6.852: Distributed Algorithms
Fall, 2009

Class 19

Today's plan

- Techniques for implementing concurrent objects:
  - Coarse-grained mutual exclusion
  - Fine-grained locking (mutex and read/write)
  - Optimistic locking
  - Lock-free/nonblocking algorithms
  - "Lazy" synchronization
- We illustrate on list-based sets, but the techniques apply to other data structures
- Reading:
  - Herlihy, Shavit, Chapter 9
- Next:
  - Transactional memory
  - HS, Chapter 18
  - Guerraoui, Kapalka

Shared-memory model

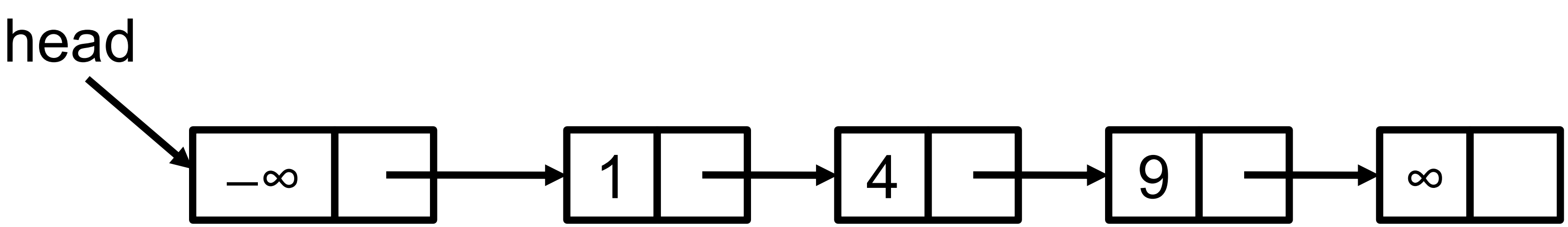
- Shared variables
- At most one memory access per atomic step.
- Read/write access
- Synchronization primitives:
  - Compare-and-swap (CAS)
  - Load-linked/store-conditional (LL/SC)
  - Assume lock and unlock methods for every object.
- Most (not all) of our algorithms use locking.
- Memory management:
  - Allocate objects dynamically, assume unlimited supply.
  - In practice, would garbage collect and reuse, but we won't worry about this.
- Assume no failures (mostly).

Correctness guarantees

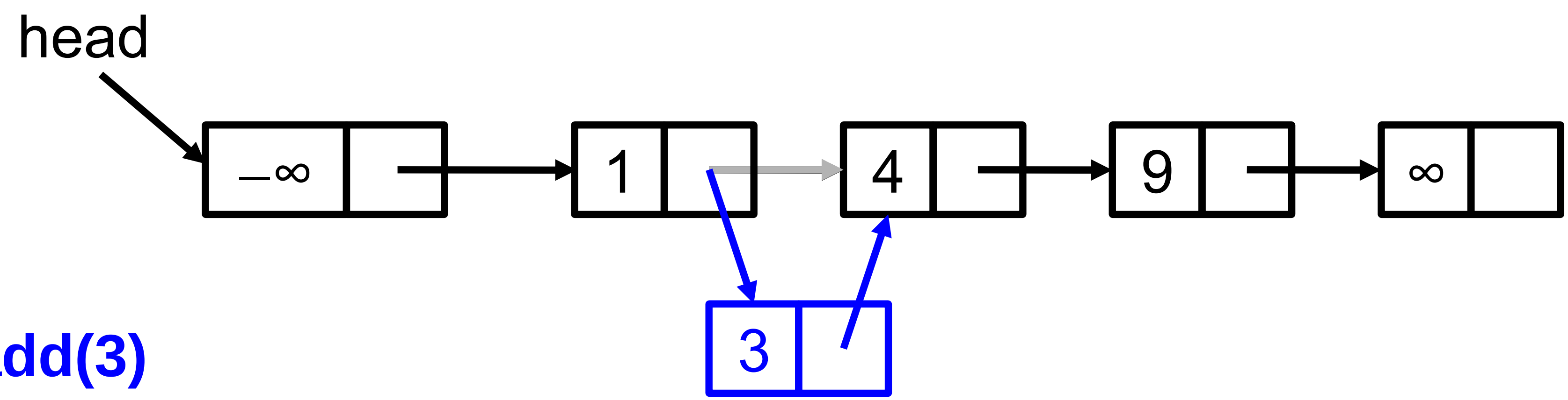
- Linearizability (atomicity) of object operations.
- Liveness properties:
  - Different guarantees for different algorithms.
  - Progress:
    - Some operations keep completing.
  - Lockout-freedom (AKA starvation-freedom):
    - Every operation completes.
  - "Nonblocking" conditions:
    - Wait-freedom: Even if other processes stop, a particular operation by a process that keeps taking steps eventually finishes.
    - Lock-freedom: Even if some processes stop, if some keep taking steps, then some operation finishes.
    - Can think of the stopped processes as failing, or as going slowly.
    - Captures the idea that slow processes don't block others.
    - Rules out locking strategies.
- Performance
  - Worst-case (time bounds) vs. average case (throughput).
  - No good formal models

List-based sets

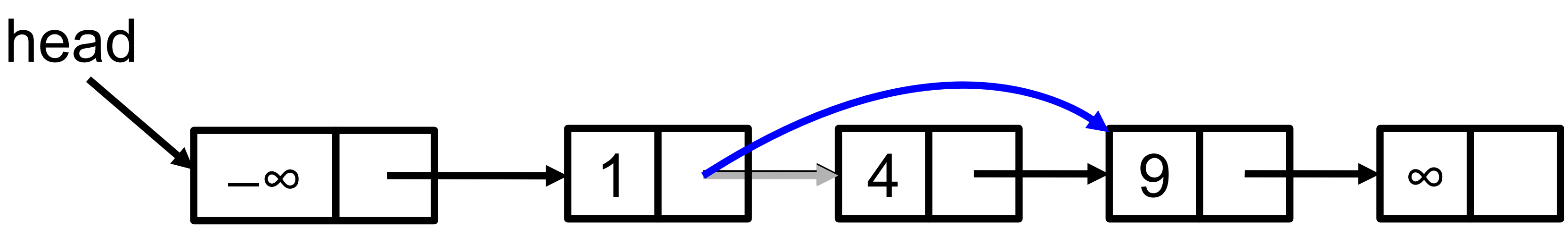
- Data type: Set S of integers (no duplicates)
  - S.add(x): Boolean: S := S ∪ {x}; return true iff x not already in S
  - S.remove(x): Boolean: S := S \ {x}; return true iff x in S initially
  - S.contains(x): Boolean: return true iff x in S (no change to S)
- Simple ordered linked-list-based implementation
  - Illustrates techniques useful for pointer-based data structures.
  - Unless set is small, this is a poor data structure for this specific data type--better to use arrays, hash tables, etc.



Sequential list-based set



add(3)



remove(4)

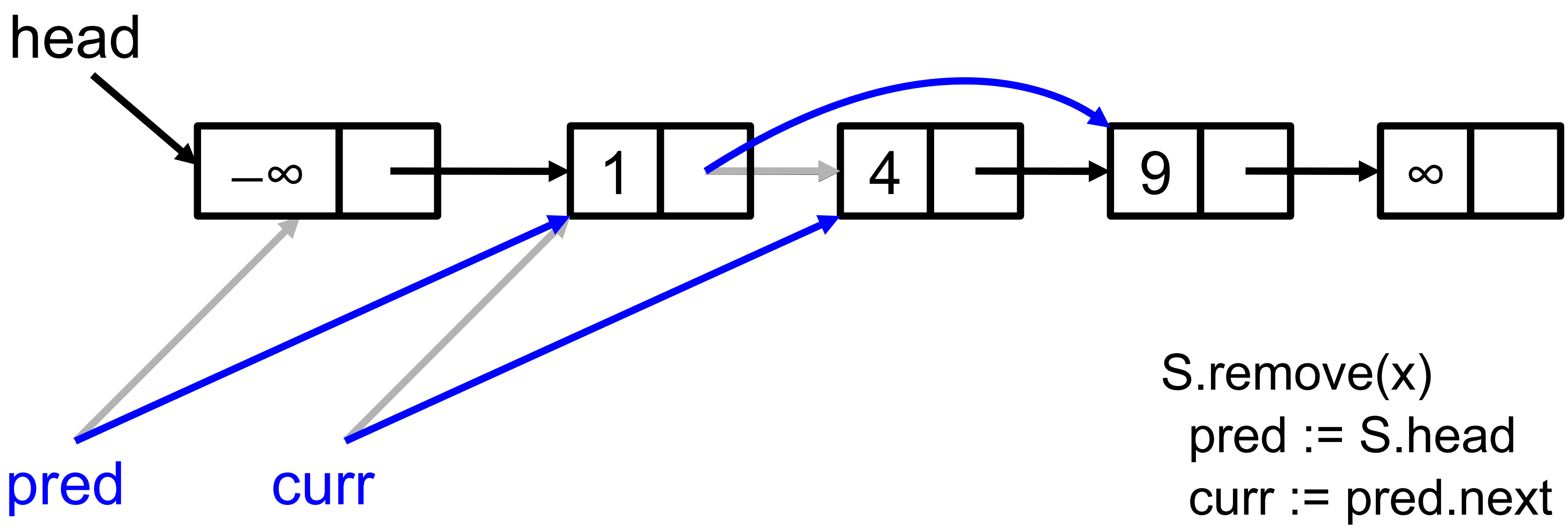
Sequential list-based set

```
S.add(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    return false
  else
    node := new Node(x)
    node.next := curr
    pred.next := node
    return true
```

```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next := curr.next
    return true
  else
    return false
```

```
S.contains(x)
  curr := S.head
  while (curr.key < x)
    curr := curr.next
  if curr.key = x then
    return true
  else
    return false
```

Sequential list-based set



remove(4)

```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next := curr.next
    return true
  else
    return false
```

Correctness

- Assume algorithm queues up operations, runs them sequentially.
- Atomicity (linearizability):
  - Show the algorithm implements a canonical atomic set object.
  - Use forward simulation relation: Set consists of those elements that are reachable from the head of the list via list pointers.
  - When do "perform" steps occur?
    - add(x): If successful, then when pred.next := node, else any time during the operation.
    - remove(x): If successful, then when pred.next := curr.next, else any time during the operation.
    - contains(x): Any time during the operation.
  - Proof uses invariants saying that the list is ordered and contains no duplicates.
- Liveness: Lockout-free, but blocking (not wait-free or lock-free)

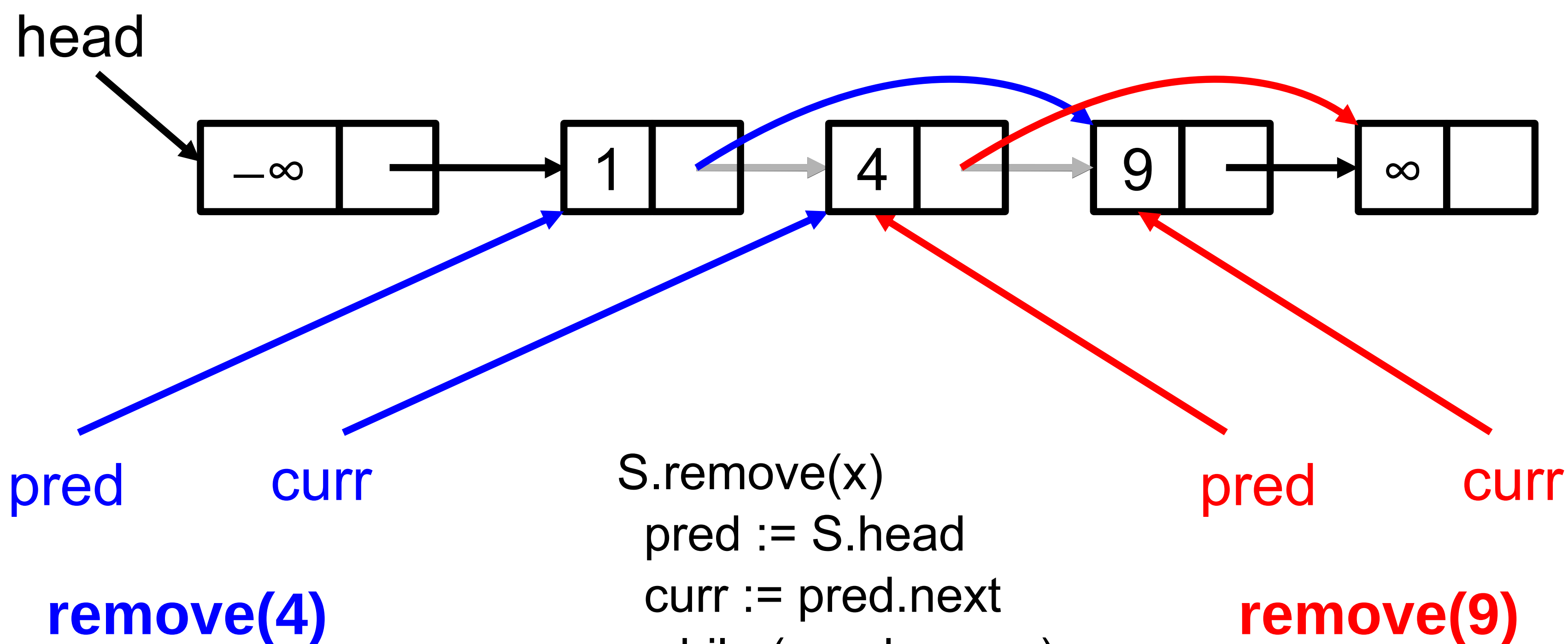
Invariants

- Keys strictly increase down the list.
  - List is ordered.
  - No duplicates.
- Keys of first and last nodes (i.e., the "sentinels") are $-\infty$ and ∞ respectively.
- pred.key < x
- pred.key < curr.key
- pred.next ≠ null
- …

Allowing concurrent access

- Can this algorithm tolerate concurrent execution of the operations by different processes?
- What can go wrong?
- How can we fix it?

Concurrent operations (bad)




```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next := curr.next
    return true
  else
    return false
```

Techniques for managing concurrent operations

- Coarse-grained mutual exclusion
- Fine-grained locking
- Optimistic locking
- Lock-free/nonblocking algorithms
- “Lazy” synchronization

Coarse-grained mutual exclusion

- Each process acquires a global lock, for the entire time it is executing significant steps of an operation implementation.

Coarse-grained locking

Why can we unlock early here?

```
S.add(x)
  S.lock()
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    S.unlock()
    return false
  else
    node := new Node(x)
    node.next := curr
    pred.next := node
    S.unlock()
   return true
```

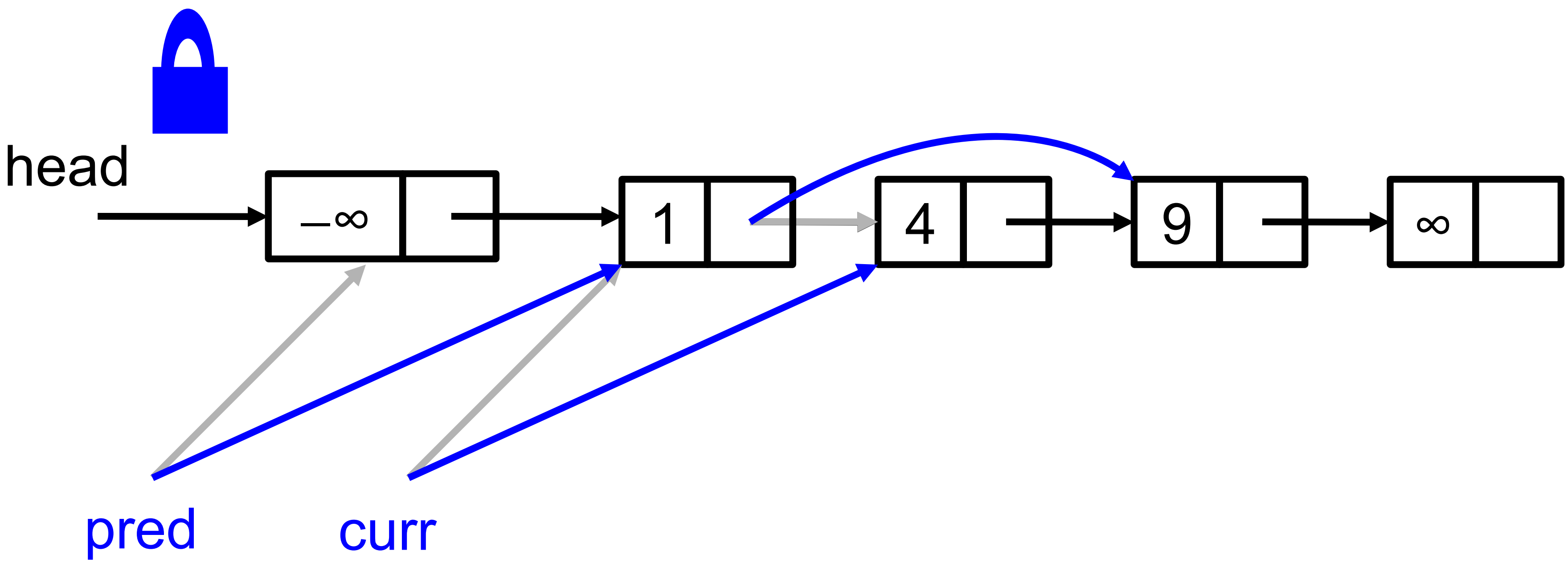
```
S.lock()
pred := S.head
curr := pred.next
while (curr.key < x)
  pred := curr
  curr := pred.next
if curr.key = x then
  pred.next := curr.next
  S.unlock()
  return true
else
  S.unlock()
  return false
```

```
S.contains(x)
  S.lock()
  curr := S.head
  while (curr.key < x)
    curr := curr.next
  S.unlock()
  if curr.key = x then
    return true
  else
    return false
```

Correctness

- Similar to sequential implementation.
- Atomicity:
  - Show the algorithm implements a canonical atomic set object.
  - Use forward simulation: S = elements that are reachable in the list
  - When do "perform" steps occur?
    - add(x): If successful, then when pred.next := node, else any time the lock is held.
    - remove(x): If successful, then when pred.next := curr.next, else any time the lock is held.
    - contains(x): Any time the lock is held.
  - Invariant: If an operation holds the lock, then any node it visits is reachable in the list.
- Liveness:
  - Guarantees progress, assuming that the lock does.
  - May or may not be lockout-free, depending on whether the lock is.
  - Blocking (not wait-free or lock-free):
    - Everything comes to a halt if someone stops while holding the lock.

