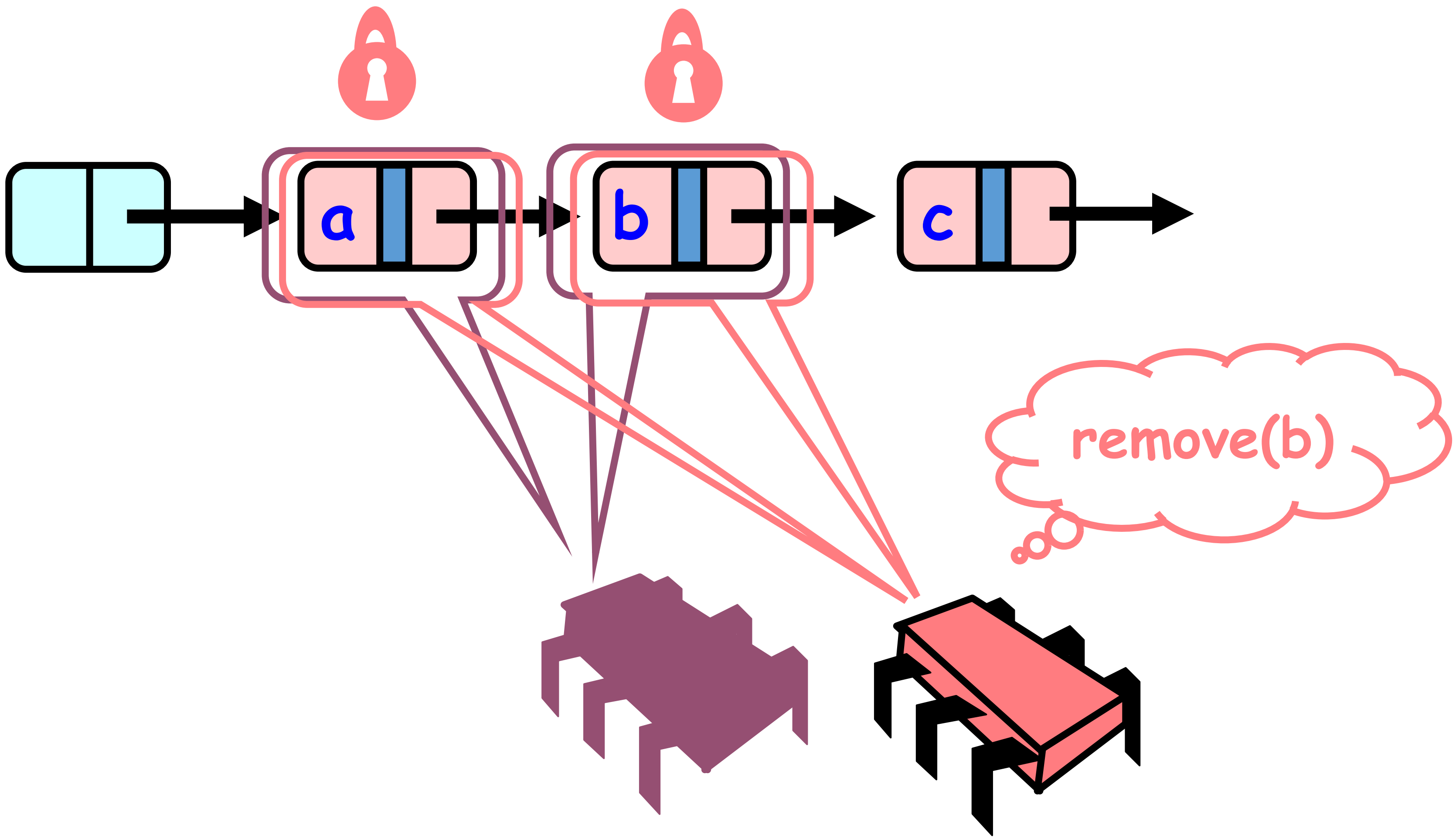
- Thread A searches for b.
  - $pred_A$ = head
  - $curr_A$ = a



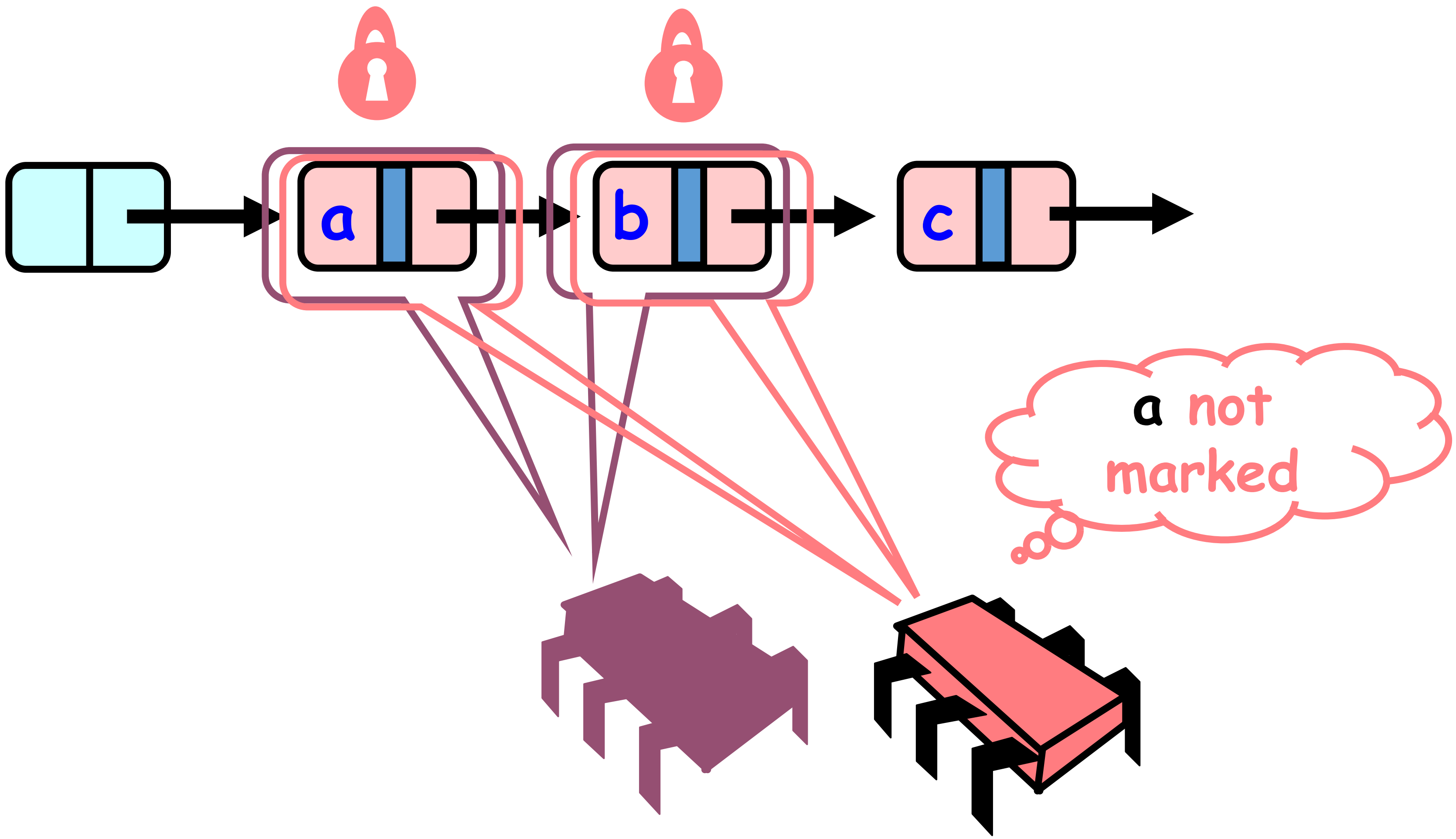
- Thread A goes to sleep before reading b.
  - $pred_A$ = a
  - $curr_A$ = b