Coarse-grained locking



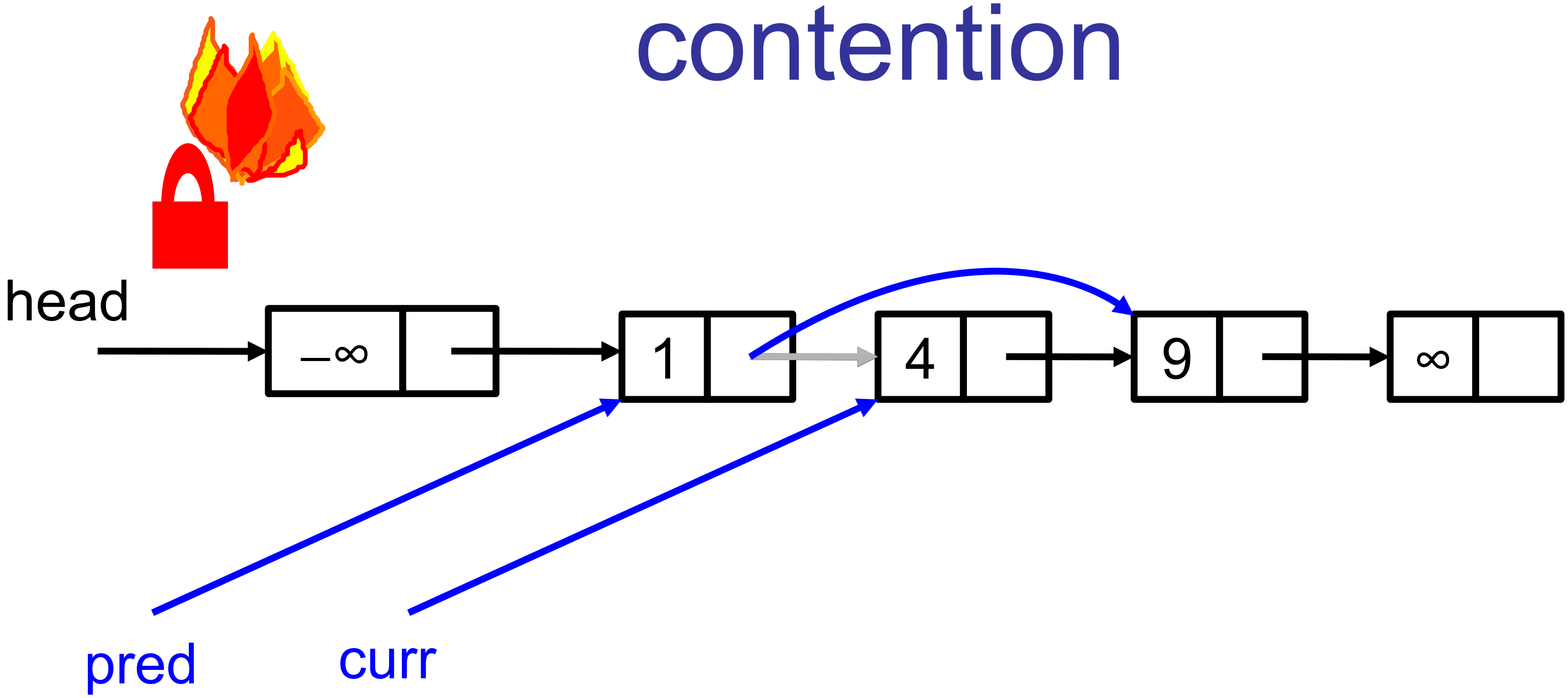
remove(4)

Coarse-grained locking

For many applications, this is the best solution!
(Don't underrate simplicity.)

- Easy
  - to write,
  - to prove correct.
- Guarantees progress
- If we use queue locks, it's lockout-free.
- But:
  - Blocking (not wait-free, not lock-free)
  - Poor performance when contention is high
    - Essentially no concurrent access.
    - But often good enough for low contention.

Coarse-grained locking with high contention



remove(4) **remove(9)** **add(6)** **contains(4)** **add(3)**

Improving coarse-grained locking

- Reader/writer locks
  - Multiple readers can hold the lock simultaneously, but writer cannot share with anyone else (reader or writer).
- Using reader/writer lock for coarse-grained locking, in the list-based set implementation:
  - Contains takes only a read lock
    - Can be a big win if contains is the most common operation.
  - What about add or remove that returns false?
    - Let add/remove start with a read lock, then “upgrade” to a write lock if needed.
    - If it can’t upgrade, abandon/restart the operation.

Fine-grained locking

- Associate locks with smaller pieces of data, not entire data structure.
- Process acquires/releases locks as it executes steps of an operation.
- Operations that work on disjoint pieces of data proceed concurrently.

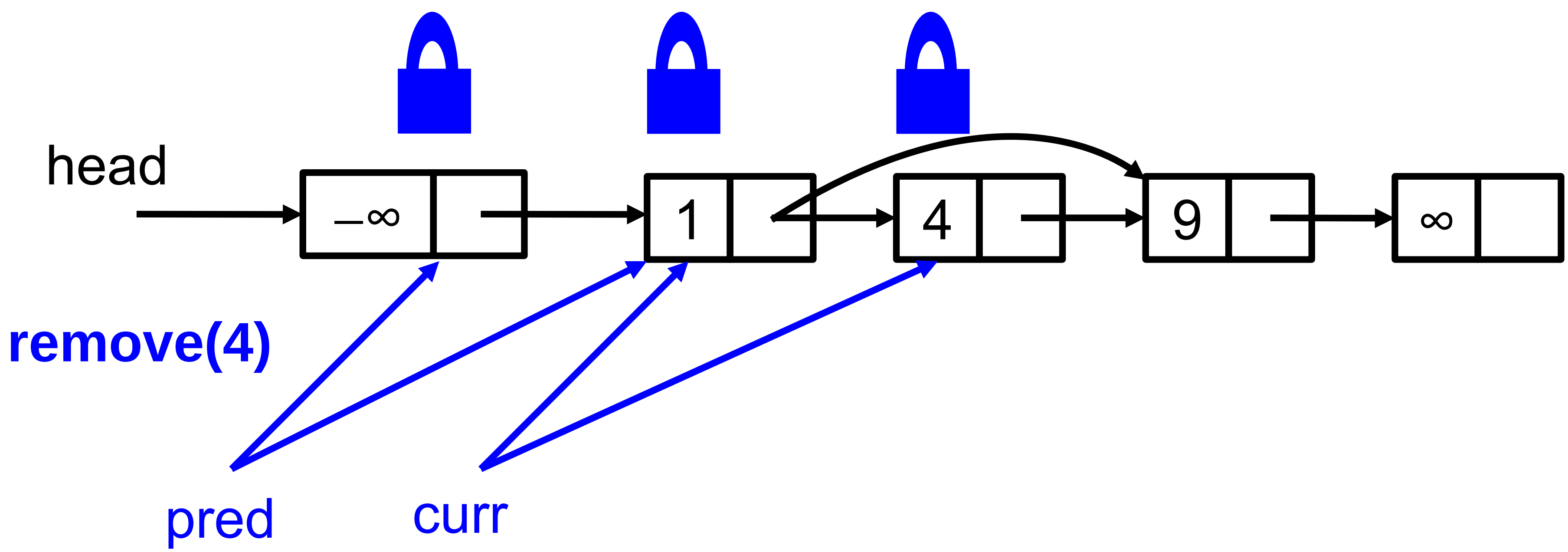
Two-phase locking

- Finish acquiring all locks before releasing any.
  - Typically, release all locks at end of the op: "strict 2-phase locking".
- Easy to prove atomicity:
  - Serialize each operation at any point when it holds all its locks.
  - For strict 2-phase locking, usually the end of the operation.
  - Algorithm behaves like sequential algorithm, with operations performed in order of serialization points.
- But acquiring all the locks at once can be costly (delays).
- Must avoid deadlock, e.g., by acquiring locks in predetermined order.

- Naïve 2-phase locking for list-based set implementation:
  - Lock each node as visited, using a mutex lock.
  - Avoids deadlock by acquiring all locks in list order.
  - Doesn't help performance.
  - Using reader/writer locks might help performance, but introduces new deadlock possibilities.

Hand-over-hand locking

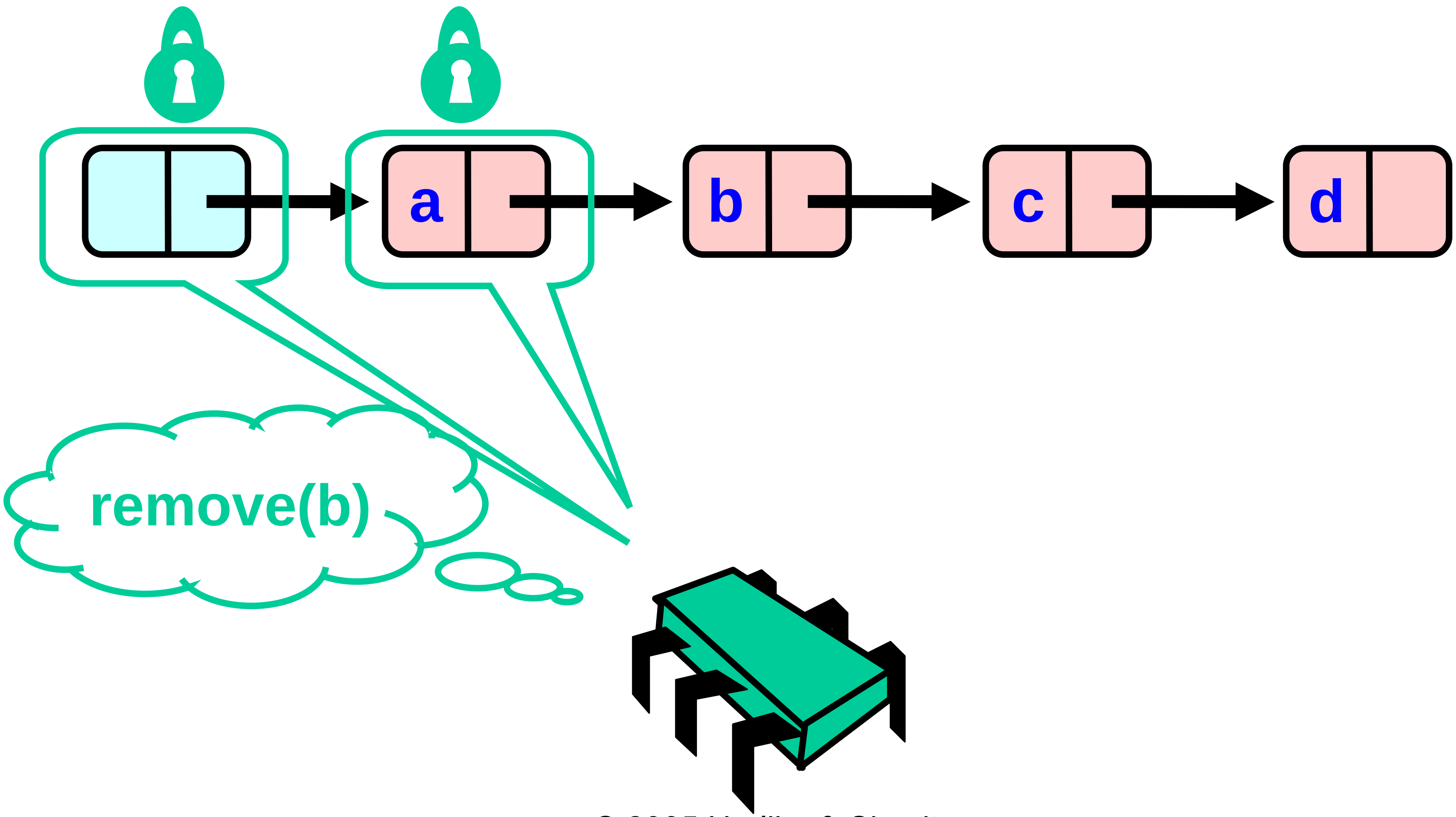
- Fine-grained locking, but not “two-phase”
  - Atomicity doesn't follow from general rule; trickier to prove.
- Each process holds at most two locks at a time.
  - Acquires lock for successor before releasing lock for predecessor.
- Keeps operations “pipelined”.



Hand-over-hand locking

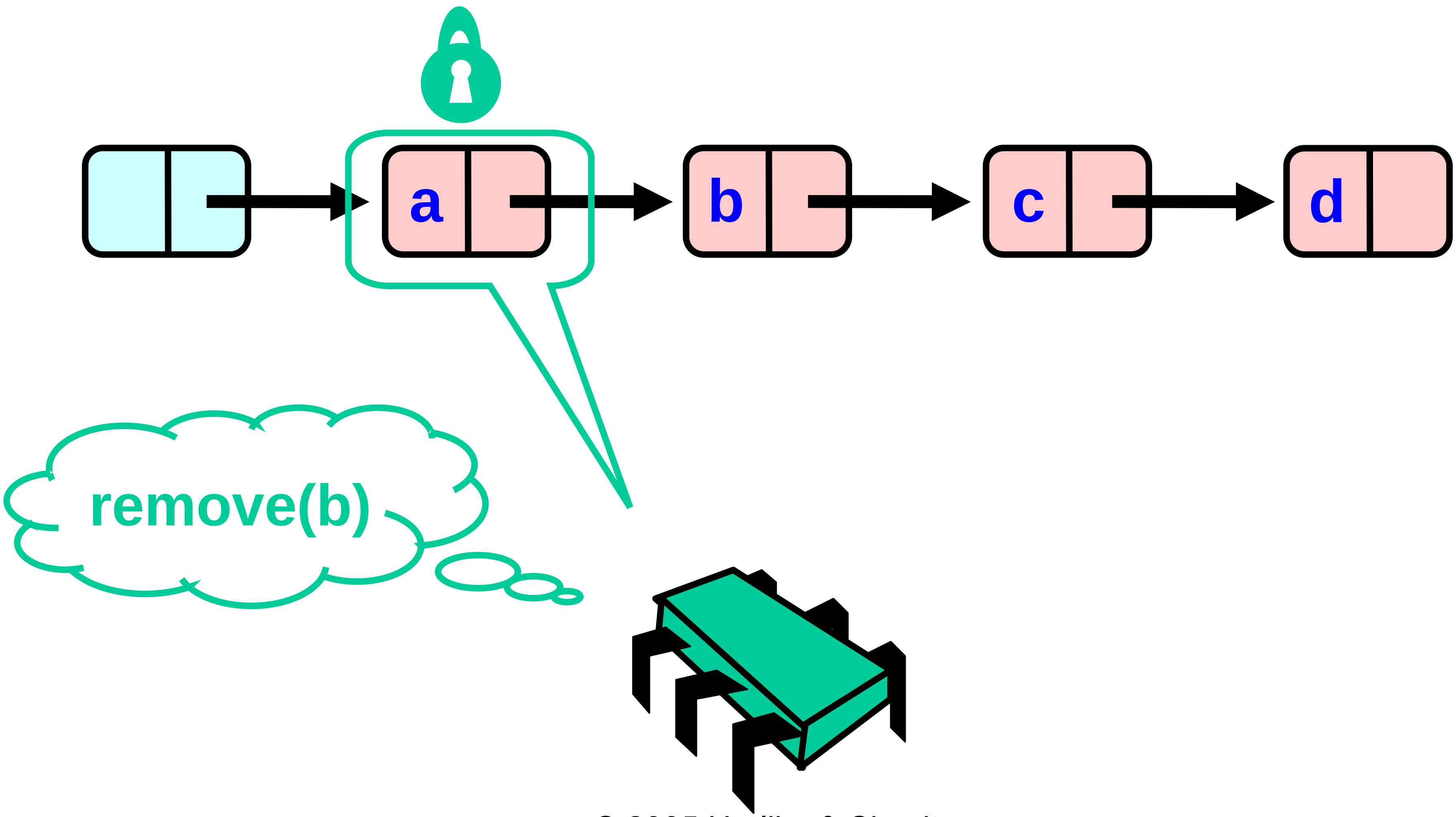
- Must we lock a node we are trying to remove?
- Can’t we just lock its predecessor, while resetting the predecessor’s next pointer?
- No. Counterexample (from Herlihy and Shavit’s slides):

Removing a Node

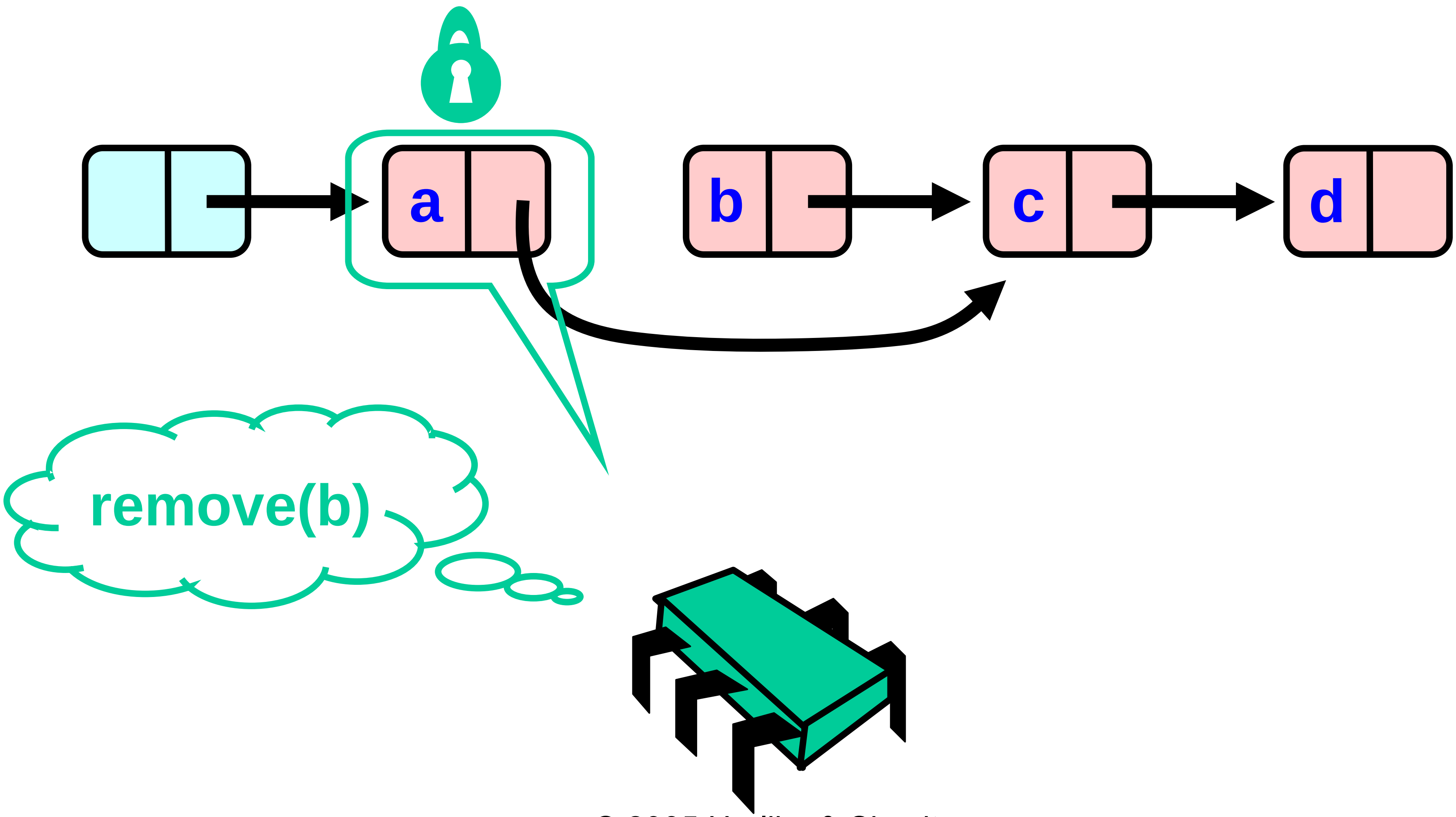


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

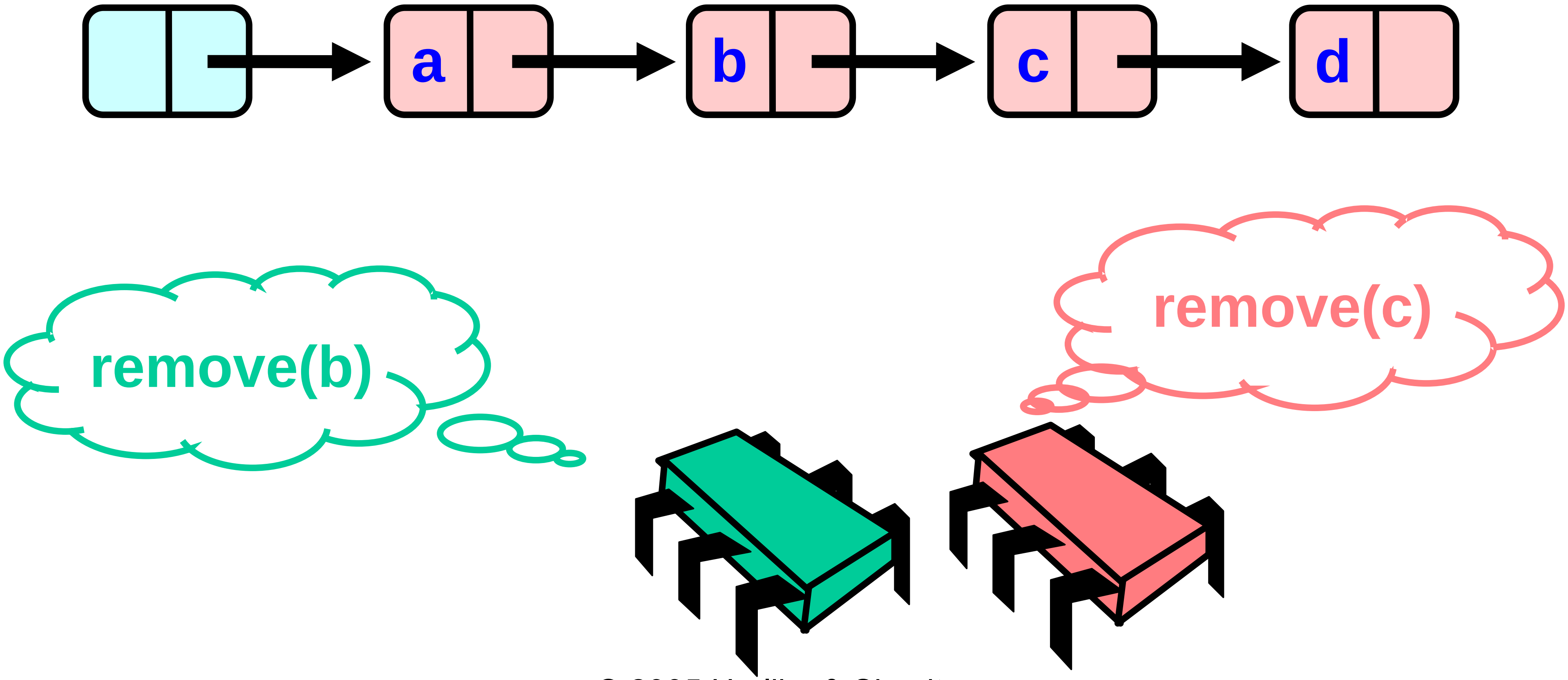
Removing a Node



Removing a Node

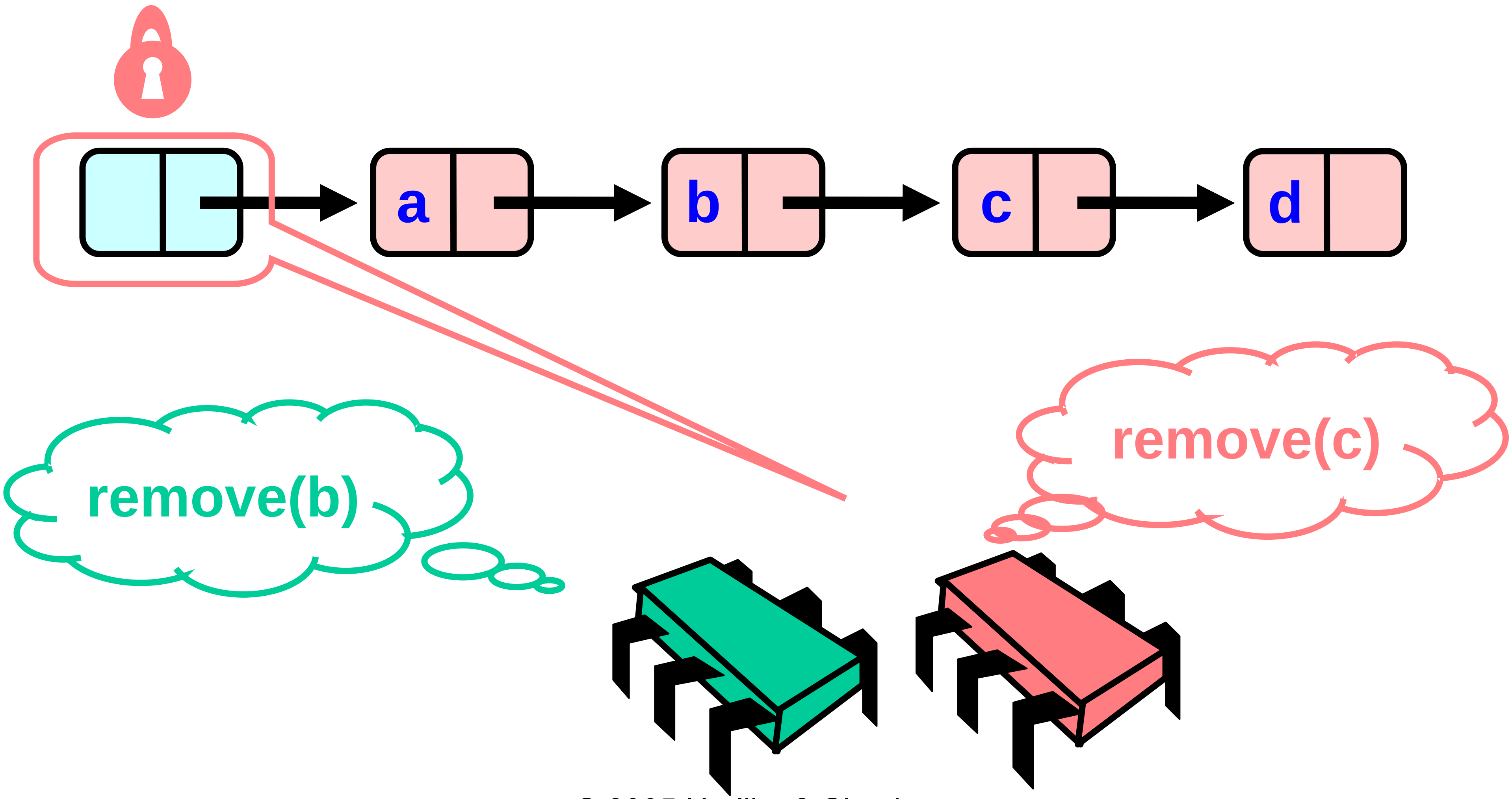


Removing Two Nodes



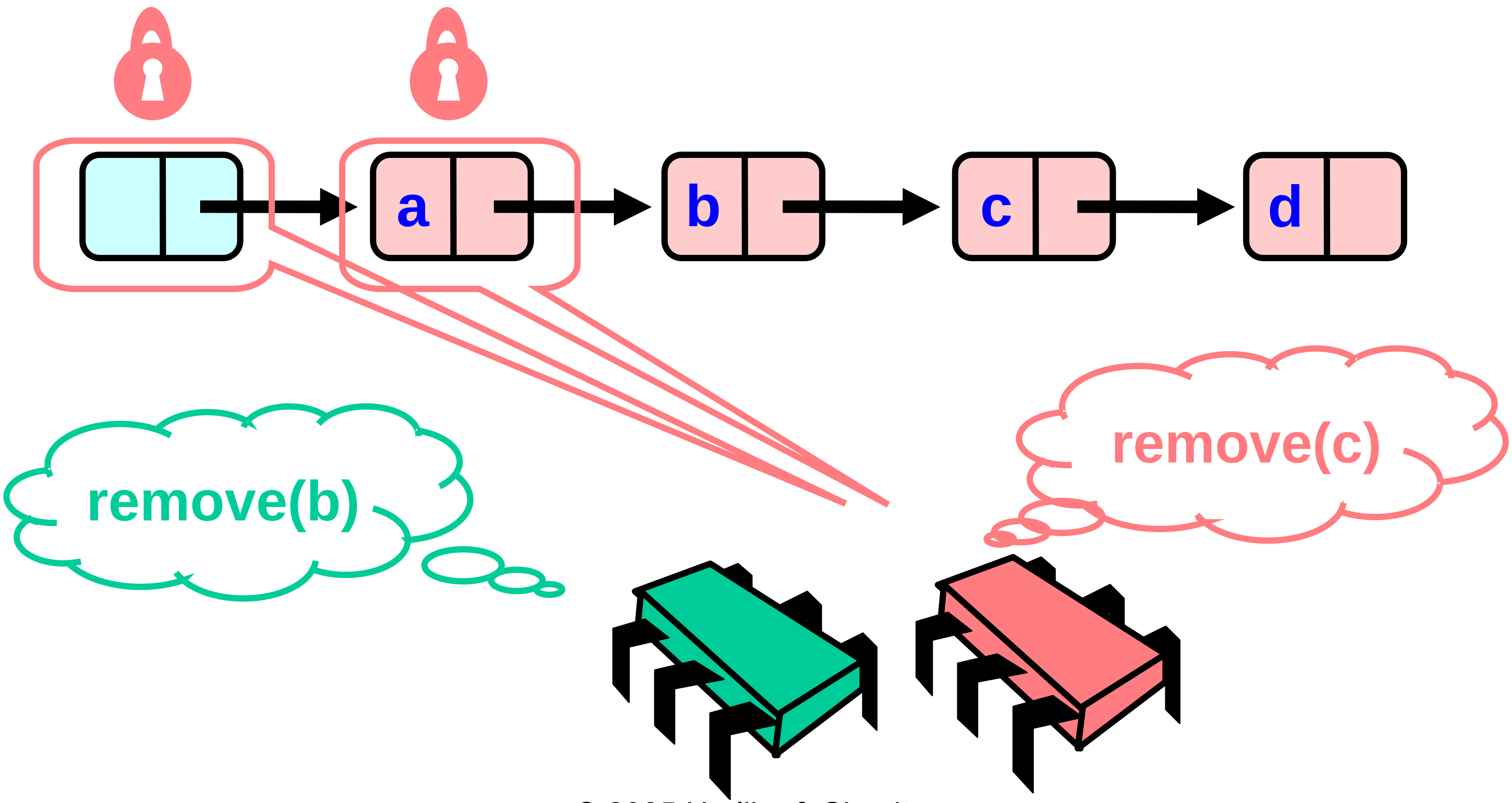
From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Removing Two Nodes