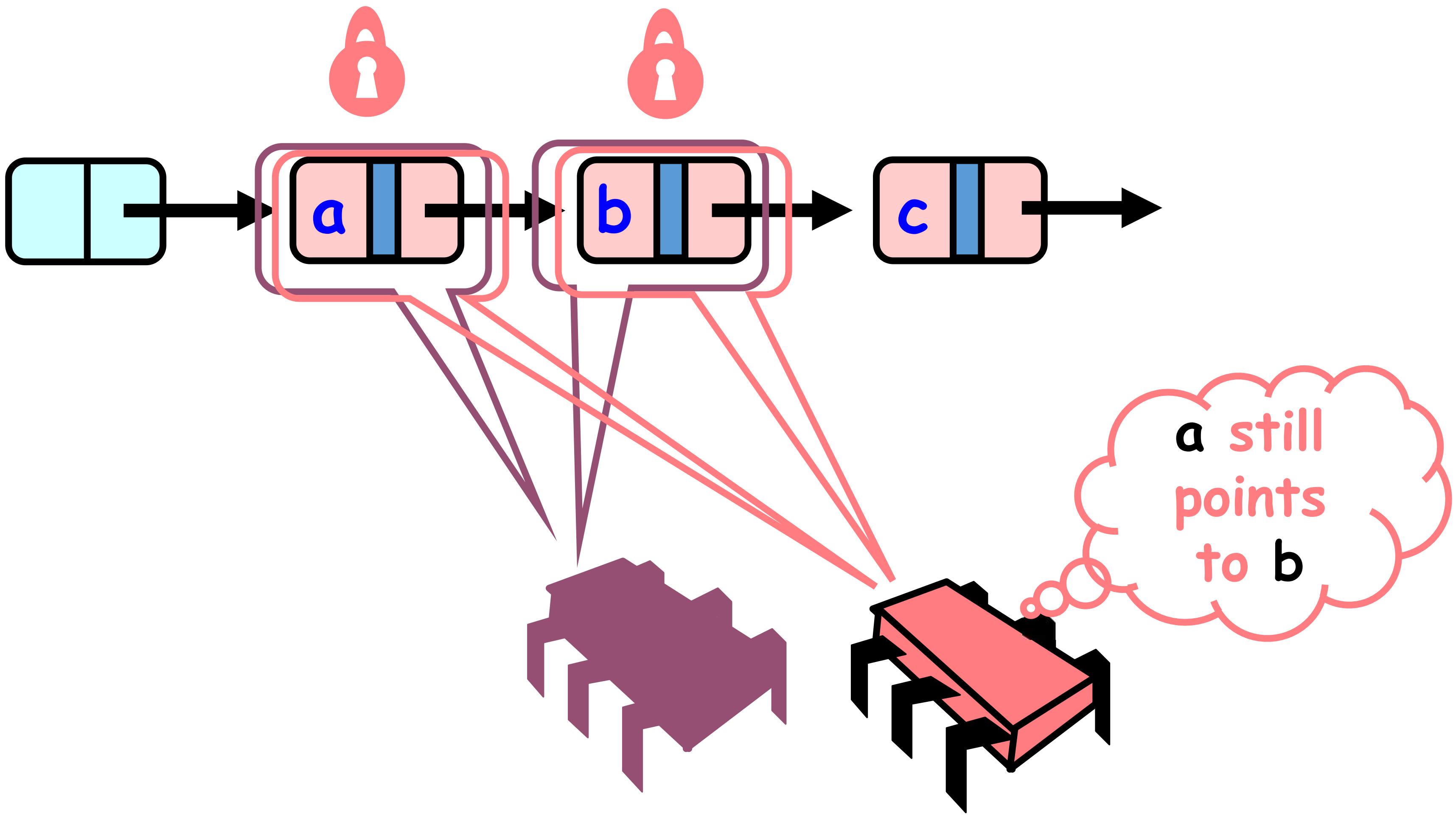
- Thread B removes node b



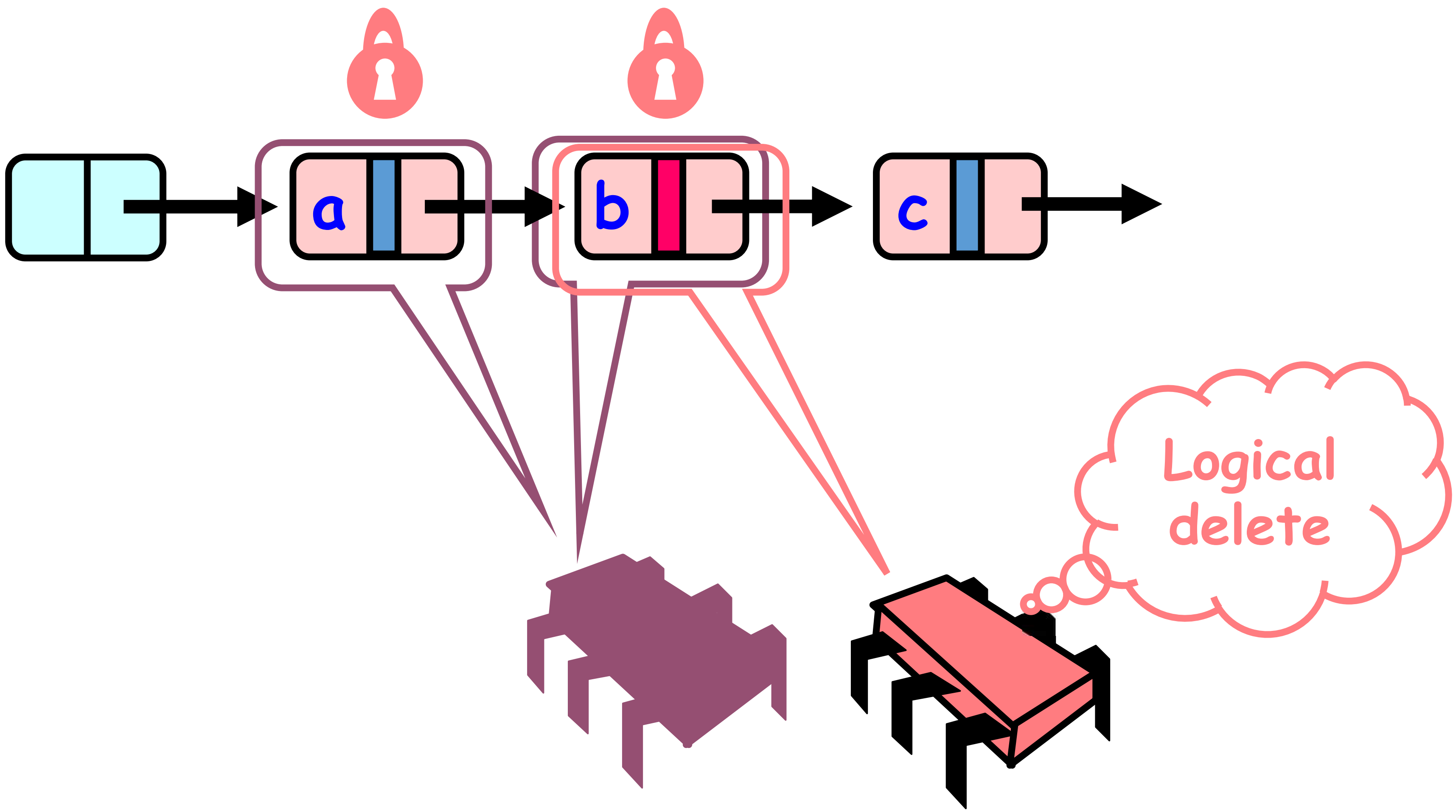
- Thread B removes node b



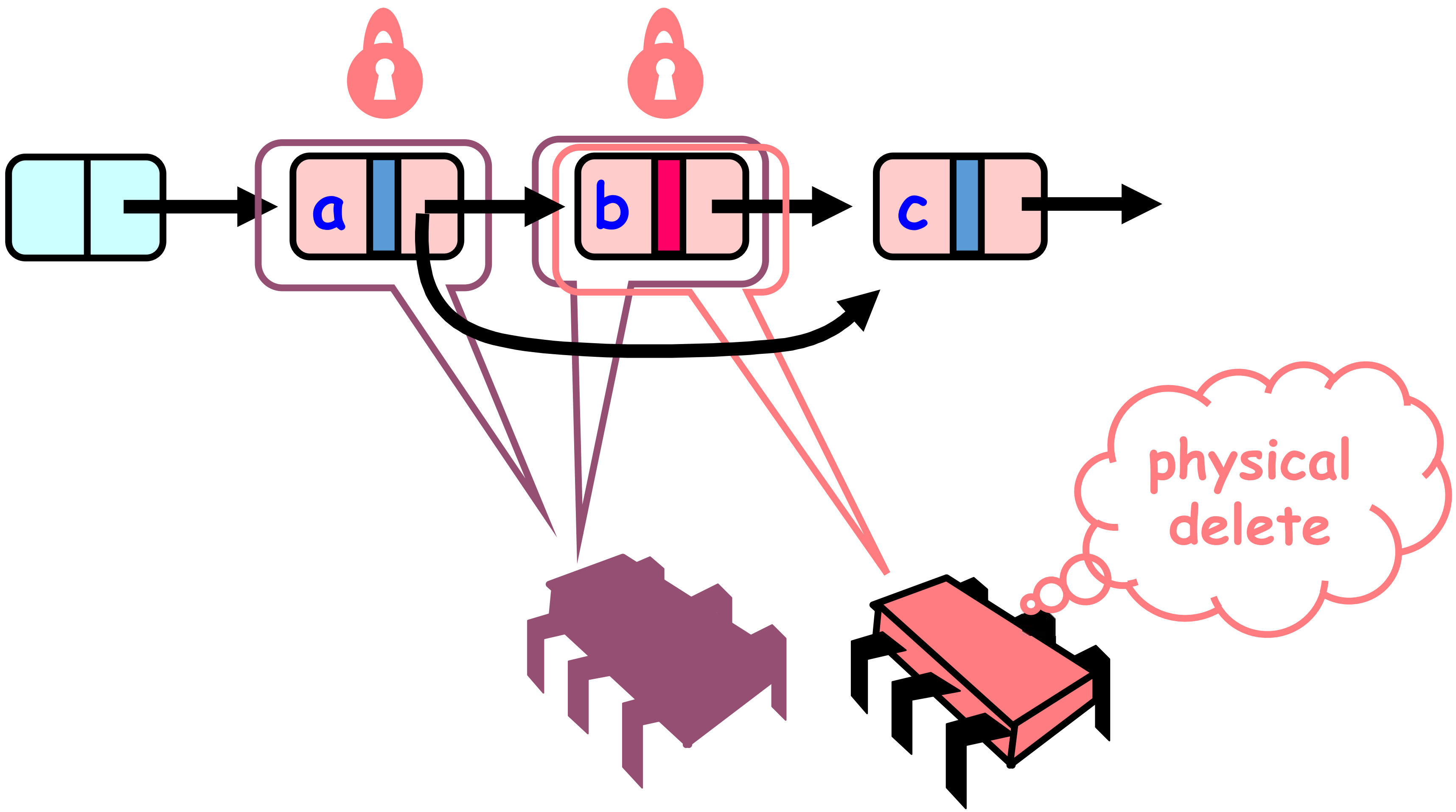
- Thread B removes node b
  - Validation confirms a.next = b



- Thread B logically removes node b

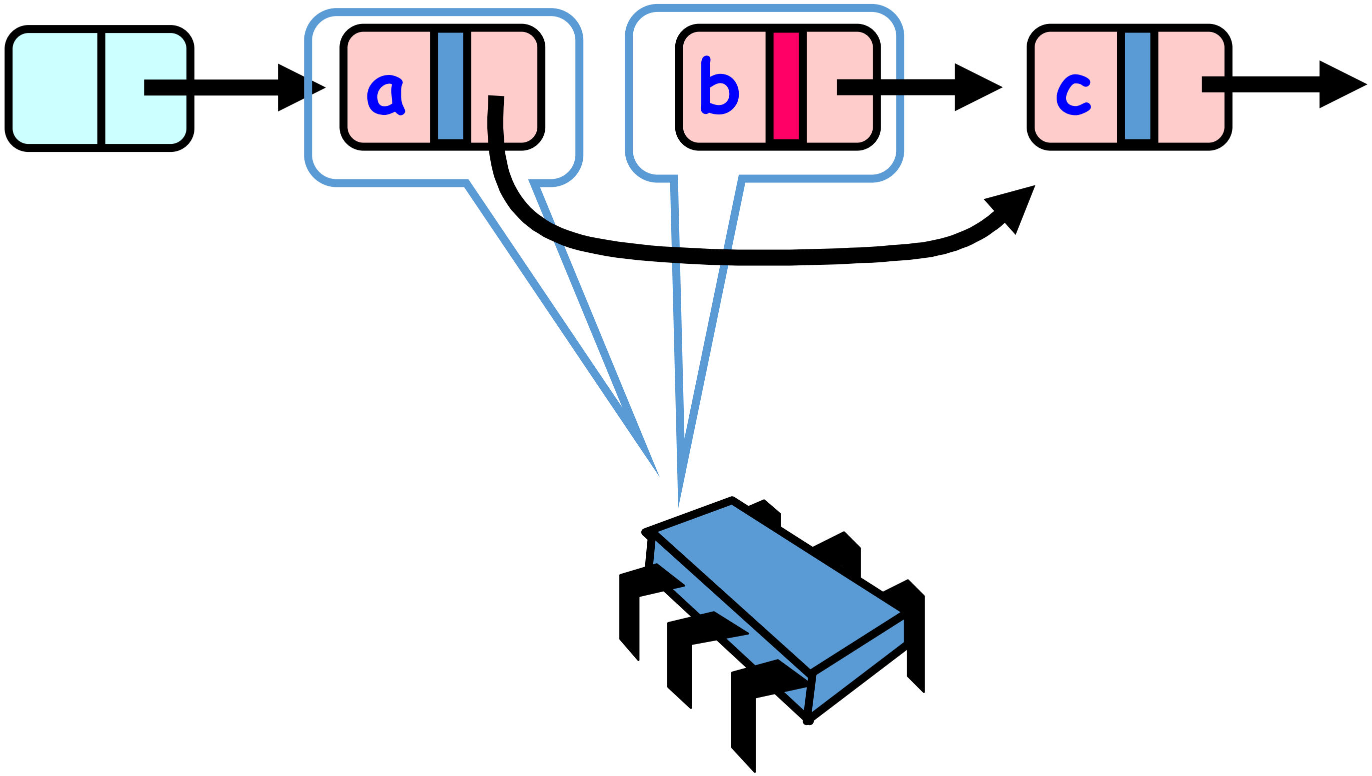


- Thread B physically removes node b



# 3. Lazy Synchronization: Concurrent Accesses

- Thread A will find out
  - either a or b is marked deleted
  - Or a is not pointing to b any more.

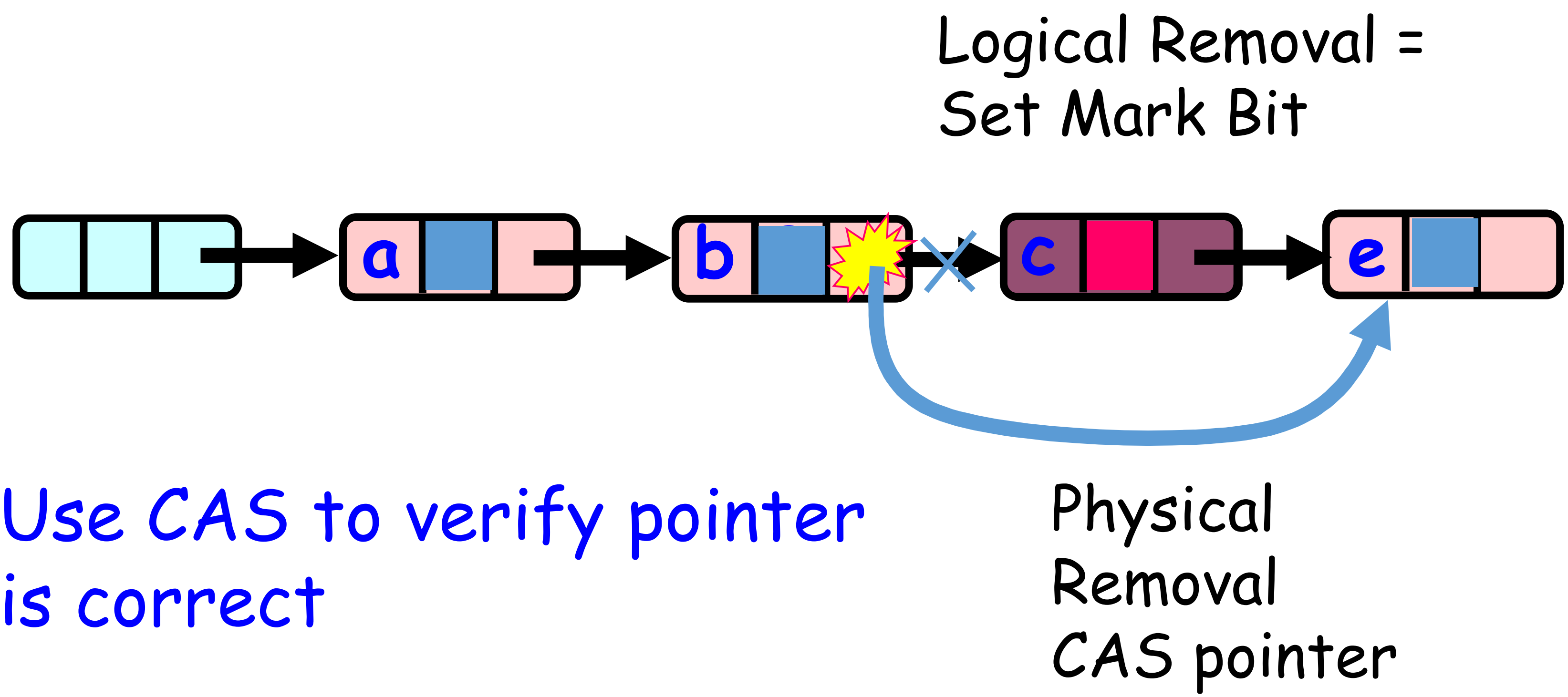


- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

# Lock-Free Lists

- Next logical step
  - Wait-free searches
  - lock-free insertions/deletions
- Use only compareAndSet() (compareAndSwap())

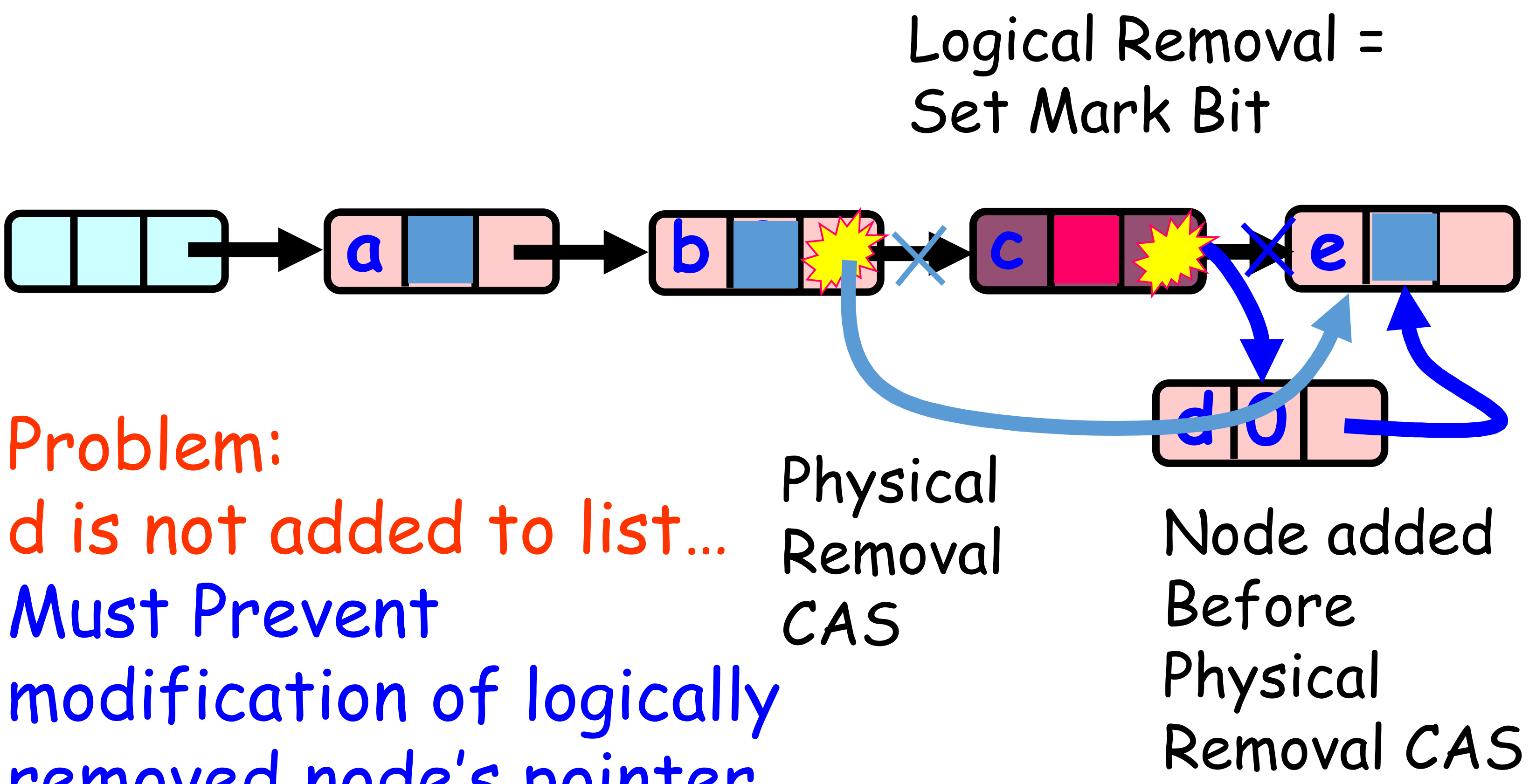
- Verify pointer with CAS




Use CAS to verify pointer is correct

Not enough!

- Problem

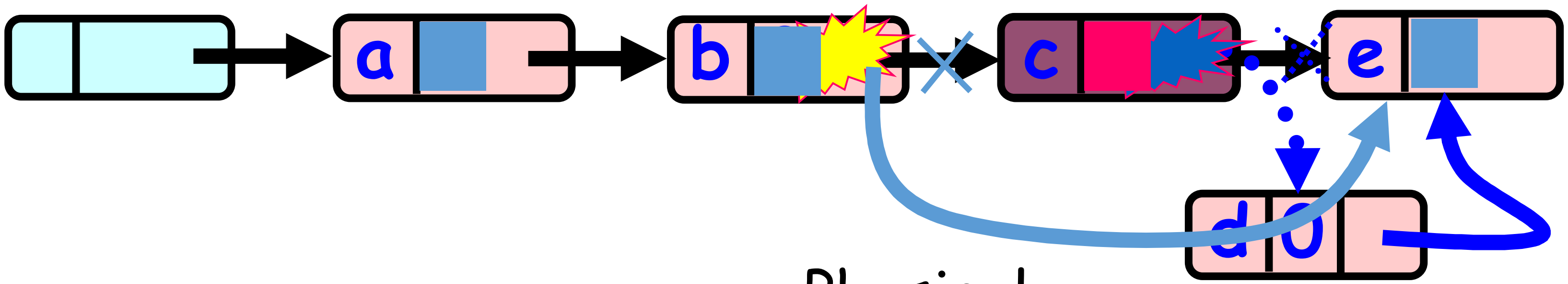



Problem:
d is not added to list…
Must Prevent
modification of logically
removed node's pointer