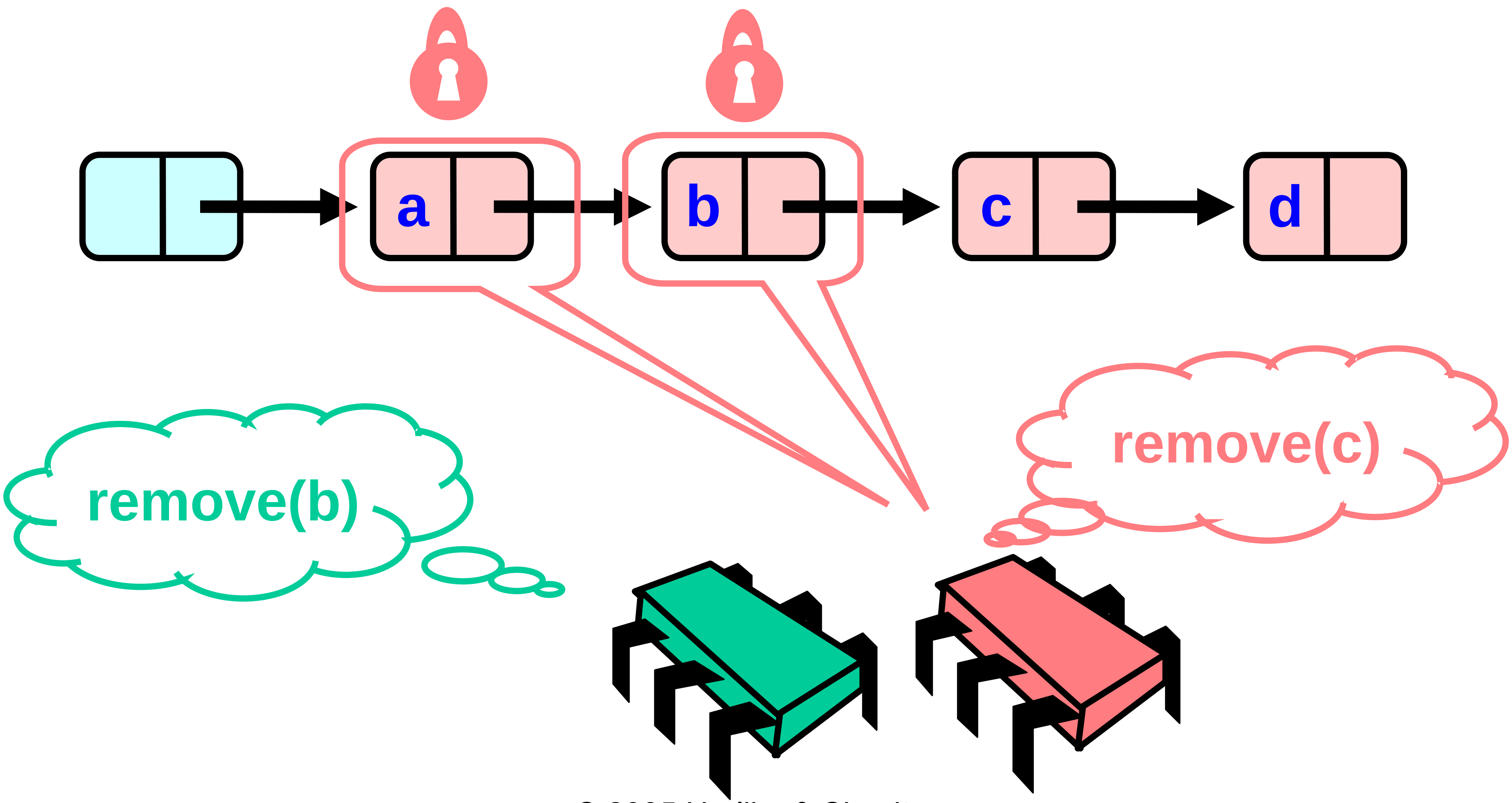


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

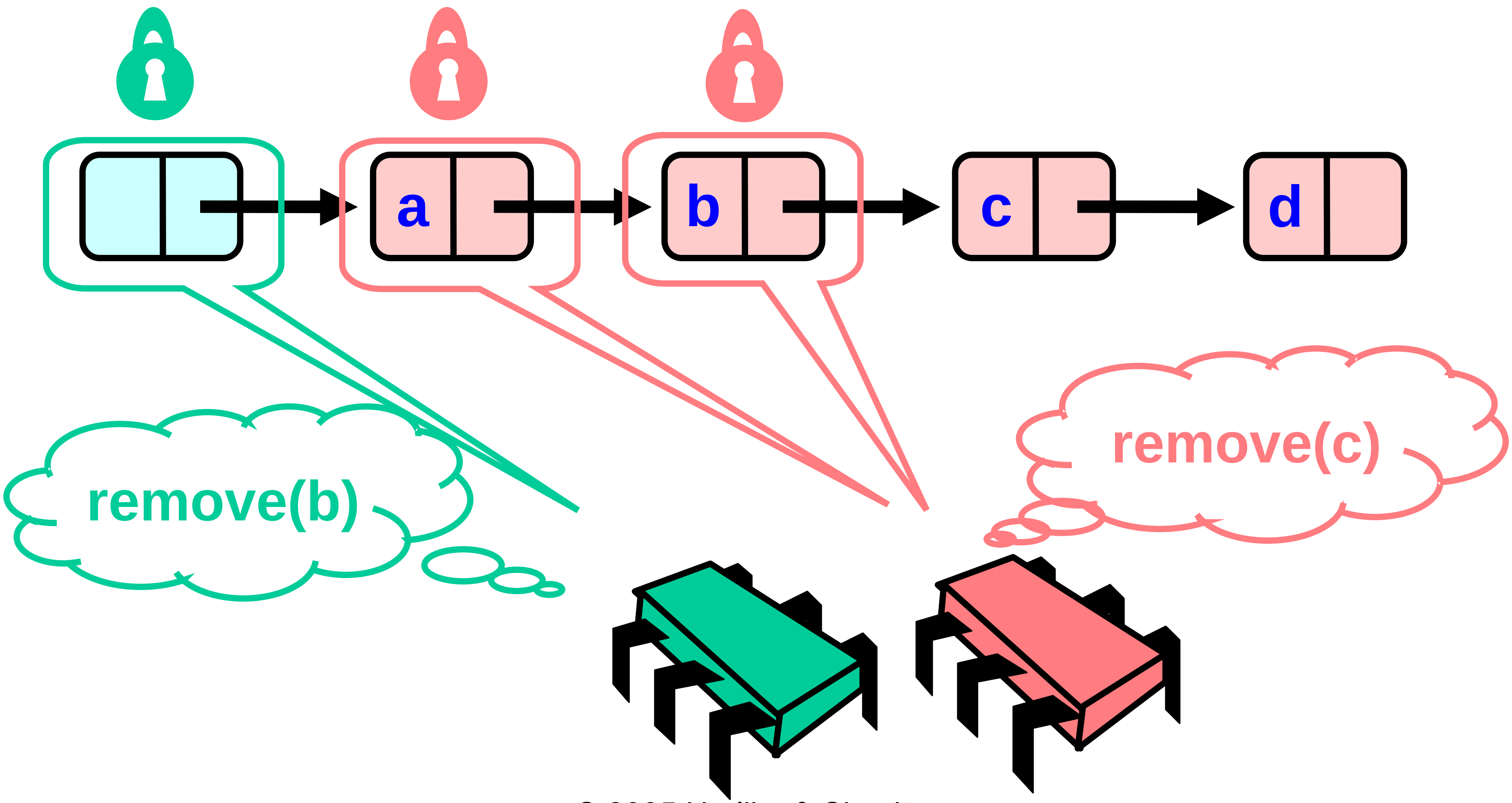
Removing Two Nodes



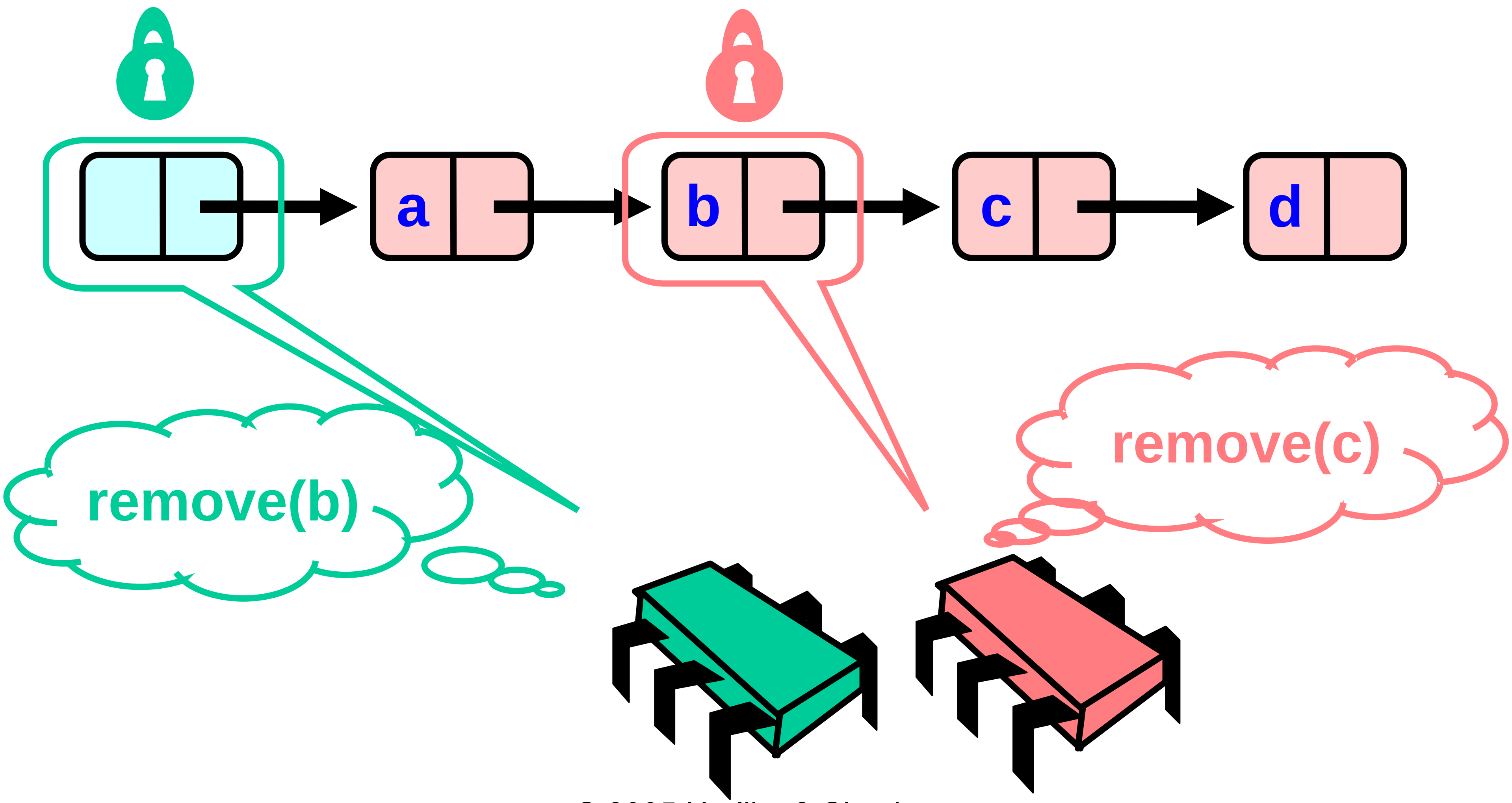
Removing Two Nodes



Removing Two Nodes

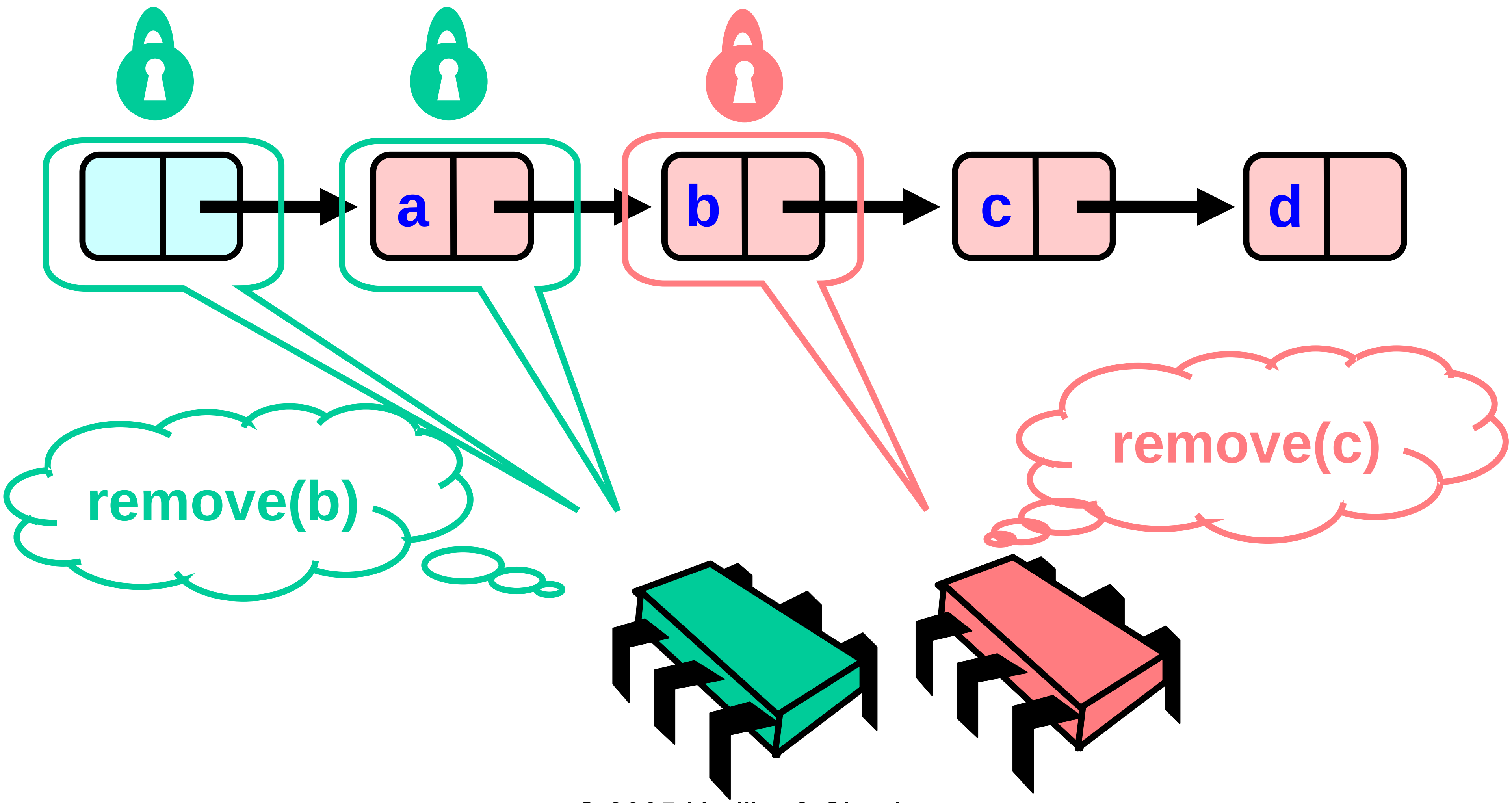


Removing Two Nodes

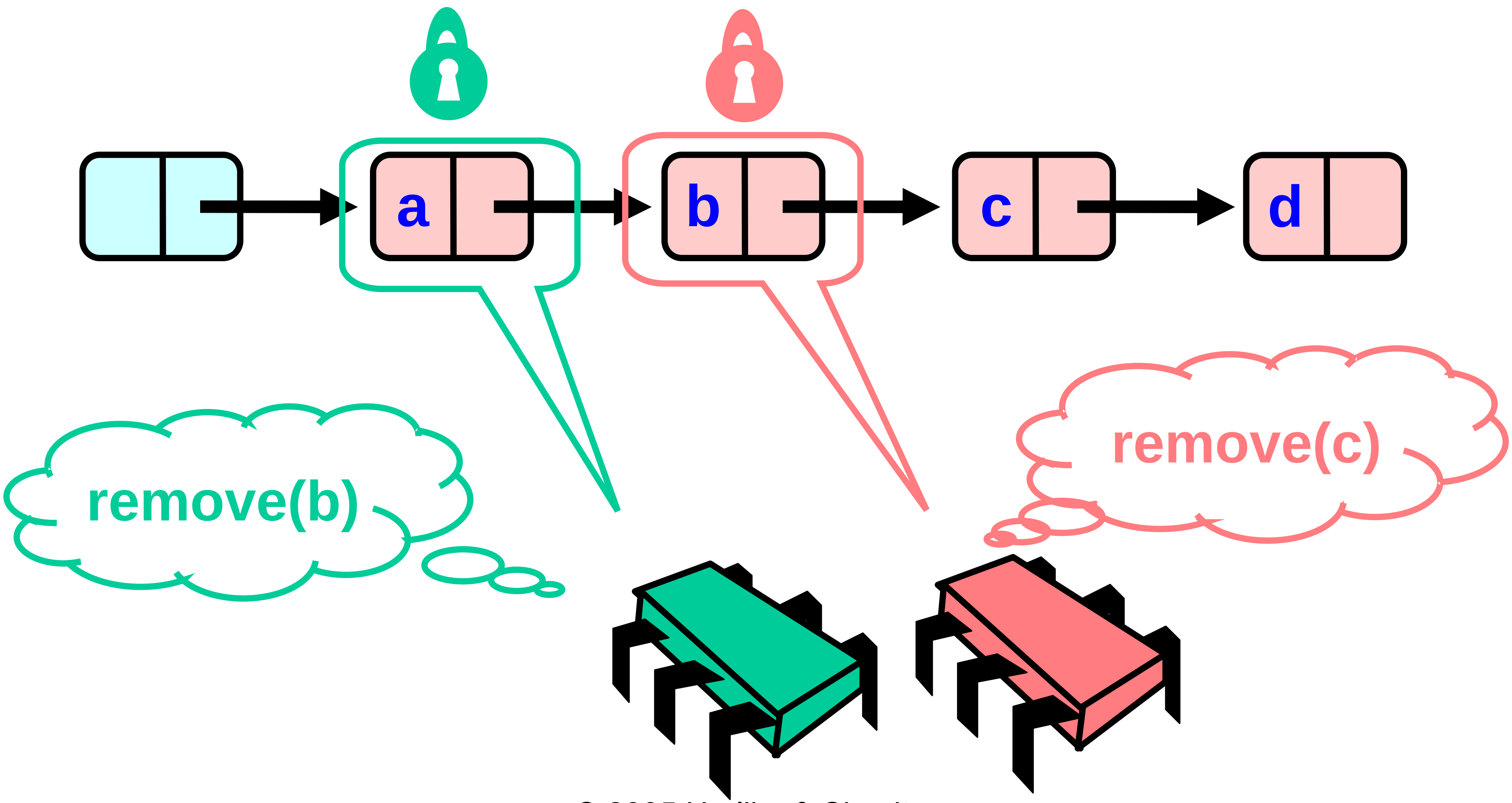


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

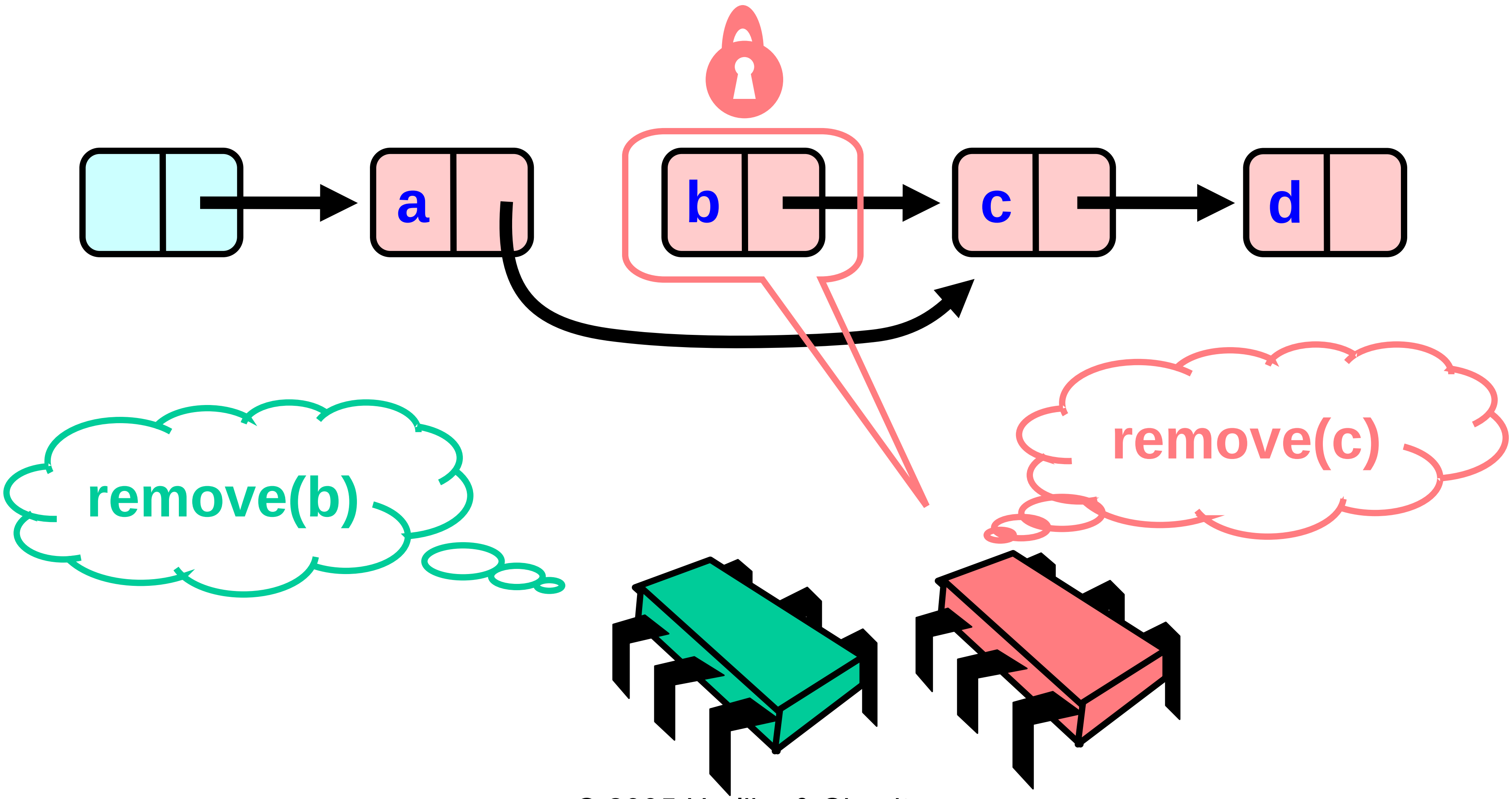
Removing Two Nodes



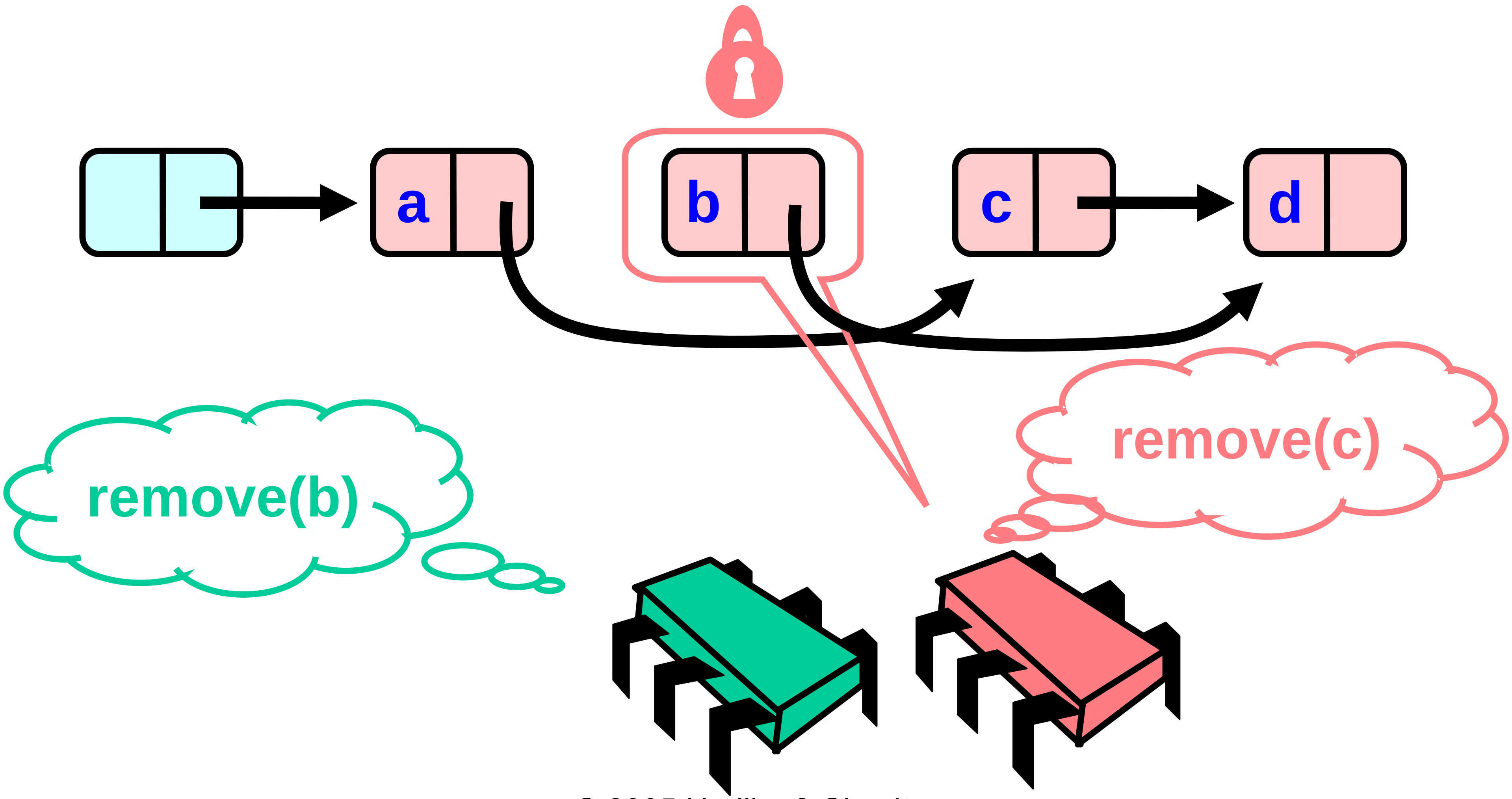
Removing Two Nodes



Removing Two Nodes



Removing Two Nodes



From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Hand-over-hand locking

- add(x)
  - Lock hand-over-hand.
  - When adding new node, keep both predecessor and successor locked (HS Fig. 9.6).
  - We could actually release the lock on the successor before adding the new node.
- contains(x)
  - Lock hand-over-hand, can unlock everything before reading curr.key.

Hand-over-hand locking

```
S.add(x)
  pred := S.head
  pred.lock()
  curr := pred.next
  curr.lock()
  while (curr.key < x)
    pred.unlock()
    pred := curr
    curr := pred.next
    curr.lock()
  if curr.key = x then
    pred.unlock()
    curr.unlock()
    return false
  else
    node := new Node(x)
    node.next := curr
    pred.next := node
    pred.unlock()
    curr.unlock()
    return true
```

```
S.remove(x)
  pred := S.head
  pred.lock()
  curr := pred.next
  curr.lock()
  while (curr.key < x)
    pred.unlock()
    pred := curr
    curr := pred.next
    curr.lock()
  if curr.key = x then
    pred.next := curr.next
    pred.unlock()
    curr.unlock()
    return true
  else
    pred.unlock()
    curr.unlock()
    return false
```

```
S.contains(x)
  curr := S.head
  curr.lock()
  while (curr.key < x)
    temp := curr
    curr := curr.next
    curr.lock()