# Lock-Free Lists: Remove Using CAS

- Solution
  - Combine Bit and Pointer

Logical Removal =
Set Mark Bit

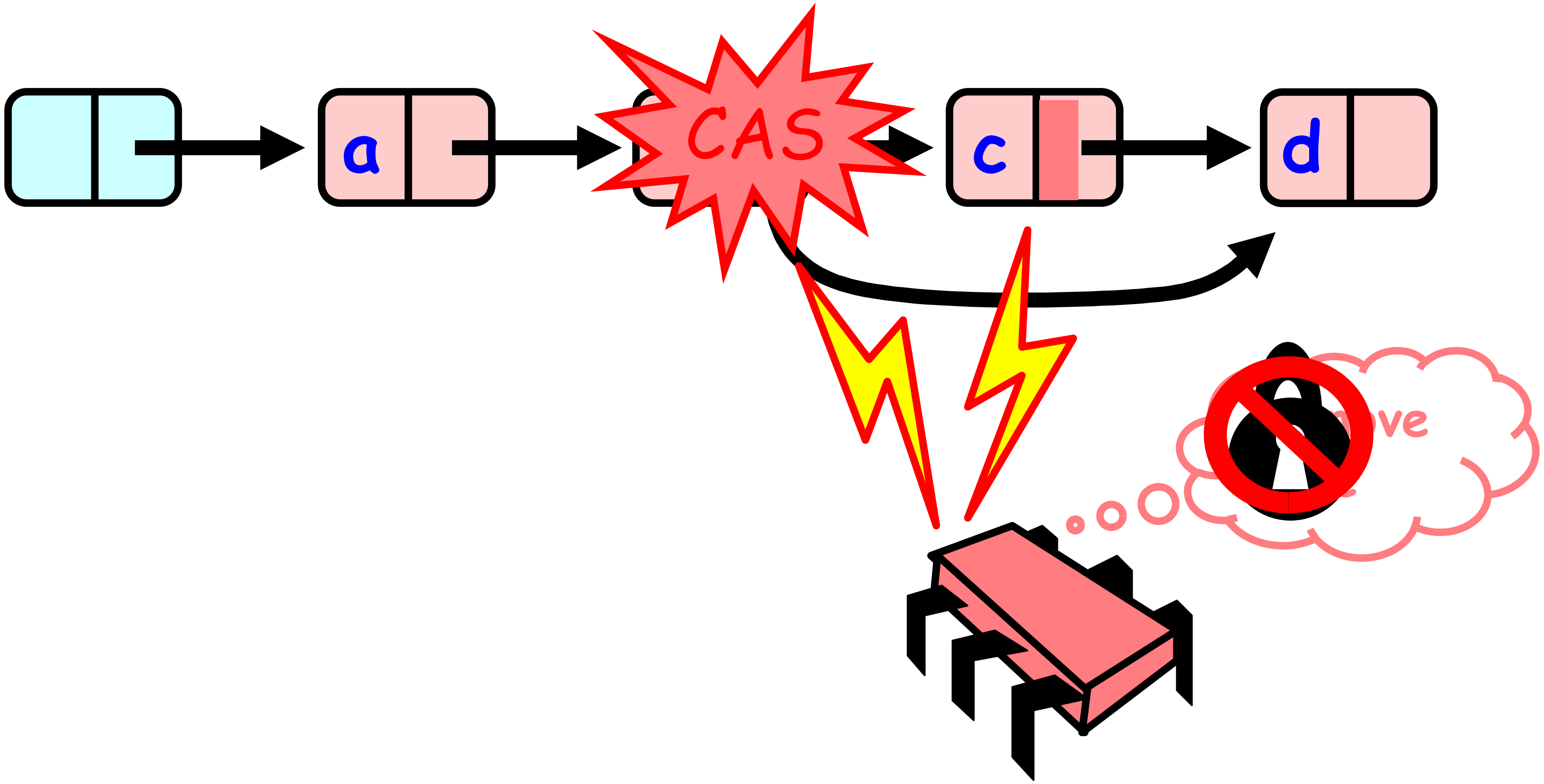


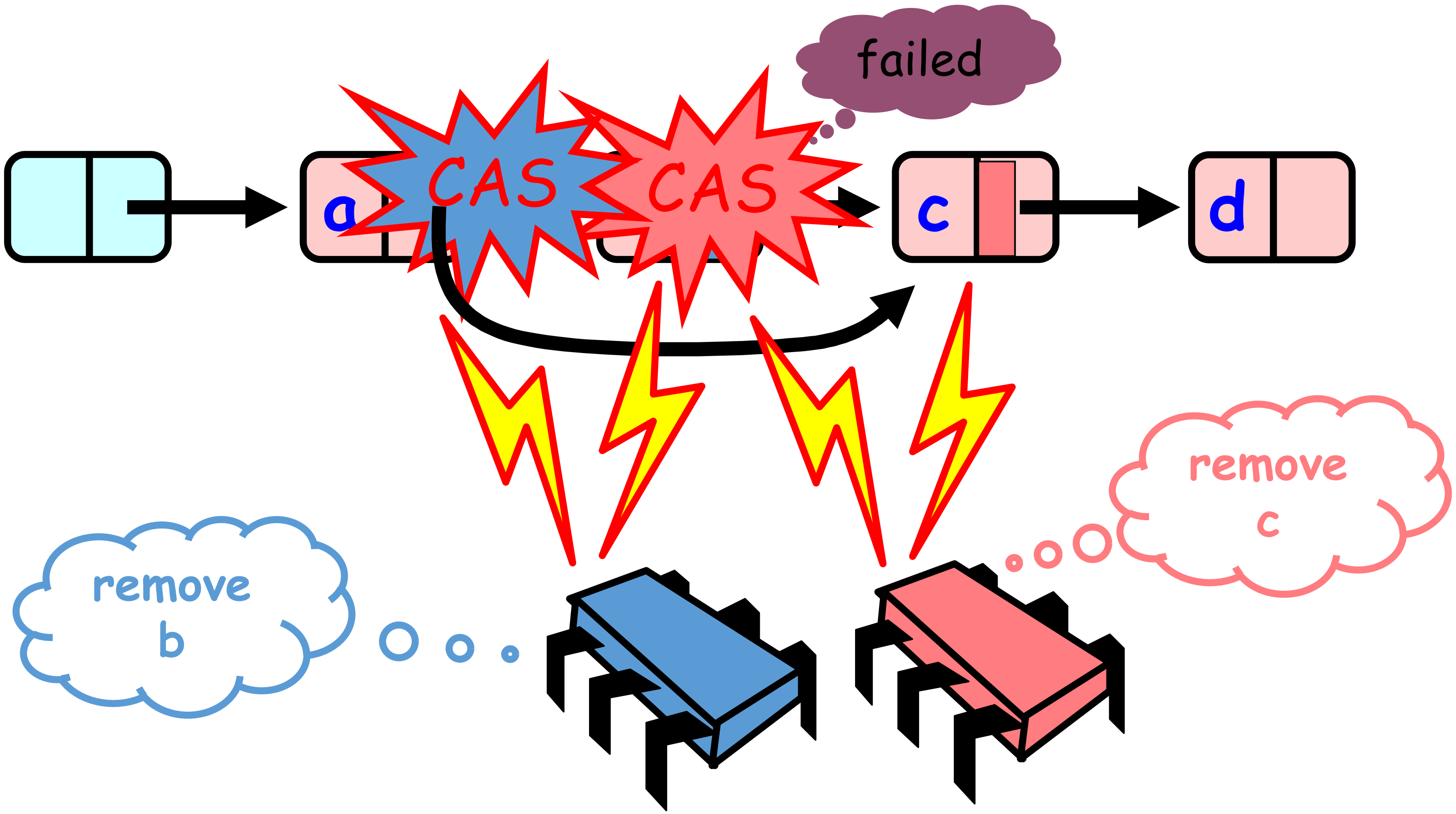

Physical
Removal
CAS

Fail CAS: Node not
added after logical
Removal

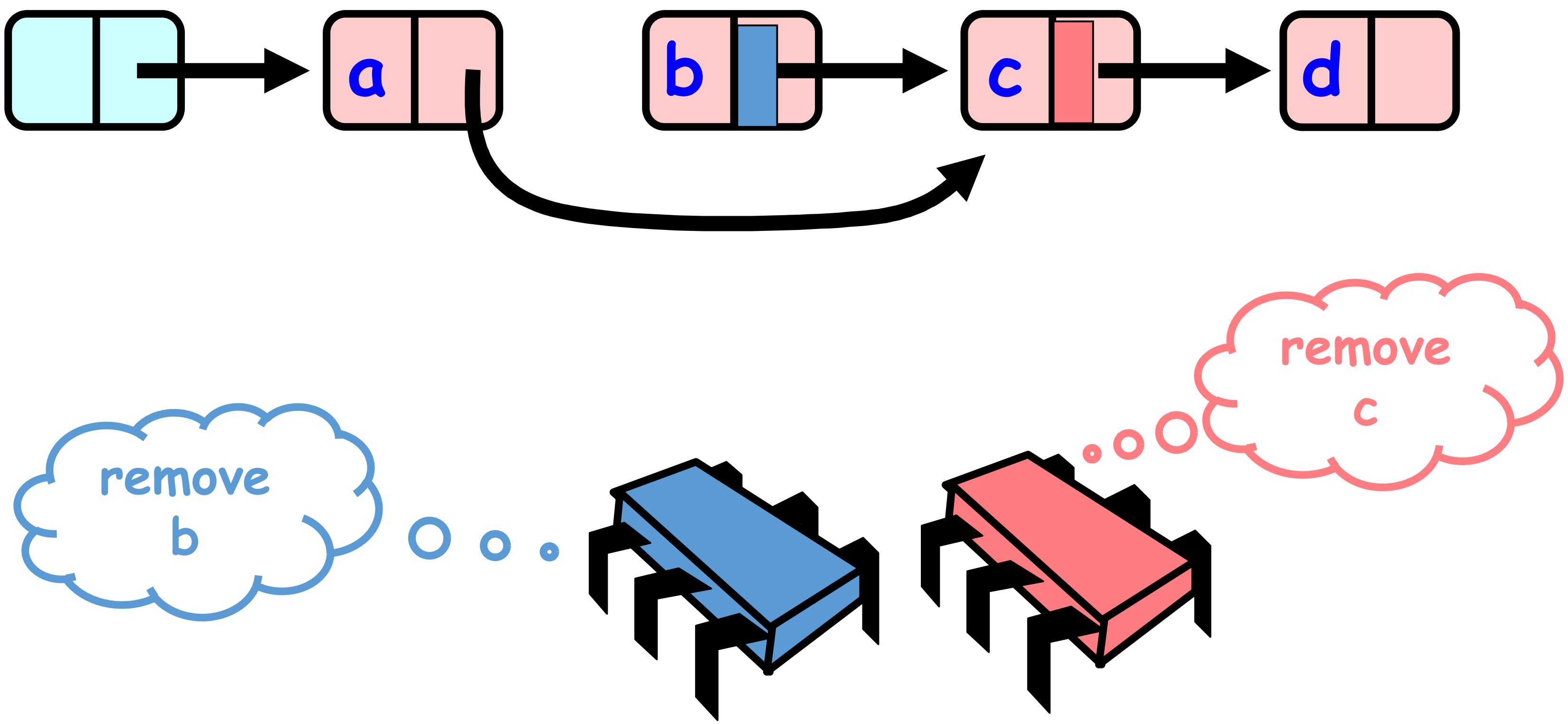
Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)

# Lock-Free Lists: Concurrent Accesses

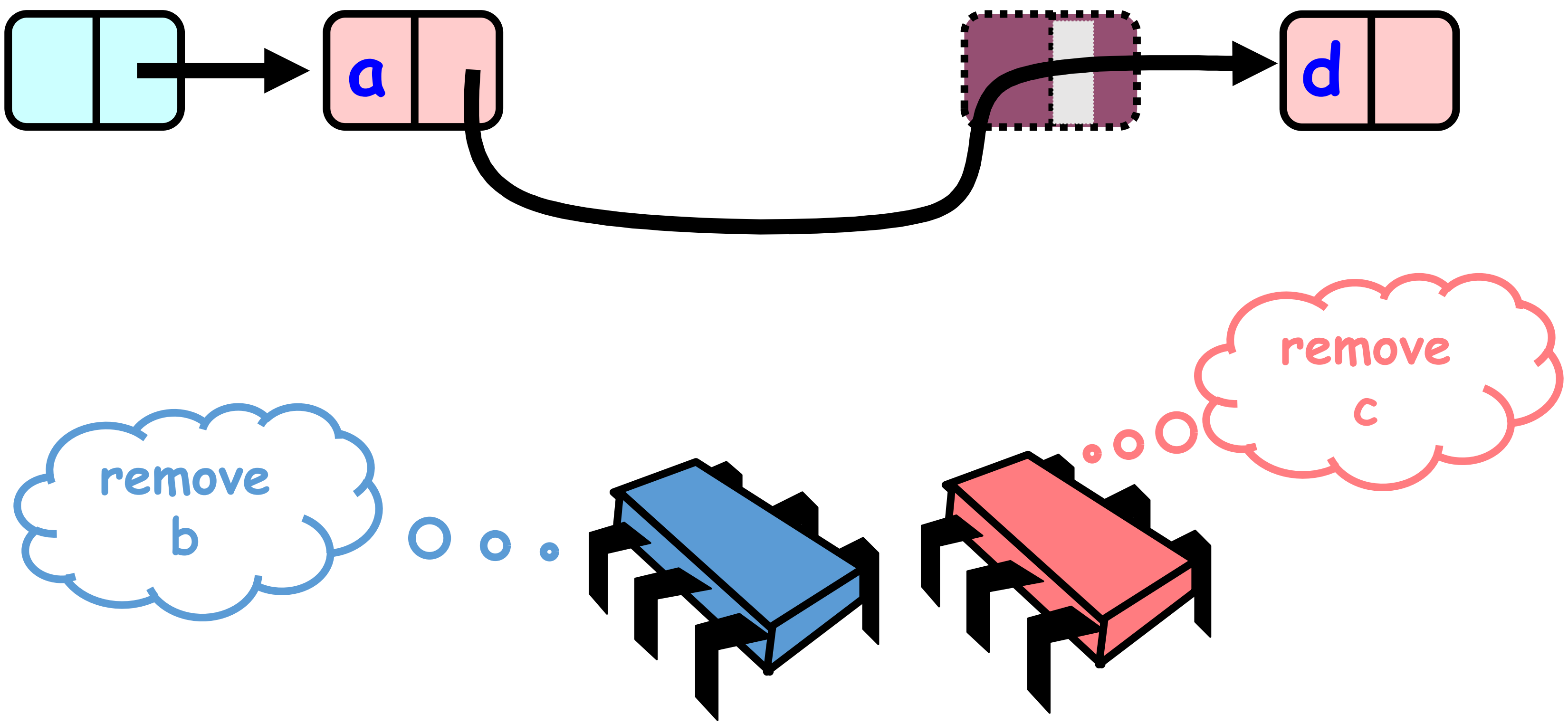



failed
a
CAS
CAS
c
d
remove
b
remove
c

# Lock-Free Lists: Concurrent Accesses



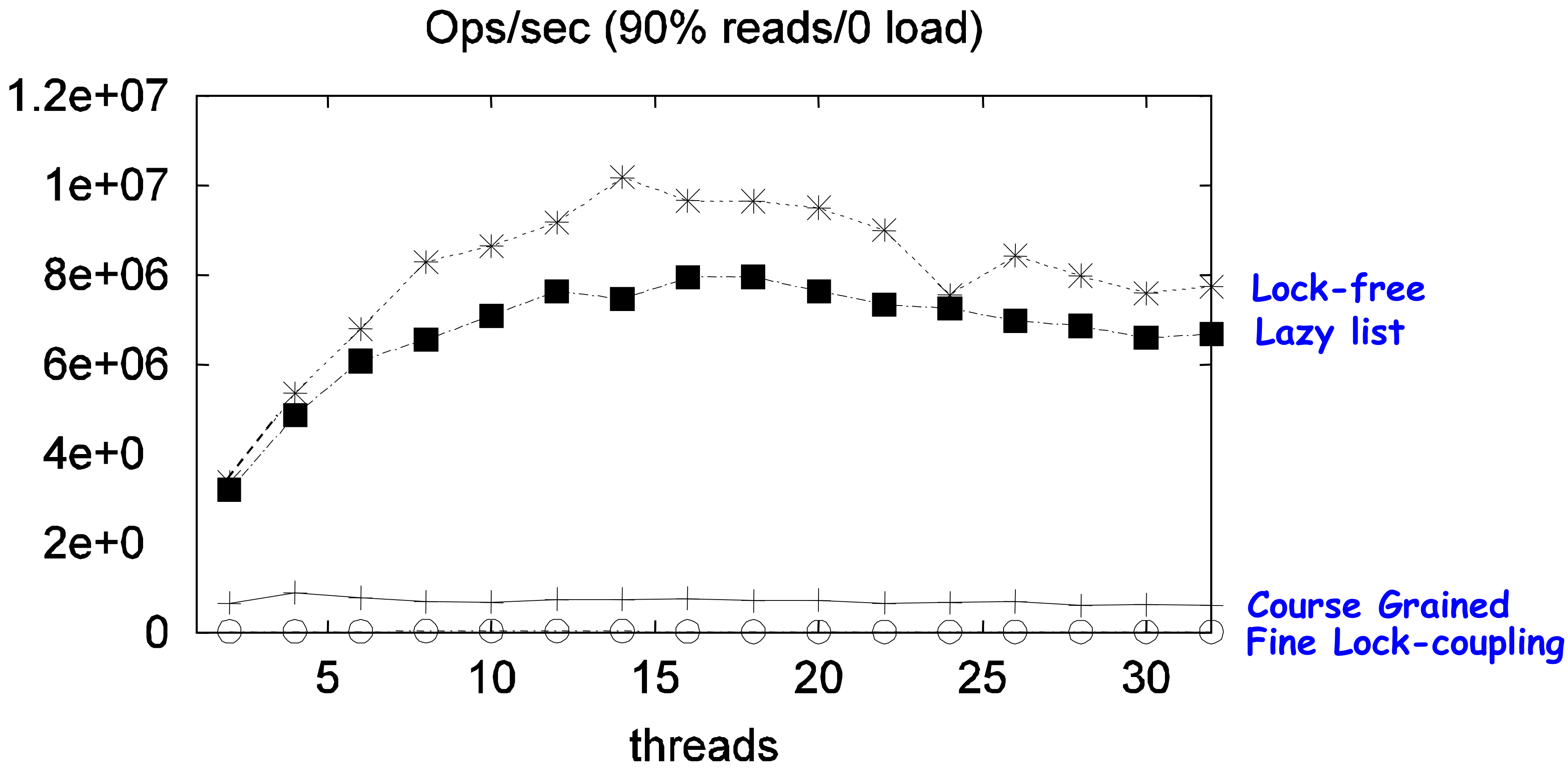
# Lock-Free Lists: Concurrent Accesses



## Performance Comparison

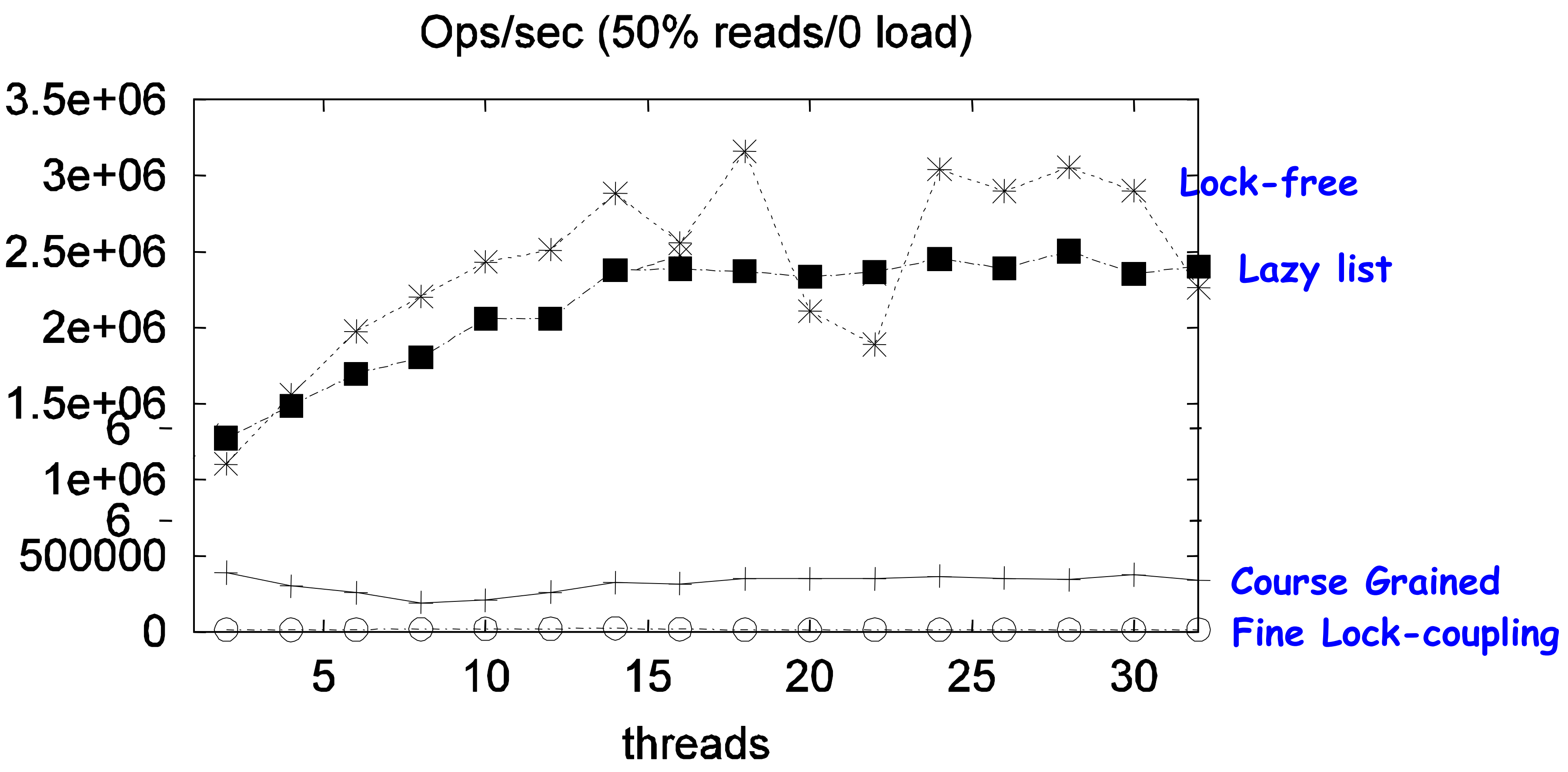
- On 16 node shared memory machine
- Benchmark throughput of Java List-based Set
- Vary % of Contains() method Calls.

- Throughput


Ops/sec (90% reads/0 load)
1.2e+07
1e+07
8e+06
6e+06
4e+0
2e+0
0
5
10
15
20
25
30
threads
Lock-free
Lazy list
Course Grained
Fine Lock-coupling

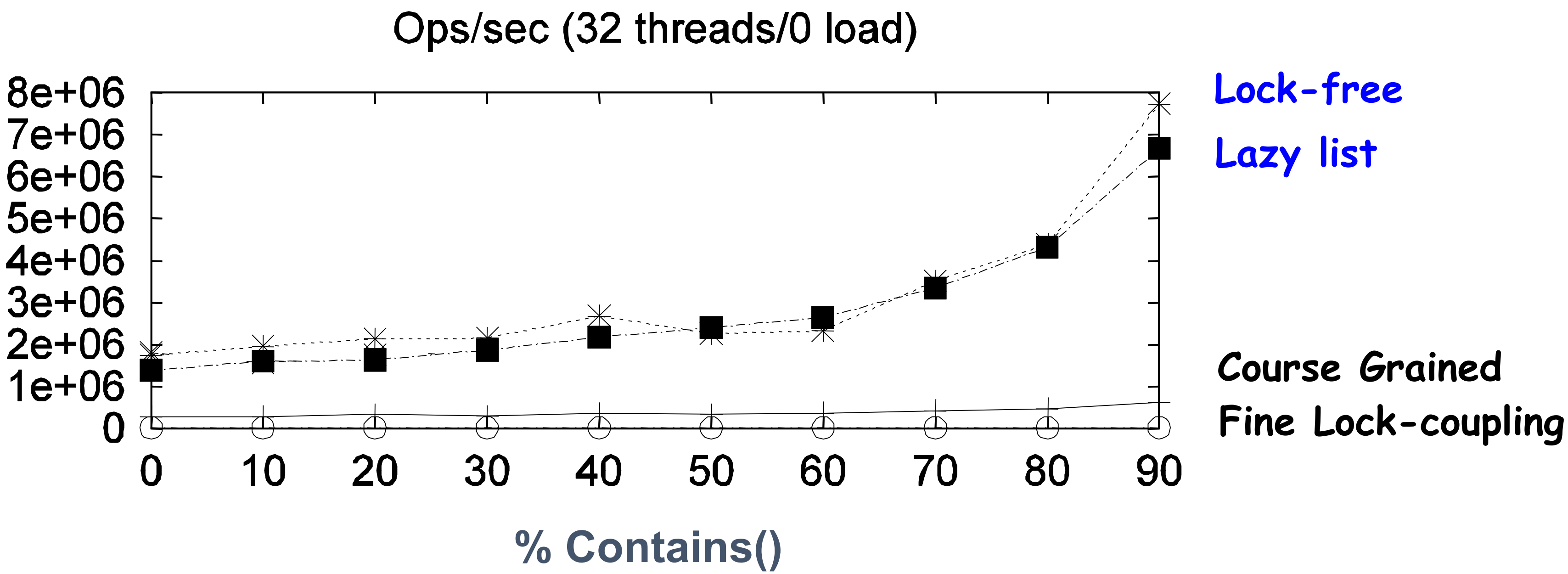
# Low Read Ratio

- Throughput

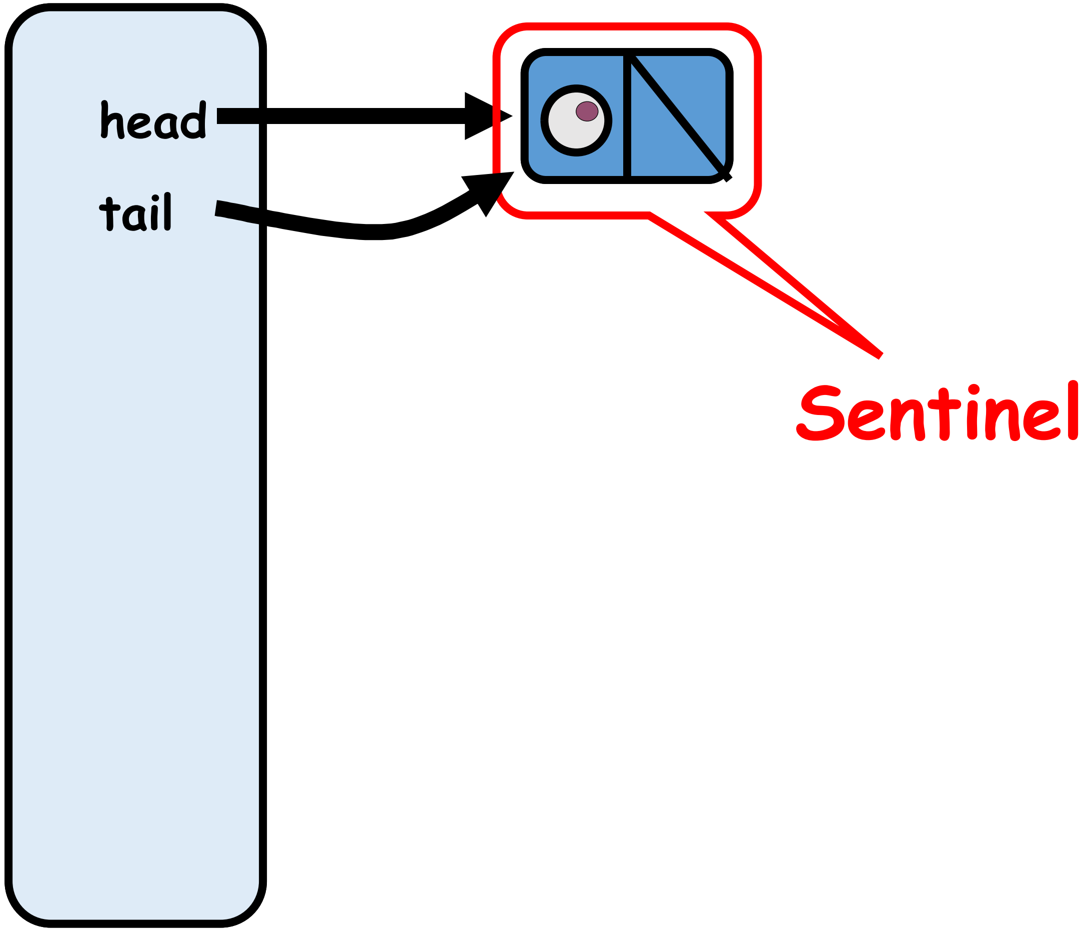


# As Read Ratio Increases

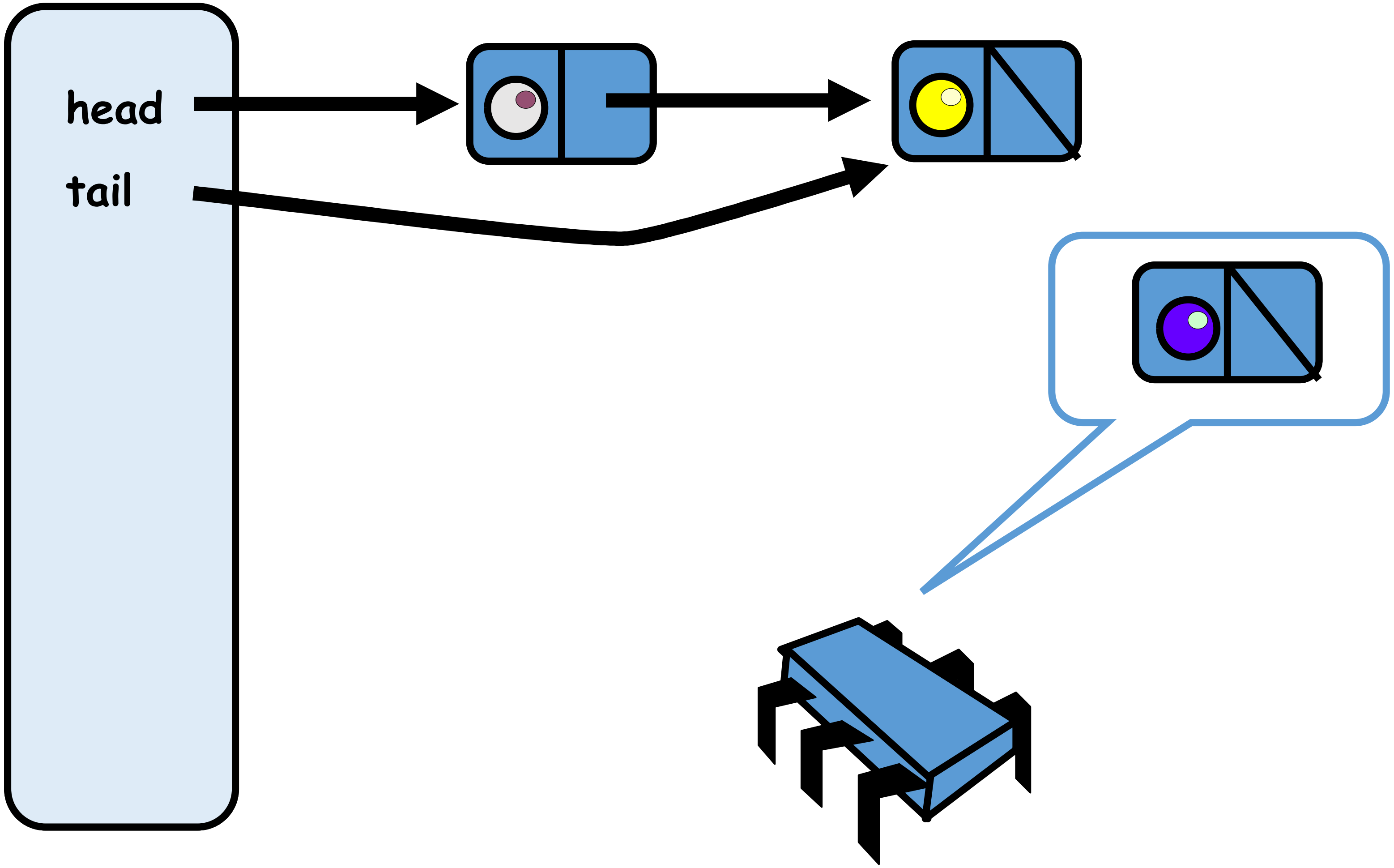
- Throughput



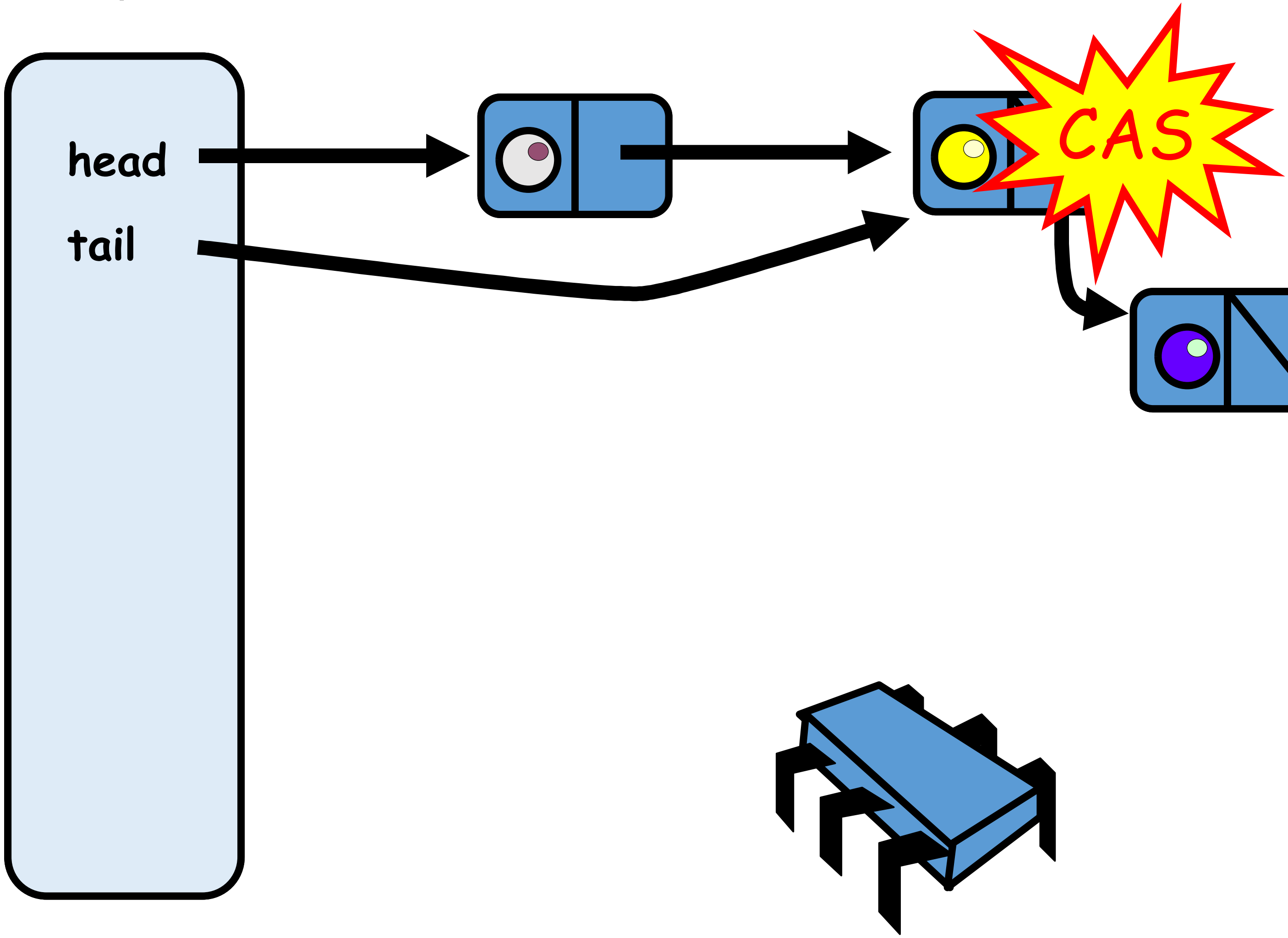
# Concurrent Lock-Free Queue



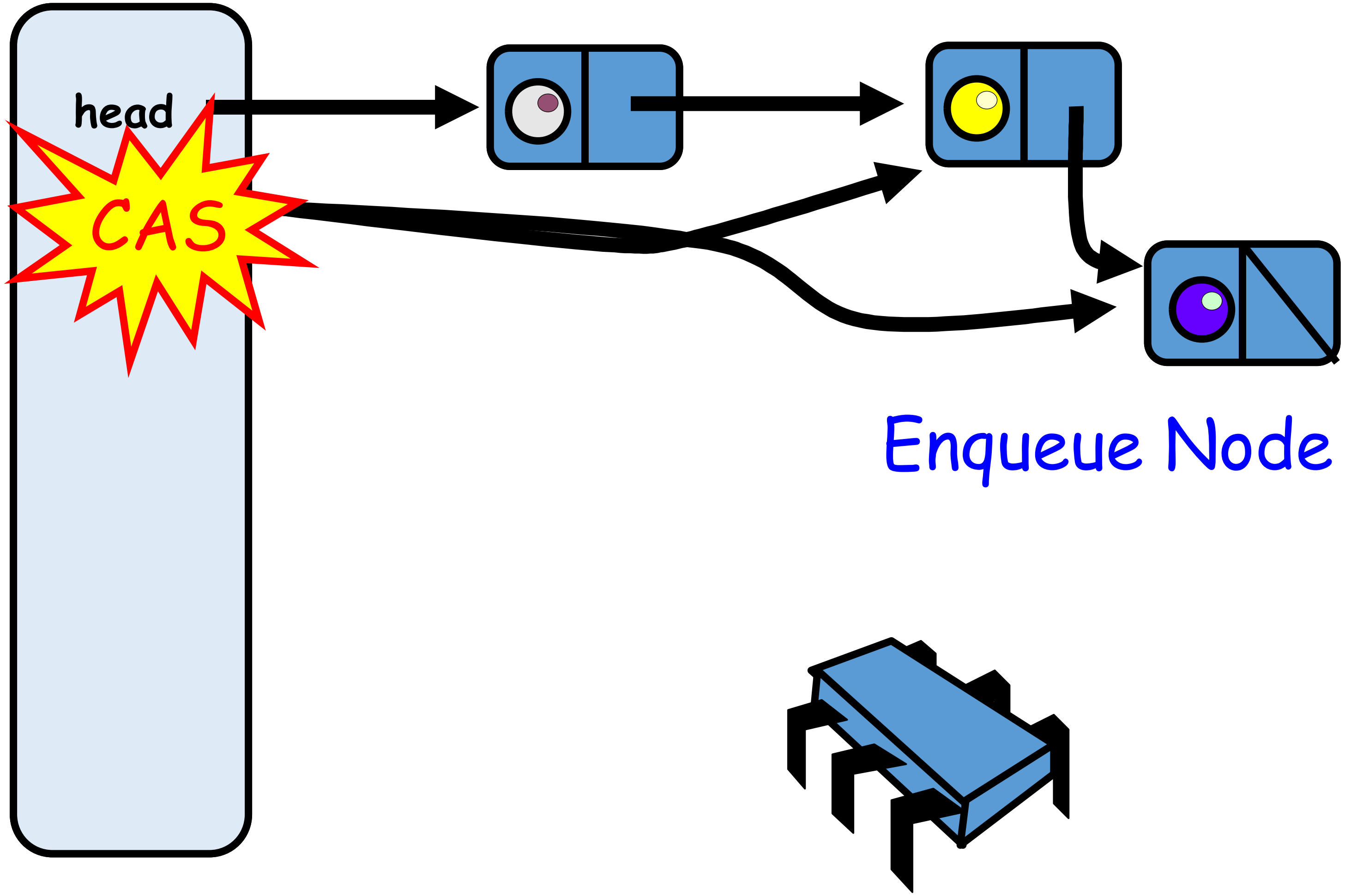