    temp.unlock()
  curr.unlock()
  if curr.key = x then
    return true
  else
    return false
```

Correctness

- Atomicity:
  - Similar to coarse-grained locking.
  - Forward simulation to canonical atomic set object: S = elements that are reachable in the list.
  - "perform" steps:
    - add(x):
      - If successful, then when pred.next := node.
      - Else any time the lock on the node already containing x is held.
    - remove(x):
      - If successful, then when pred.next := curr.next
      - Else any time the lock on the node seen to have a higher key is held.
    - contains(x): LTTR
      - If true, then any time the lock on the node containing x is held.
      - Else any time the lock on the node seen to have a higher key is held.
  - Invariant: Any locked node is reachable in the list.

Correctness

- Atomicity:
  - Forward simulation to canonical atomic set object:
    - S = elements that are reachable in the list.
- Liveness:
  - Guarantees progress, assuming that the locks do.
  - Guarantees lockout-freedom, assuming the locks do.
    - All processes compete for locks in the same order.
  - Blocking (not wait-free or lock-free).

Evaluation

- Problems:
  - Each operation must acquire O(|S|) locks.
  - Pipelining means that fast threads can get stuck behind slow threads.
  - Using reader/writer locks might help performance, but introduces new deadlock possibilities.
- Idea:
  - Can we examine the nodes first without locking, and then lock only the nodes we need?
  - Must ensure that the node we modify is still in list.
  - Optimistic locking.

Optimistic locking

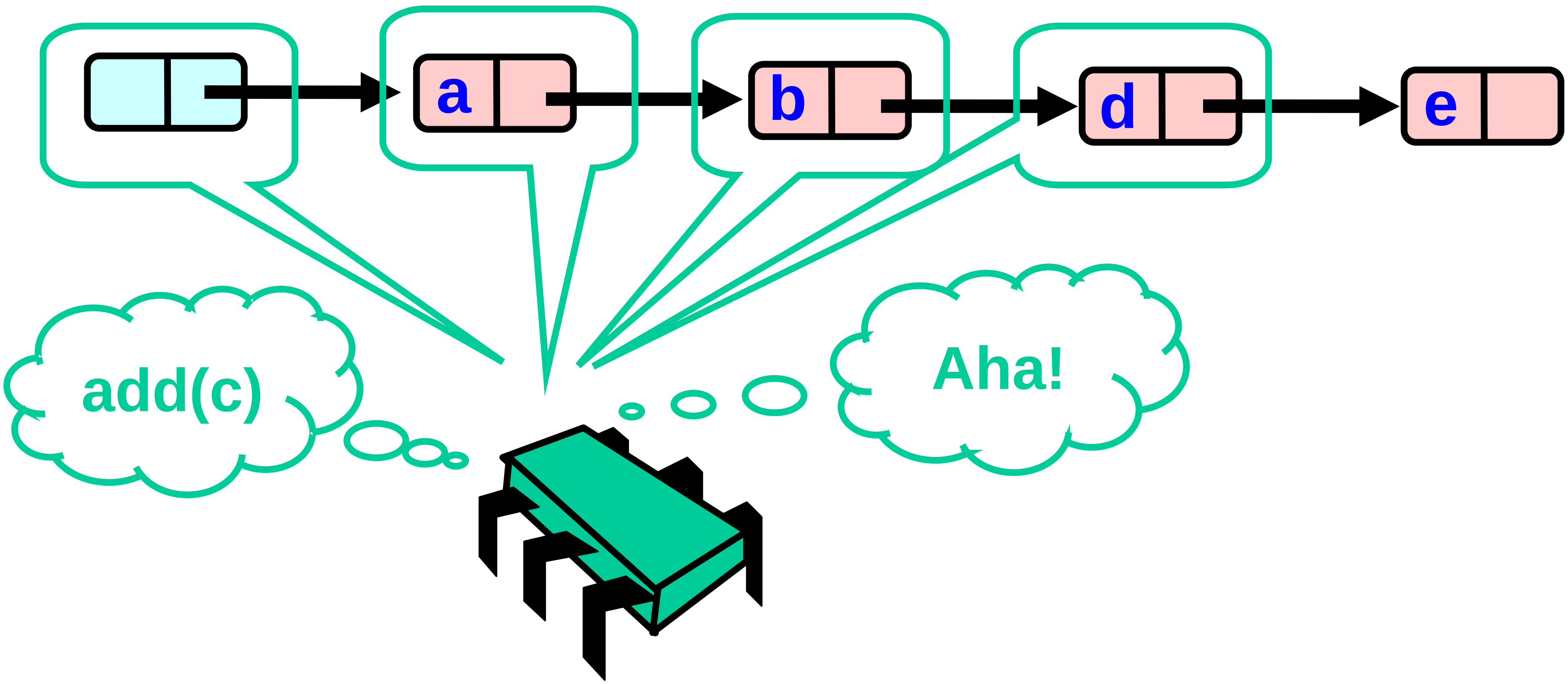
- Examine the nodes first without locking.
- Lock the nodes we need.
- Verify that the locked nodes are still in the list, before making modifications or determining results

Optimistic locking

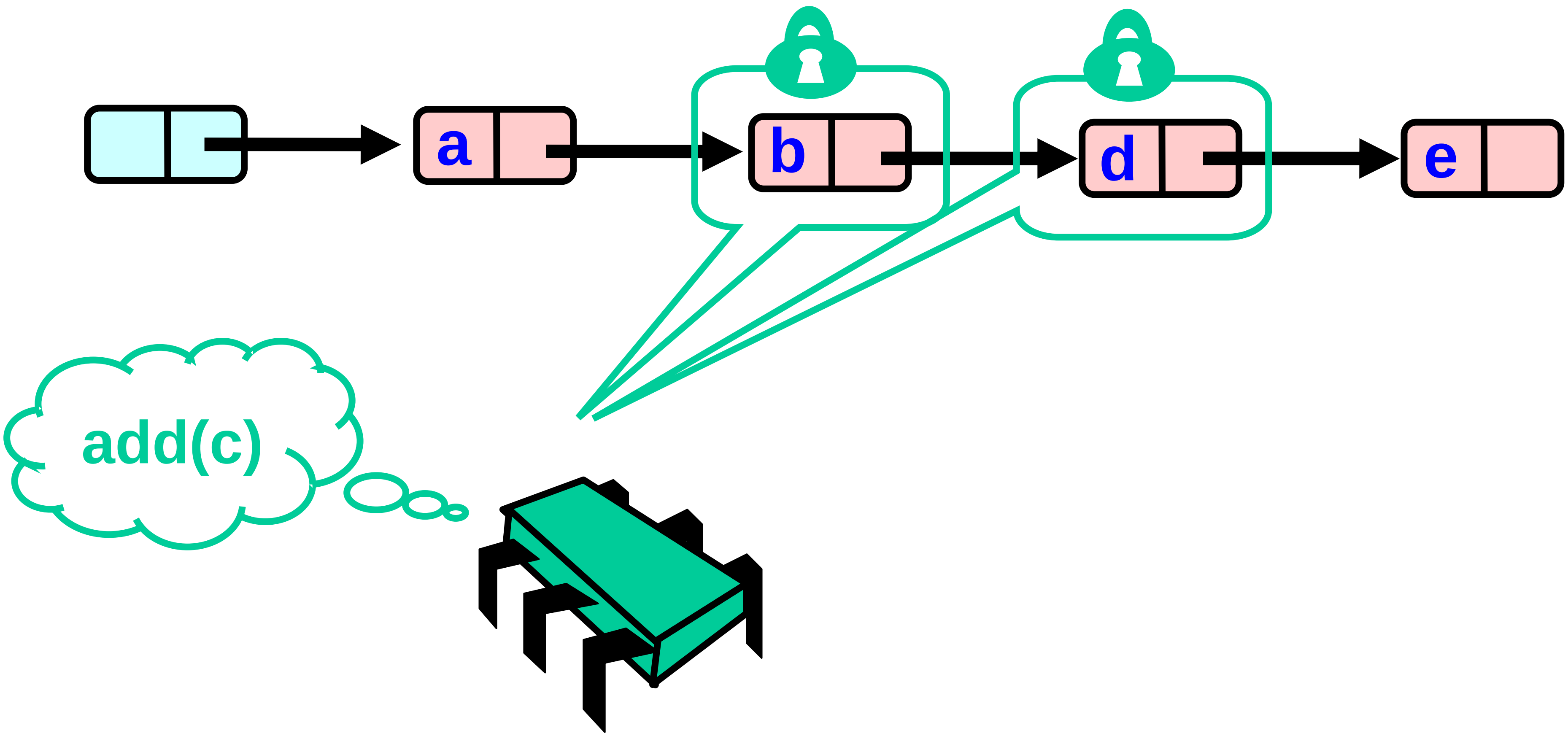
- add(x):
  - Traverse the list from the head, without locking, looking for the nodes we need (pred and curr).
  - Lock nodes pred and curr.
  - Validate that pred and curr are still in the list, and are still consecutive (pred.next = curr), by traversing the list once again.
  - If this works, then add the node and return true (or return false if it's already there).
  - If it doesn't work, start over.
- remove(x), contains(x): Similar.

- Better than hand-over-hand if
  - Traversing twice without locking is cheaper than once with locking.
  - Validation usually succeeds

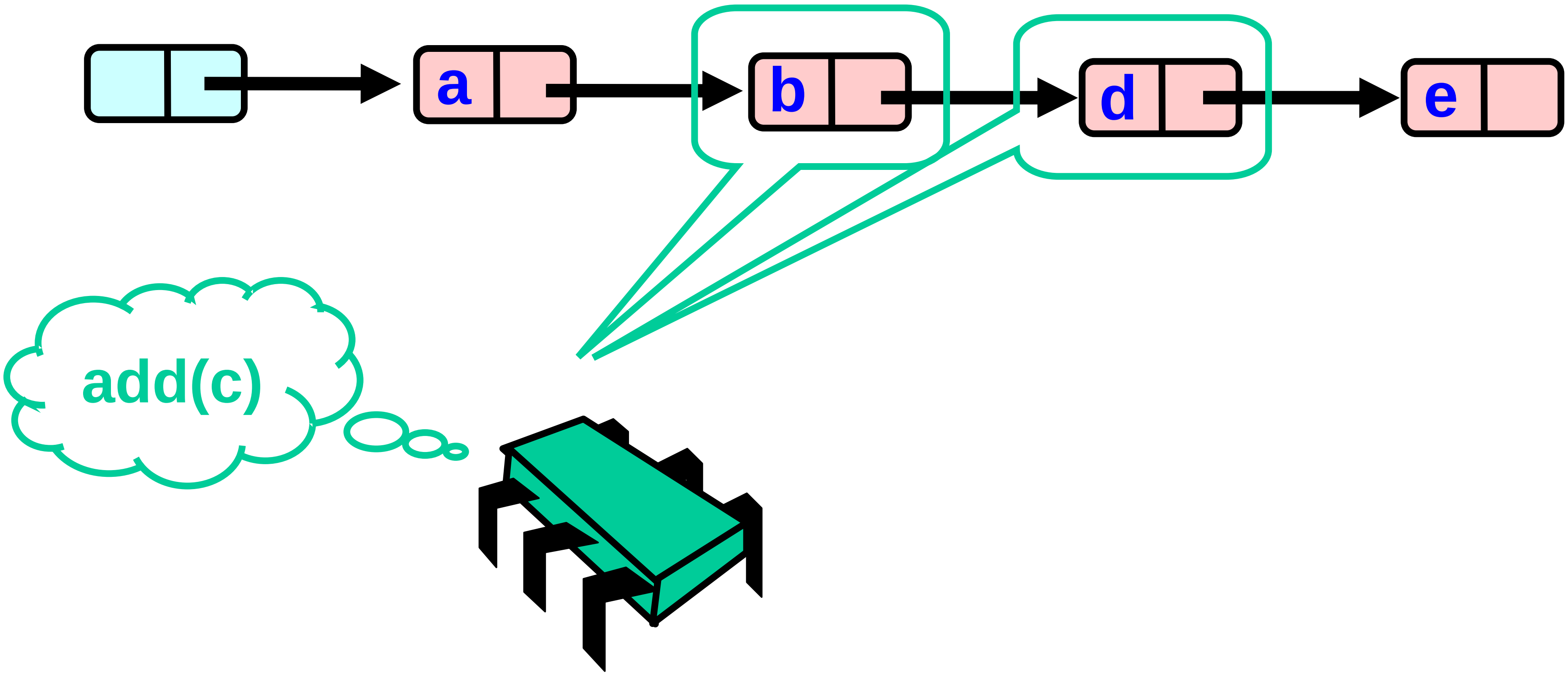
Optimistic locking



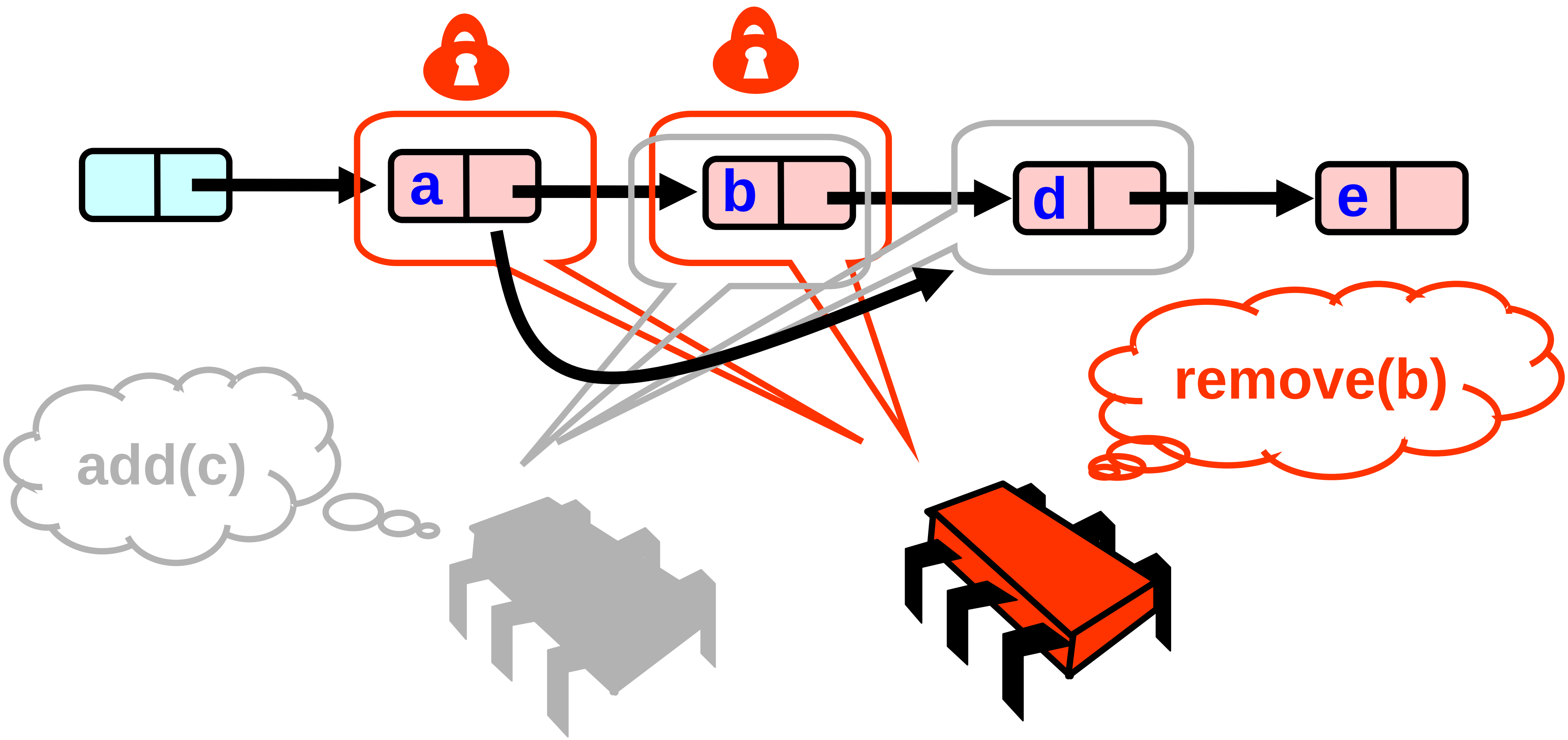
Optimistic locking



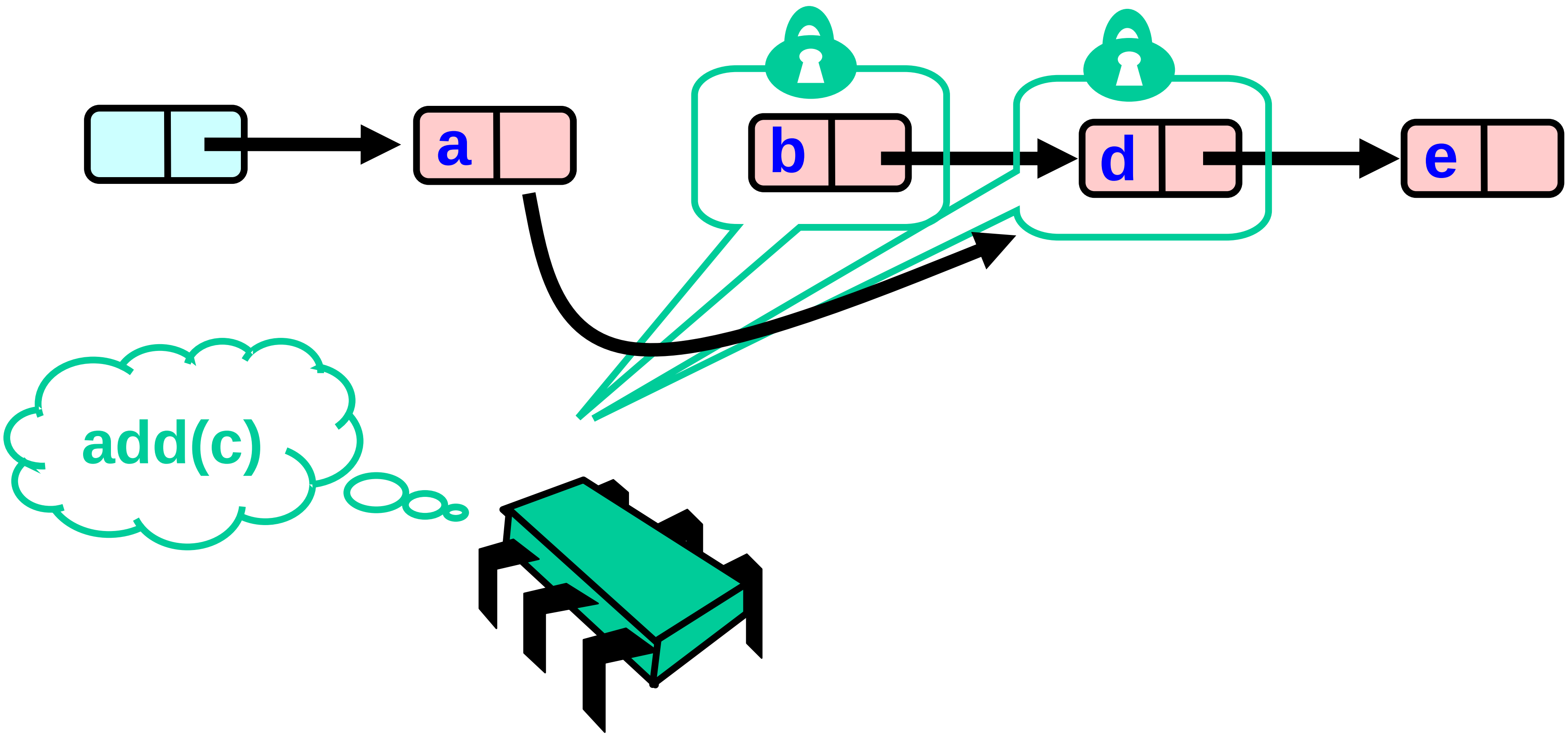
What can go wrong? (Part 1)



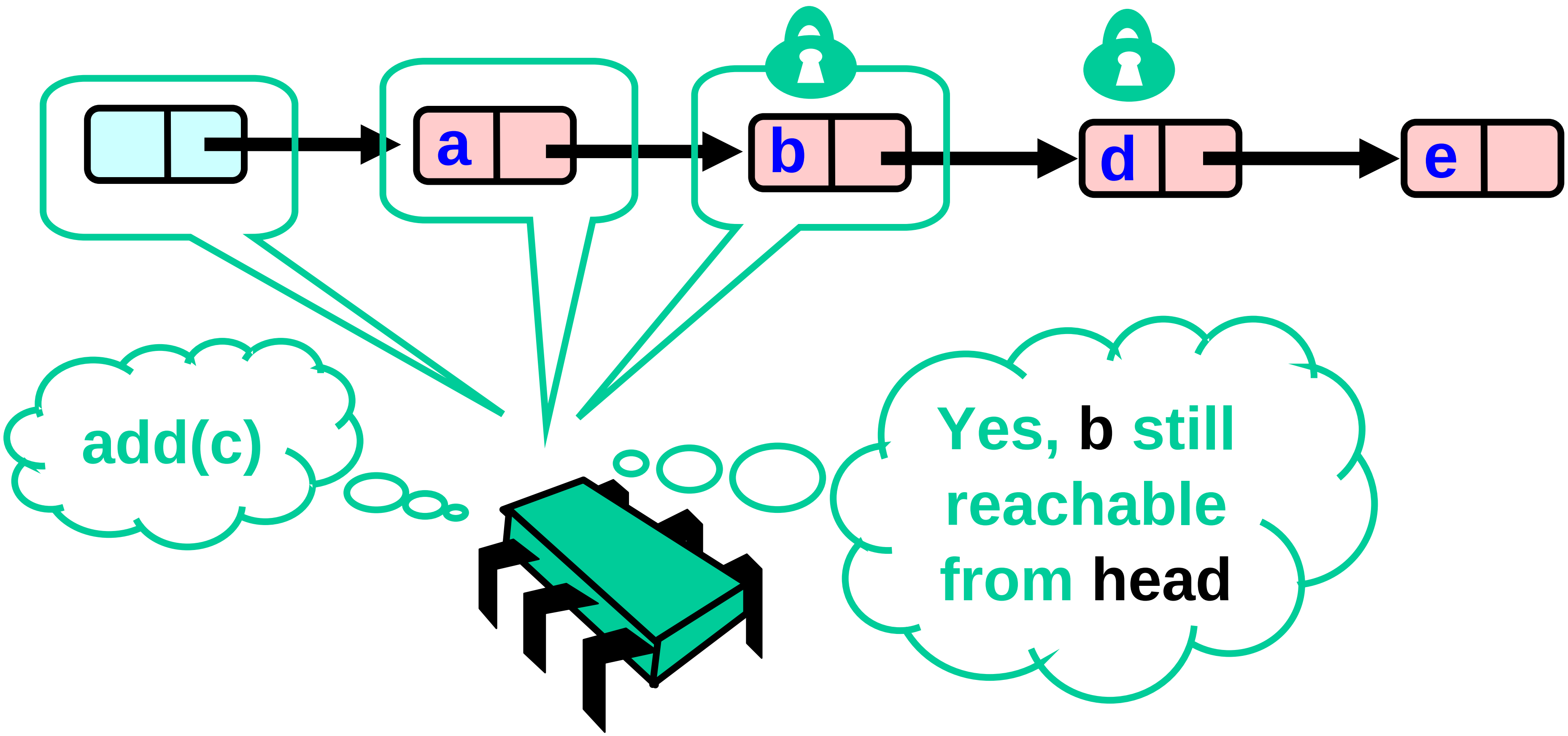
What can go wrong? (Part 1)