- Enqueue(): Compare-And-Set



- Logical Enqueue

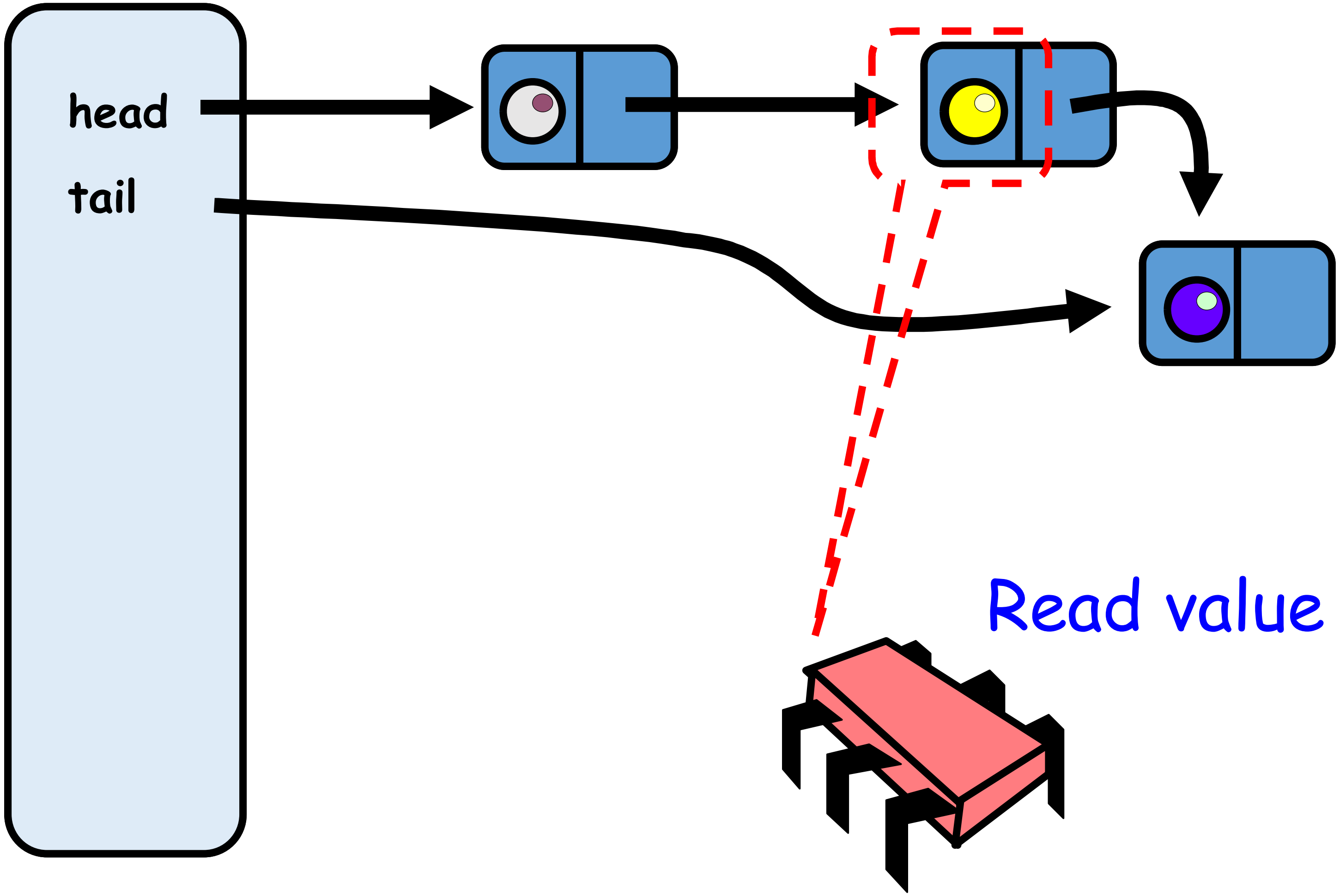


- Physical Enqueue

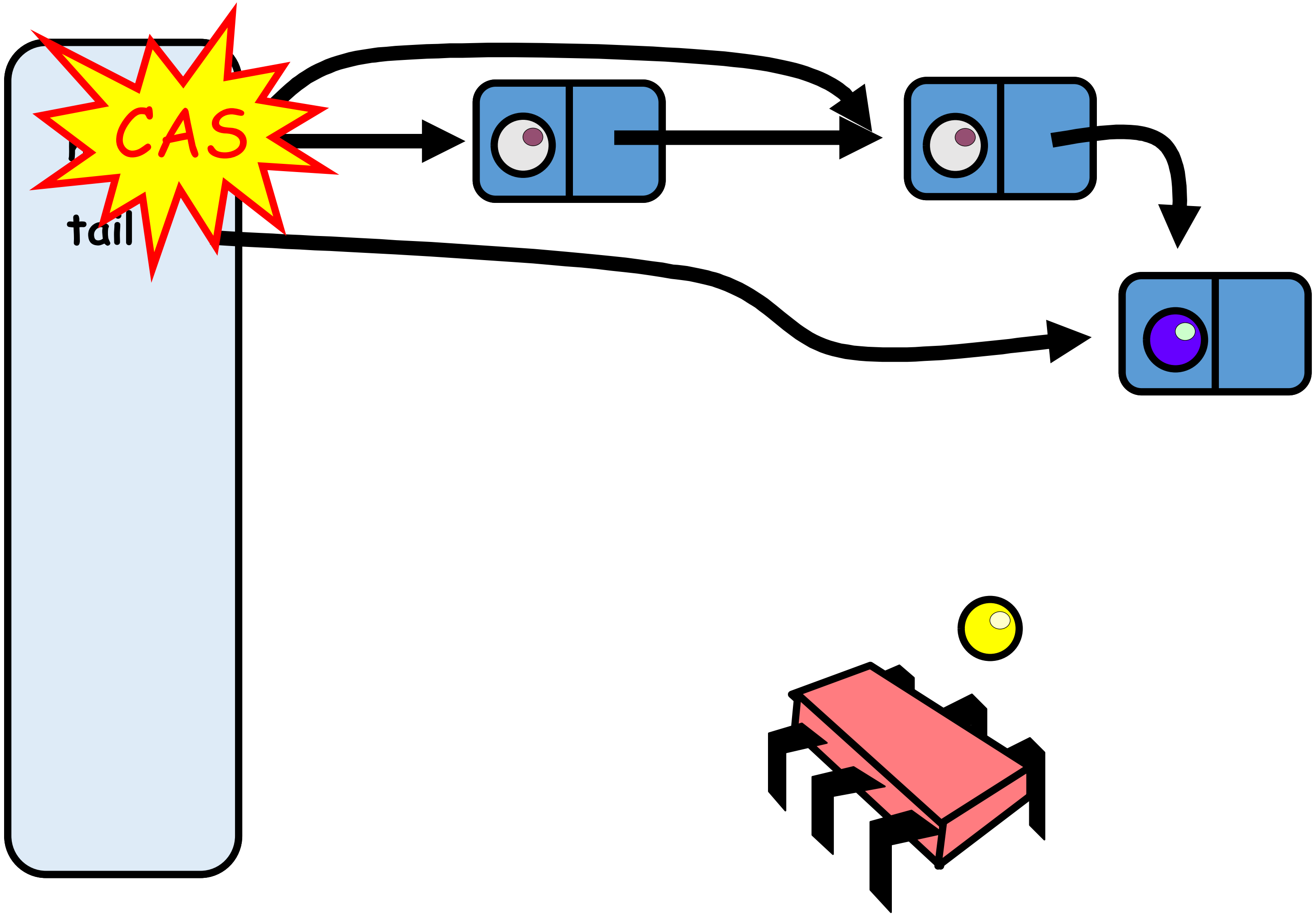


- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good) or
  - Penultimate Node (not so good)
- What do you do if you find
  - A trailing **tail**?
- Stop and help fix it
  - If **tail** node has non-*null* next field
  - CAS the queue's **tail** field to **tail.next**

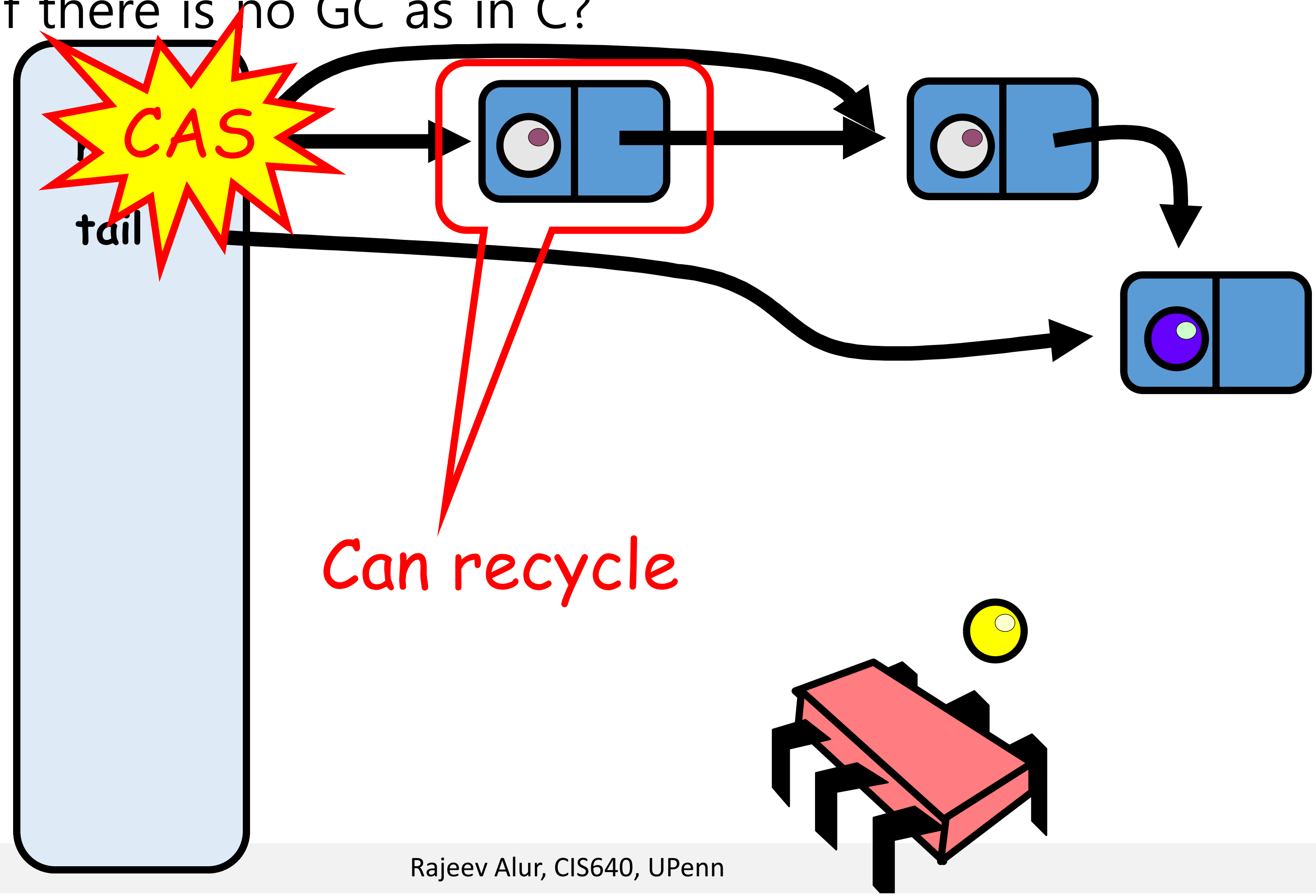
- Dequeue()



- Dequeue(): Compare-and-Set

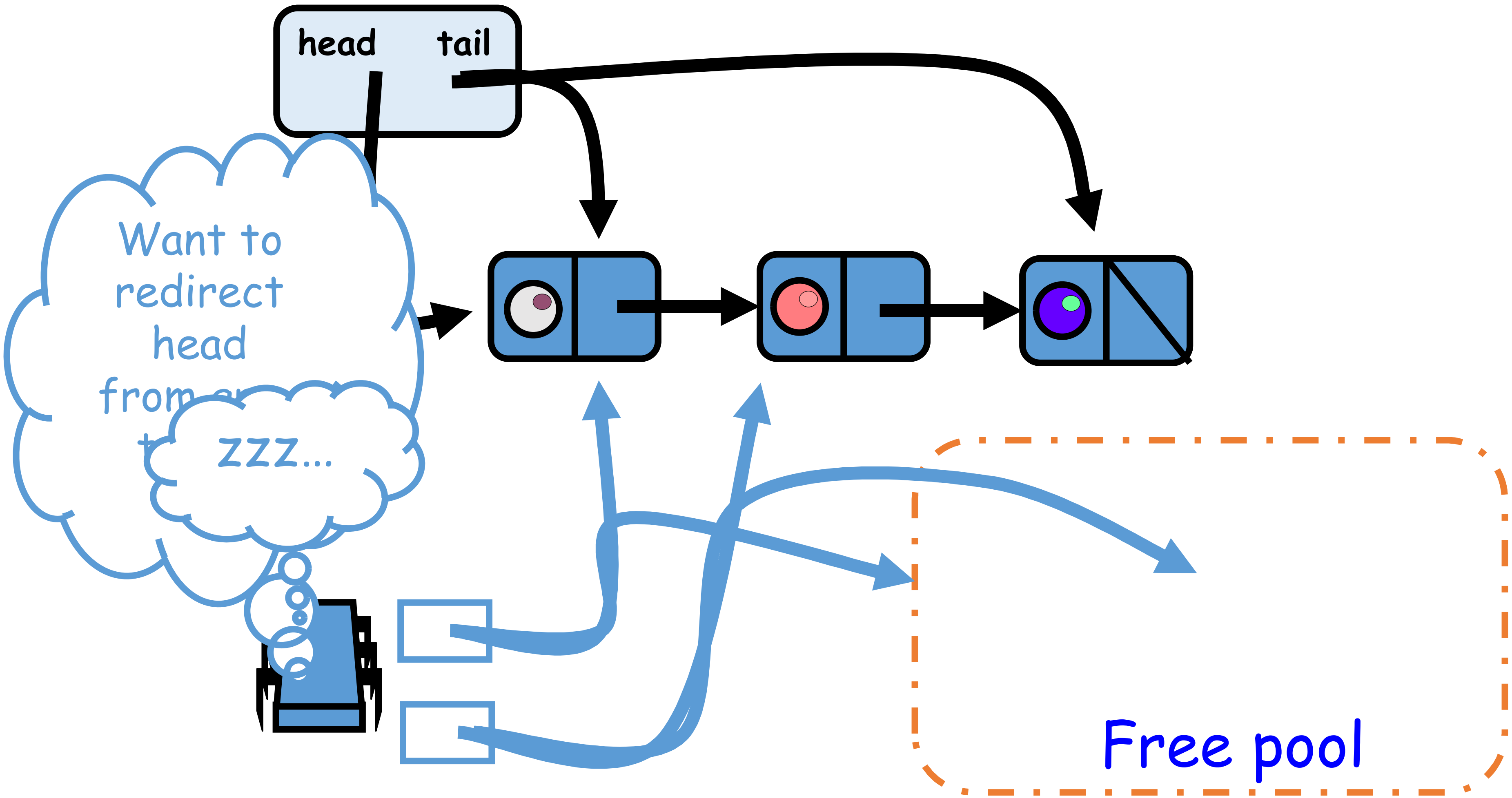


- What do we do with nodes after we dequeue them?
- Java: let garbage collector deal?
- What if there is no GC as in C?



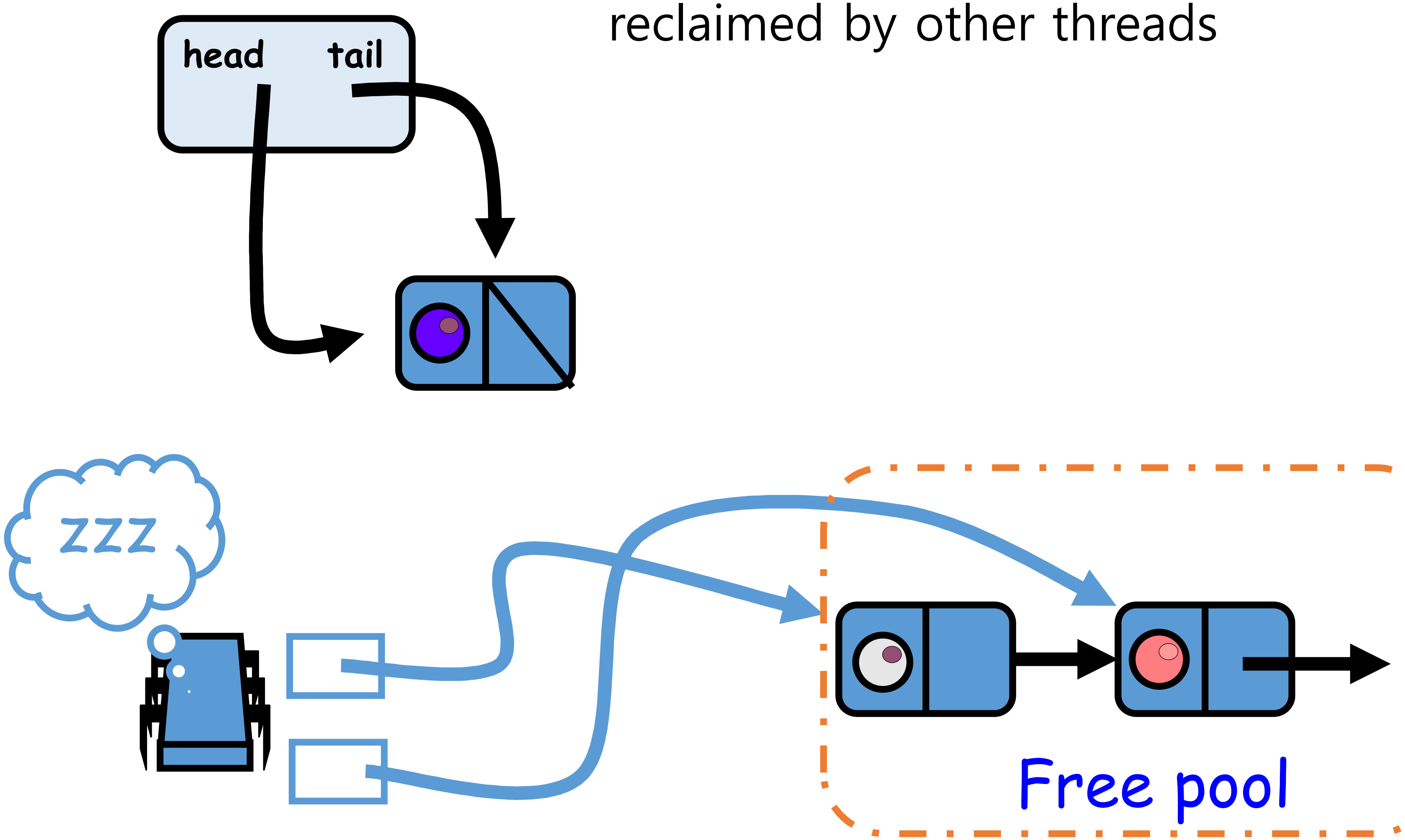
- Simple Solution
  - Each thread manages a free list of unused queue nodes
  - Allocate node: pop from list
  - Free node: push onto list
  - Deal with underflow somehow …
  - vulnerable to ABA Problem

- Why Recycling is Hard?