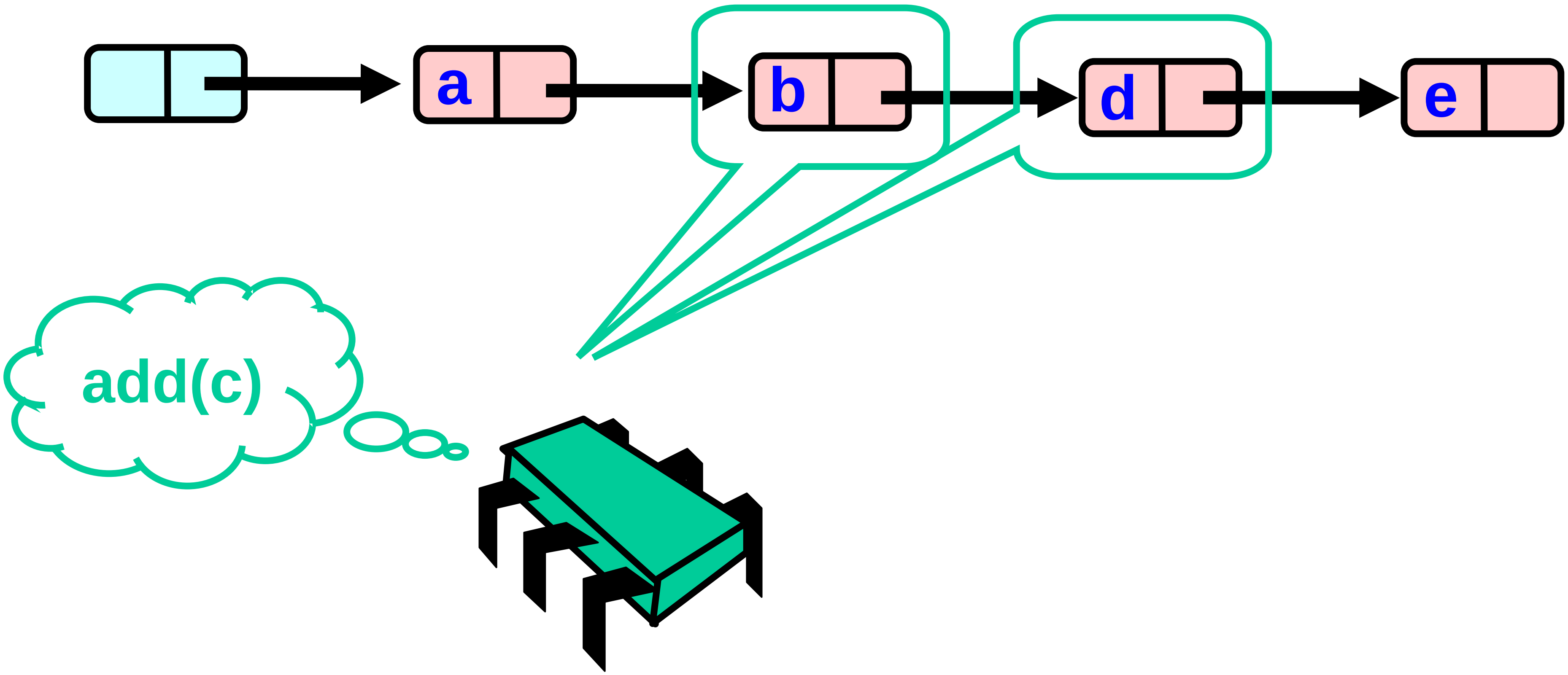
What can go wrong? (Part 1)



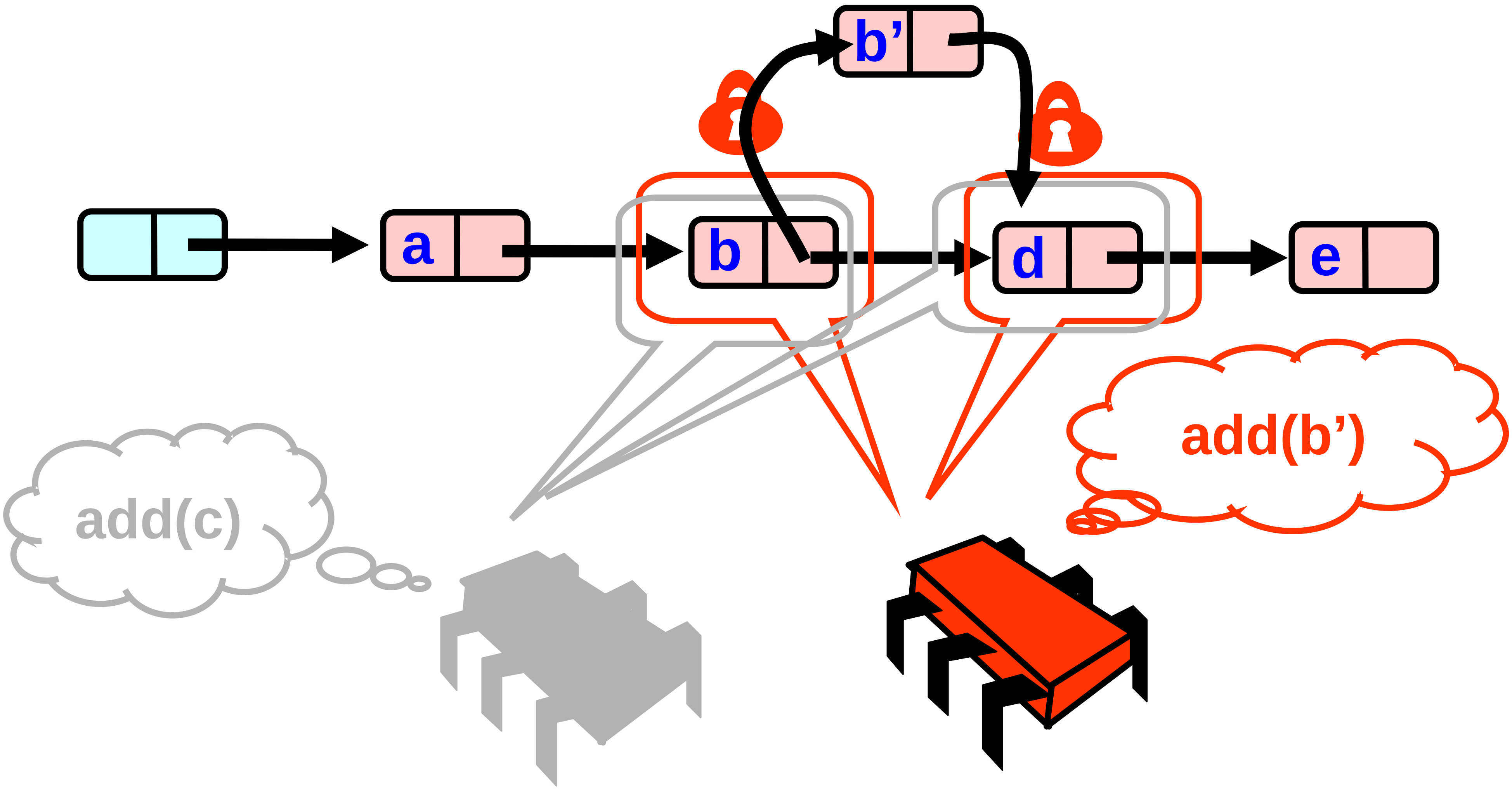
Validate (Part 1)



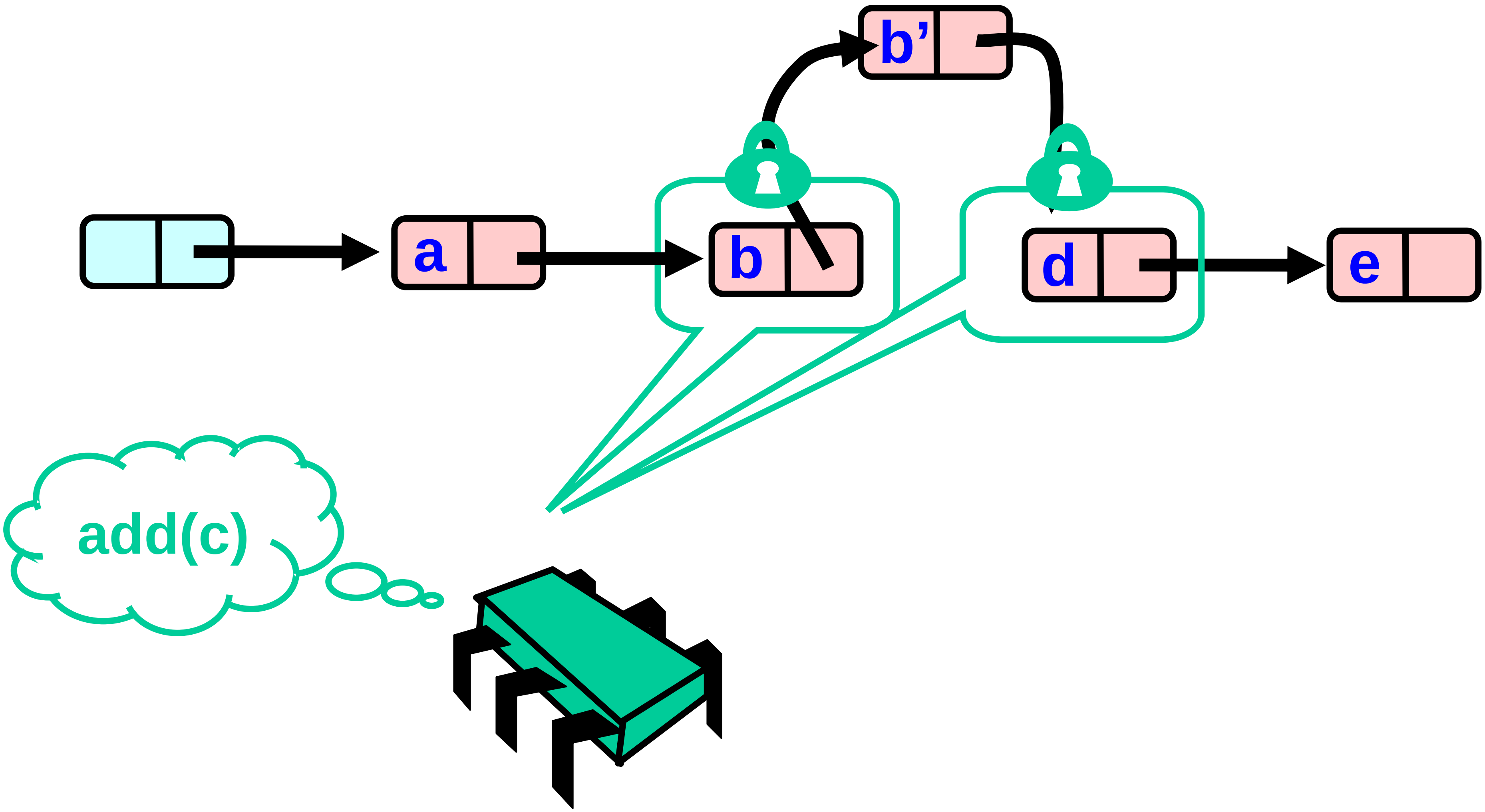
What can go wrong? (Part 2)



What can go wrong? (Part 2)

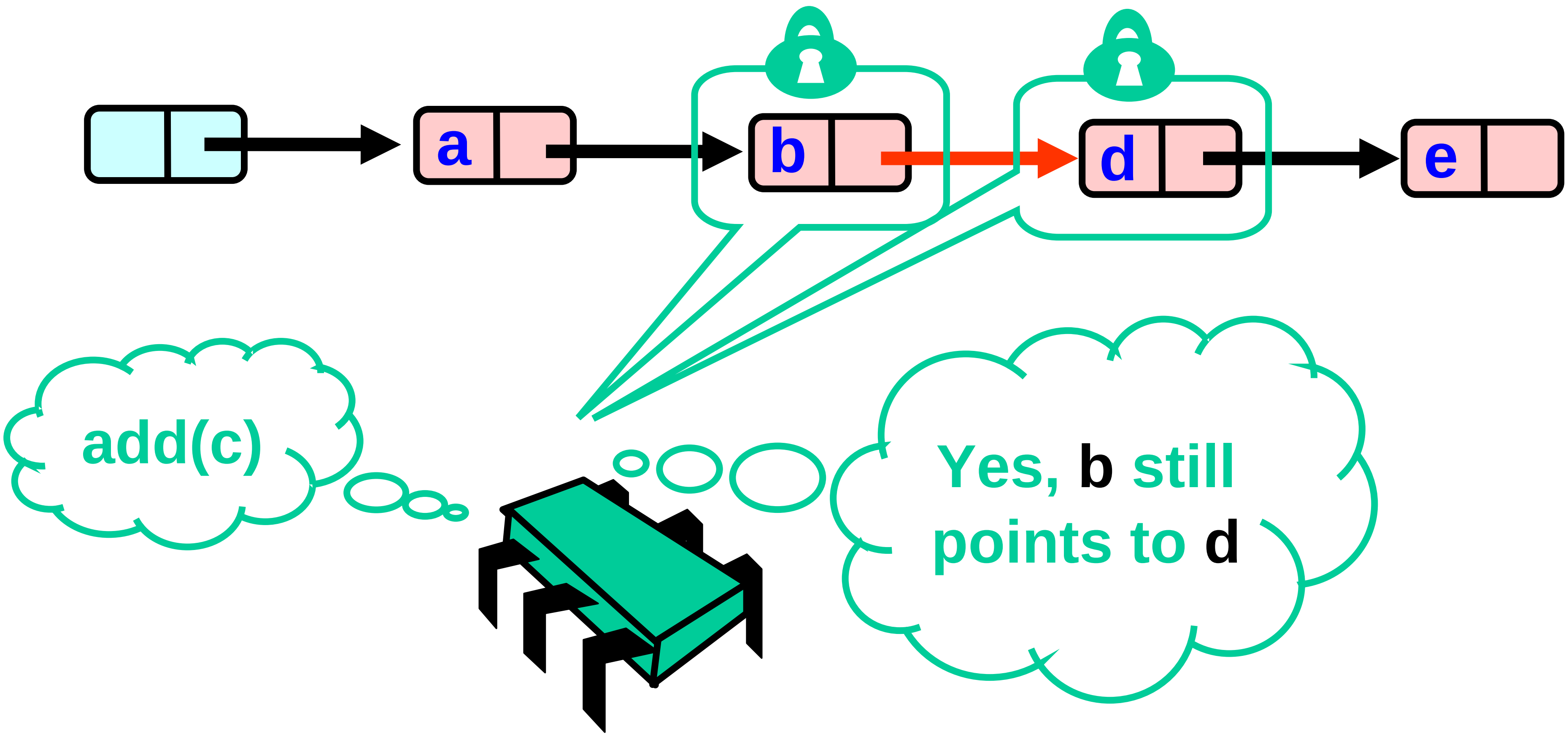


What can go wrong? (Part 2)