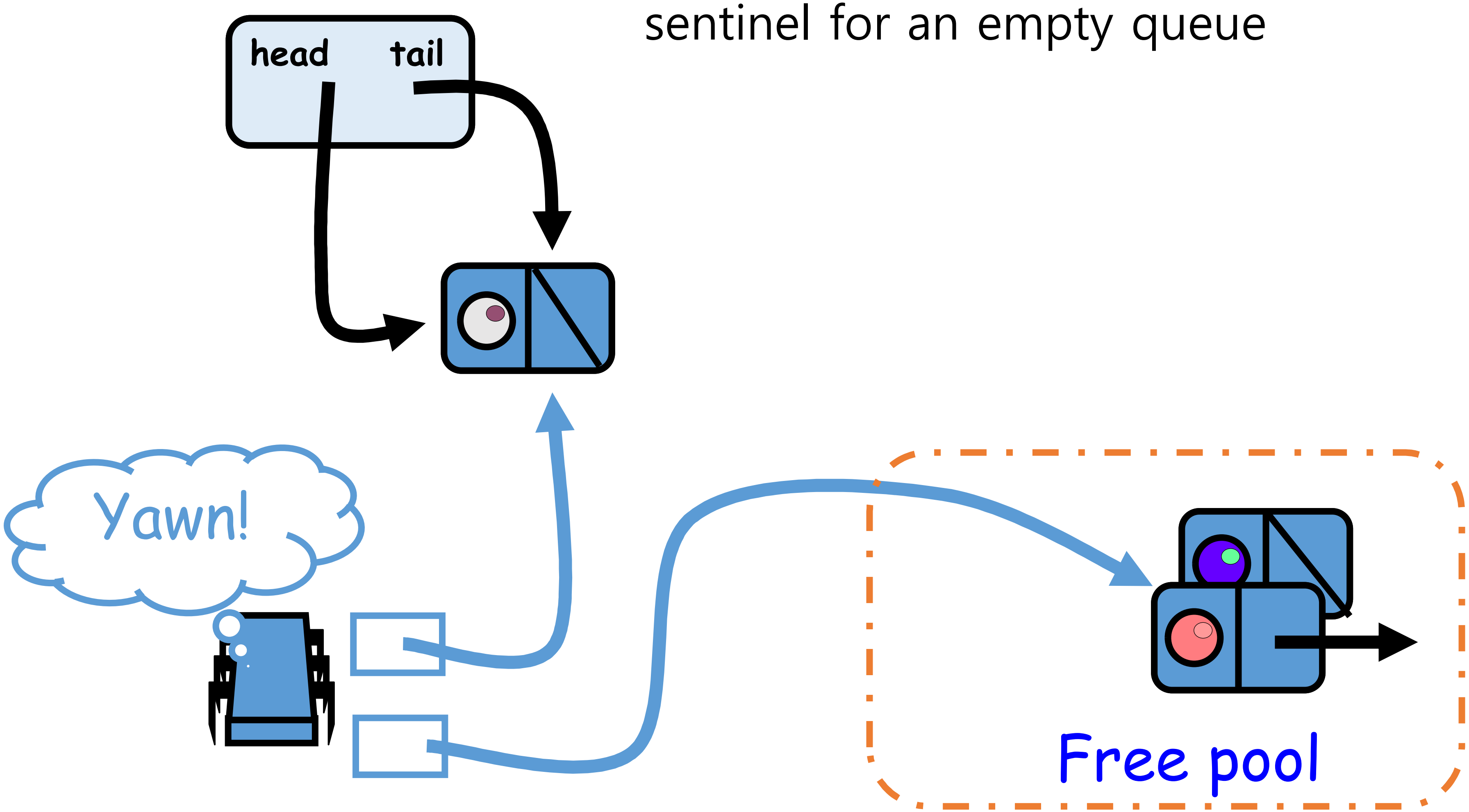


# Memory Reuse and ABA Problem

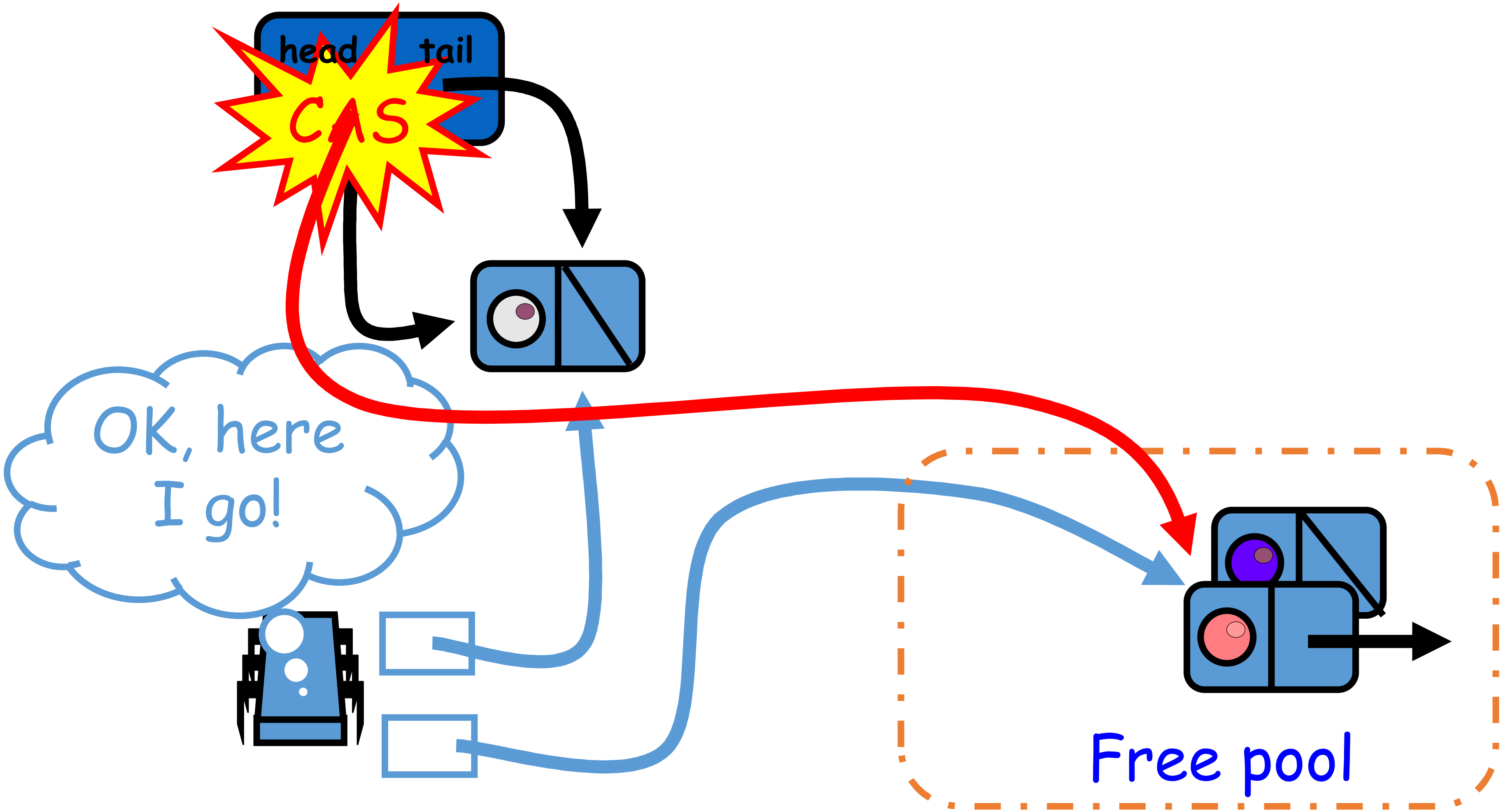
- While the process is suspended, both nodes can be reclaimed by other threads



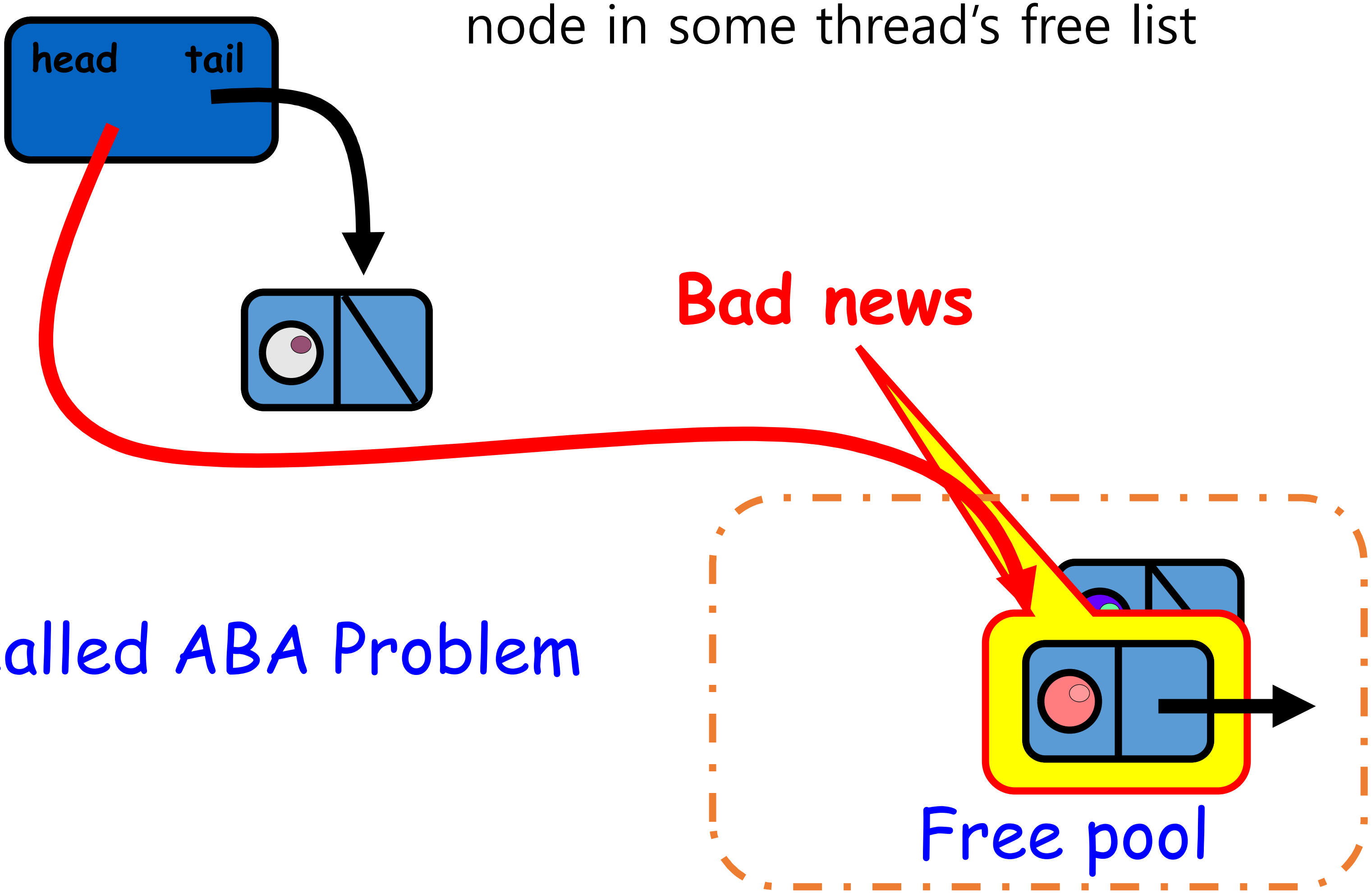
- Later, the original grey node is recycled and becomes a sentinel for an empty queue



- Surprise! CAS works because the head is the same as before.

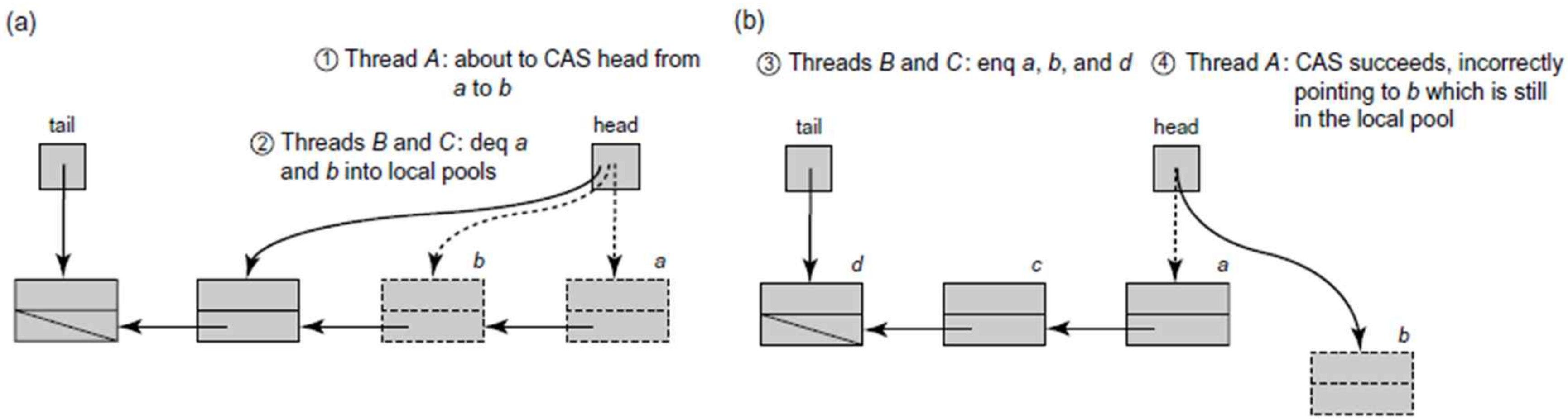


- Tail pointer points to a sentinel while the head points to a node in some thread's free list




This is called ABA Problem

# ABA Problem




- Assume that we use local pools of recycled nodes in our lock-free queue algorithm.
- In Part (a), the dequeuer thread observes that the sentinel node is a, and next node is b.
  - (Step 1) It then prepares to update head by applying a compareAndSet() with old value a and new value b.
  - (Step 2) Suppose however, that before it takes another step, other threads dequeue b, then a, placing both a and b in the free pool.
- In Part (b) (Step 3) node a is reused, and eventually reappears as the sentinel node in the queue.
- (Step 4) thread A now wakes up, calls compareAndSet(), and succeeds in setting head to b, since the old value of head is indeed a. Now, head is incorrectly set to a recycled node.

- Tag each pointer with a counter that is unique over lifetime of node
  - But, pointer size vs. word size issues exist