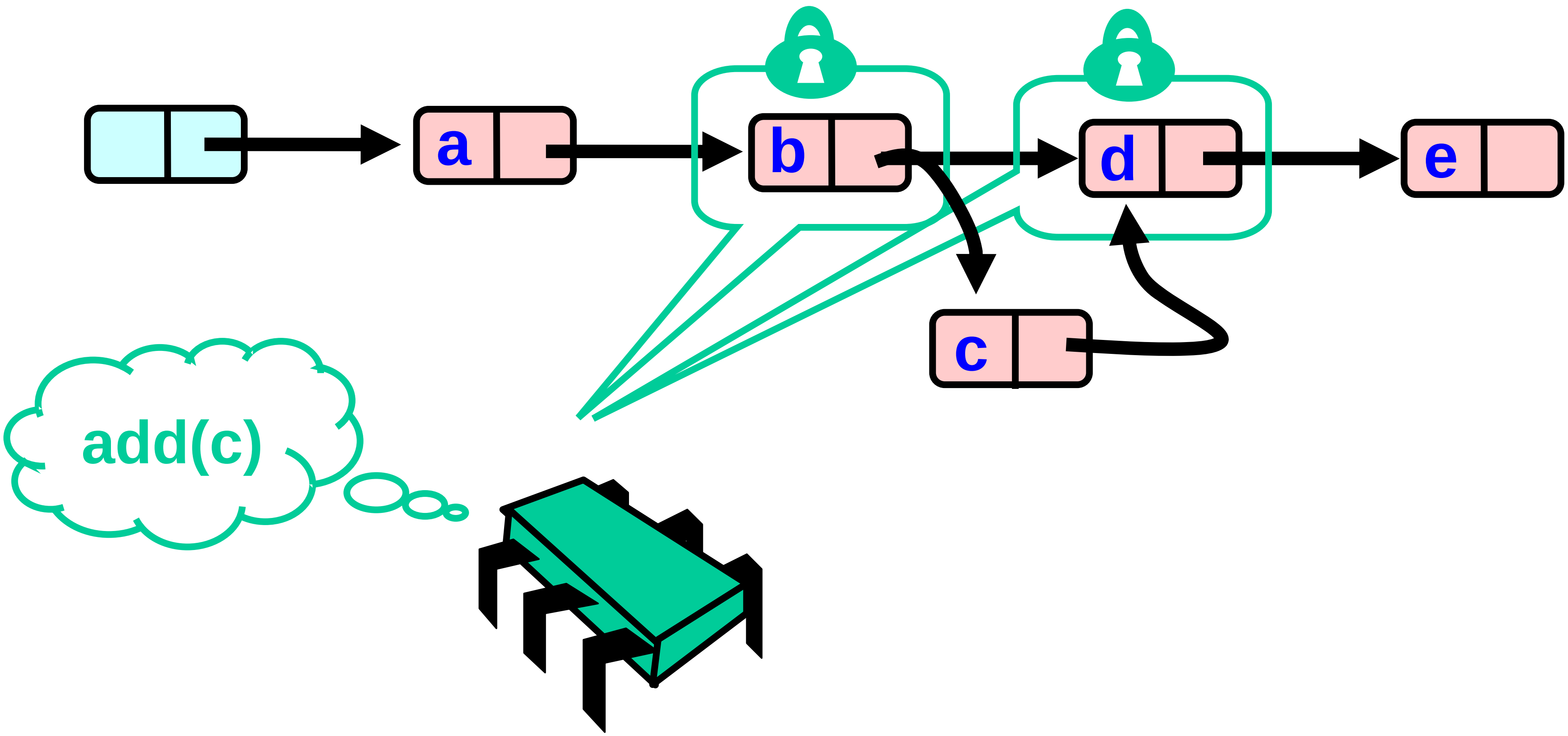


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Validate (Part 2)



Optimistic locking



Correctness

- Atomicity: Similar to hand-over-hand locking.
  - Forward simulation to canonical atomic set object:
    - S = elements that are reachable in the list.
  - “perform” steps: As for hand-over-hand locking, but consider only the last attempt (for which validation succeeds).
- Liveness:
  - Guarantees progress, assuming the locks do.
  - Does not guarantee lockout-freedom (even if locks do).
  - Blocking (not wait-free or lock-free).

Evaluation

- Works well if lock-free traversal is fast, and contention is infrequent.
- Problems:
  - Repeated traversals.
  - Need to acquire locks.
    - Even contains() needs locks.
- Locks can cause problems:
  - Some operations take 1000x (or more) longer than others, due to page faults, descheduling, etc.
  - If this happens to anyone holding a lock, everyone else who wants to access that lock must wait.

- Q: Can we avoid locks?

Lock-free algorithm

- Avoids locks/blocking entirely.
- Instead, separates logical vs. physical node removal, marking nodes before deleting them.
- Operations help other operations by deleting marked nodes.

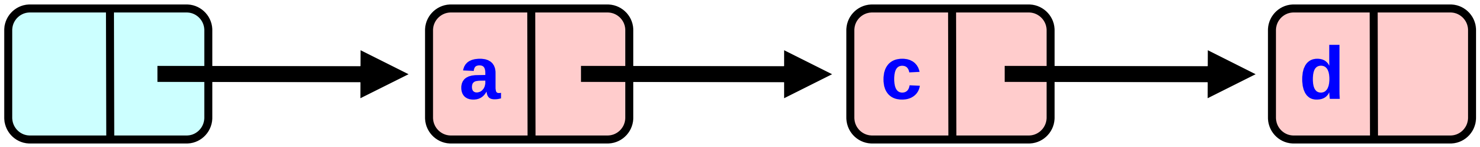
Lock-freedom

- If any process executing an operation does not stop then some operation completes.
- Weaker than wait-free: lockout is possible.
- Rules out a delayed process from blocking other processes indefinitely, and so, no locks.

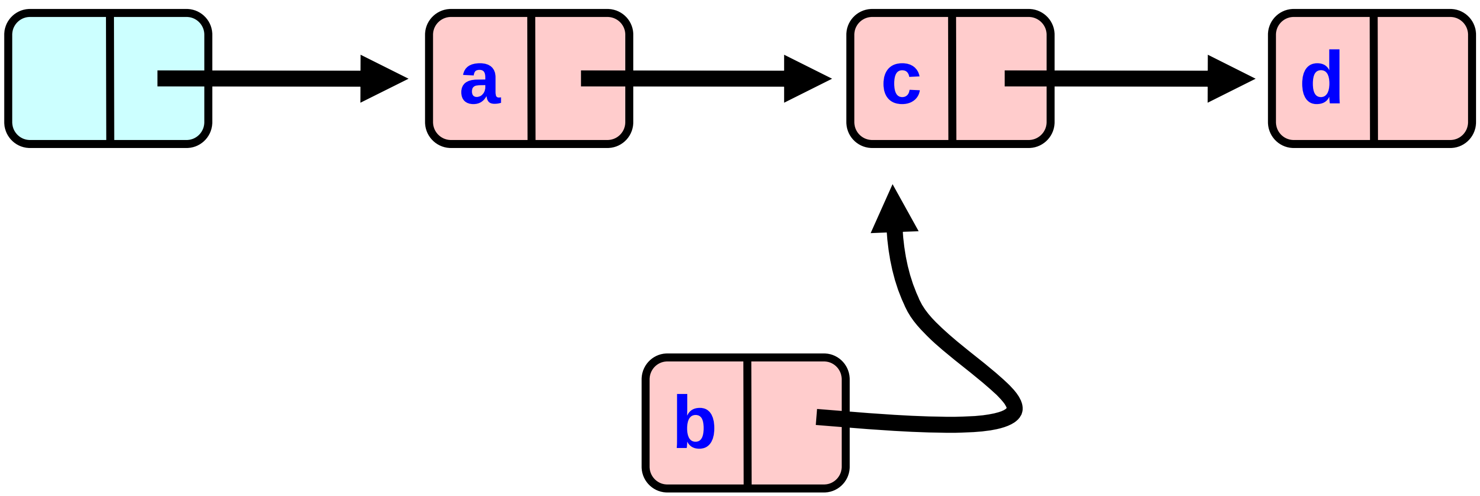
Lock-free list-based set

- Idea: Use CAS to change pred.next pointer.
- Make sure pred.next pointer hasn't changed since you read it.

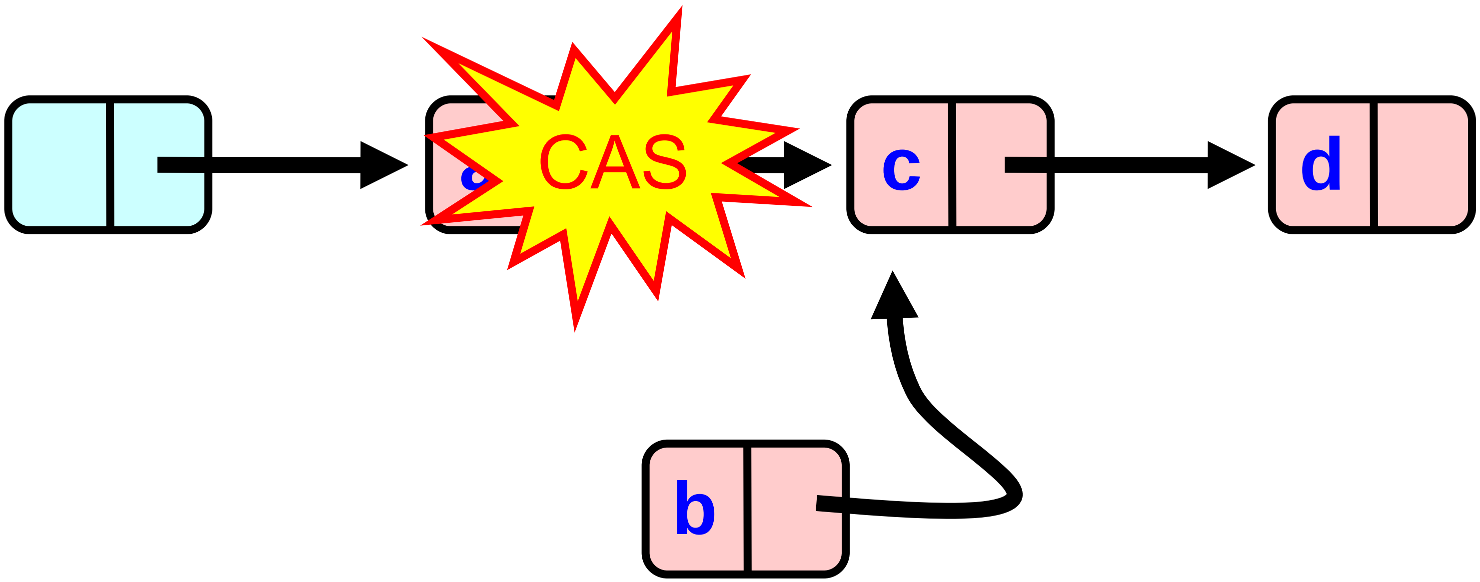
Adding a Node



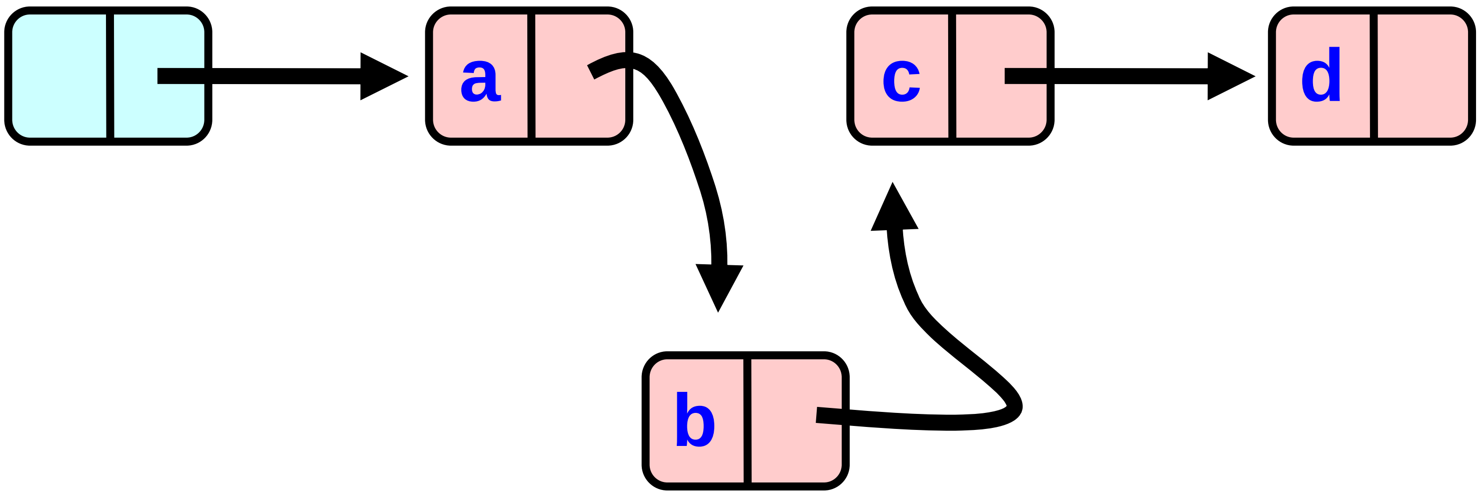
Adding a Node



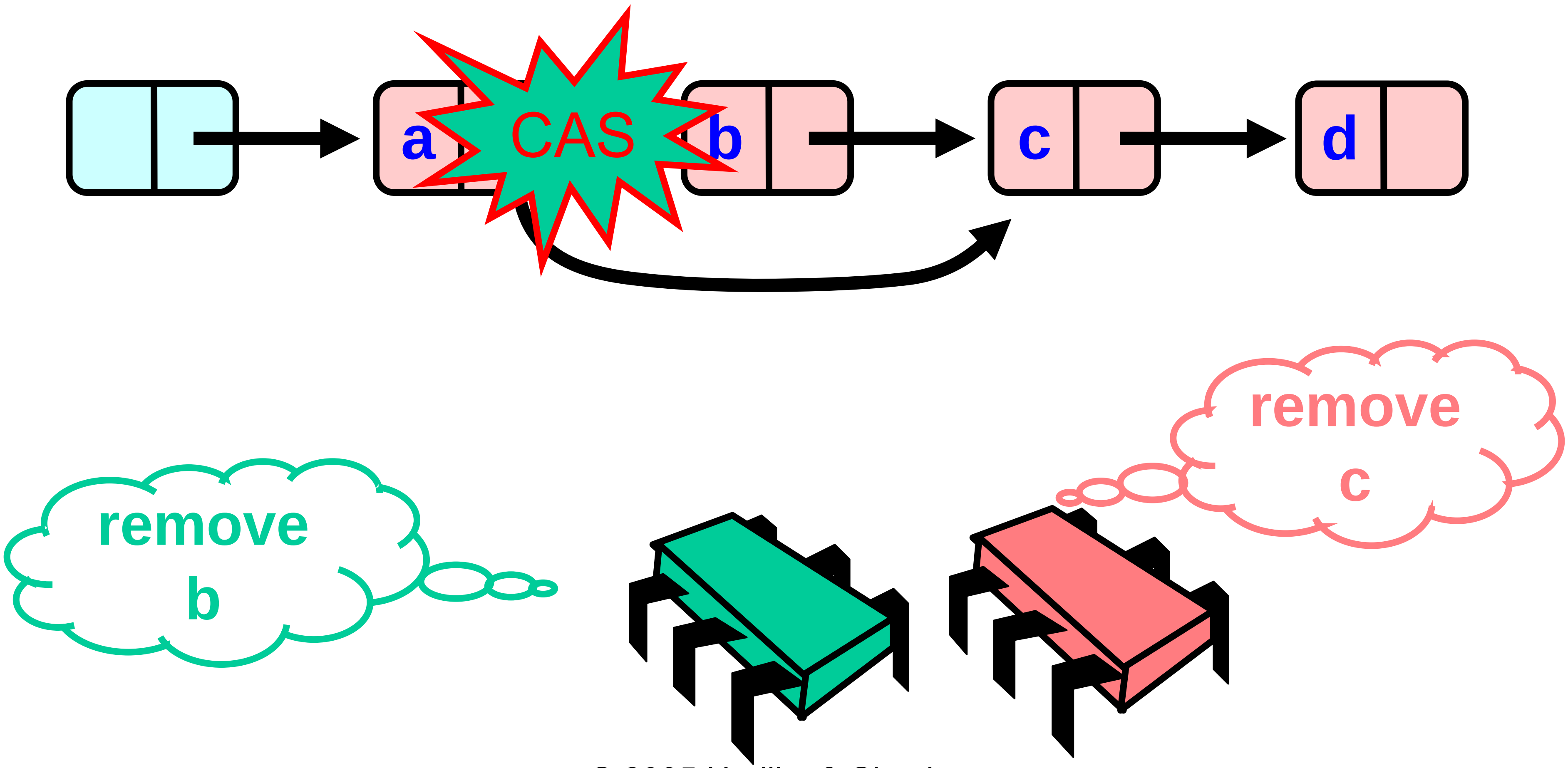
Adding a Node



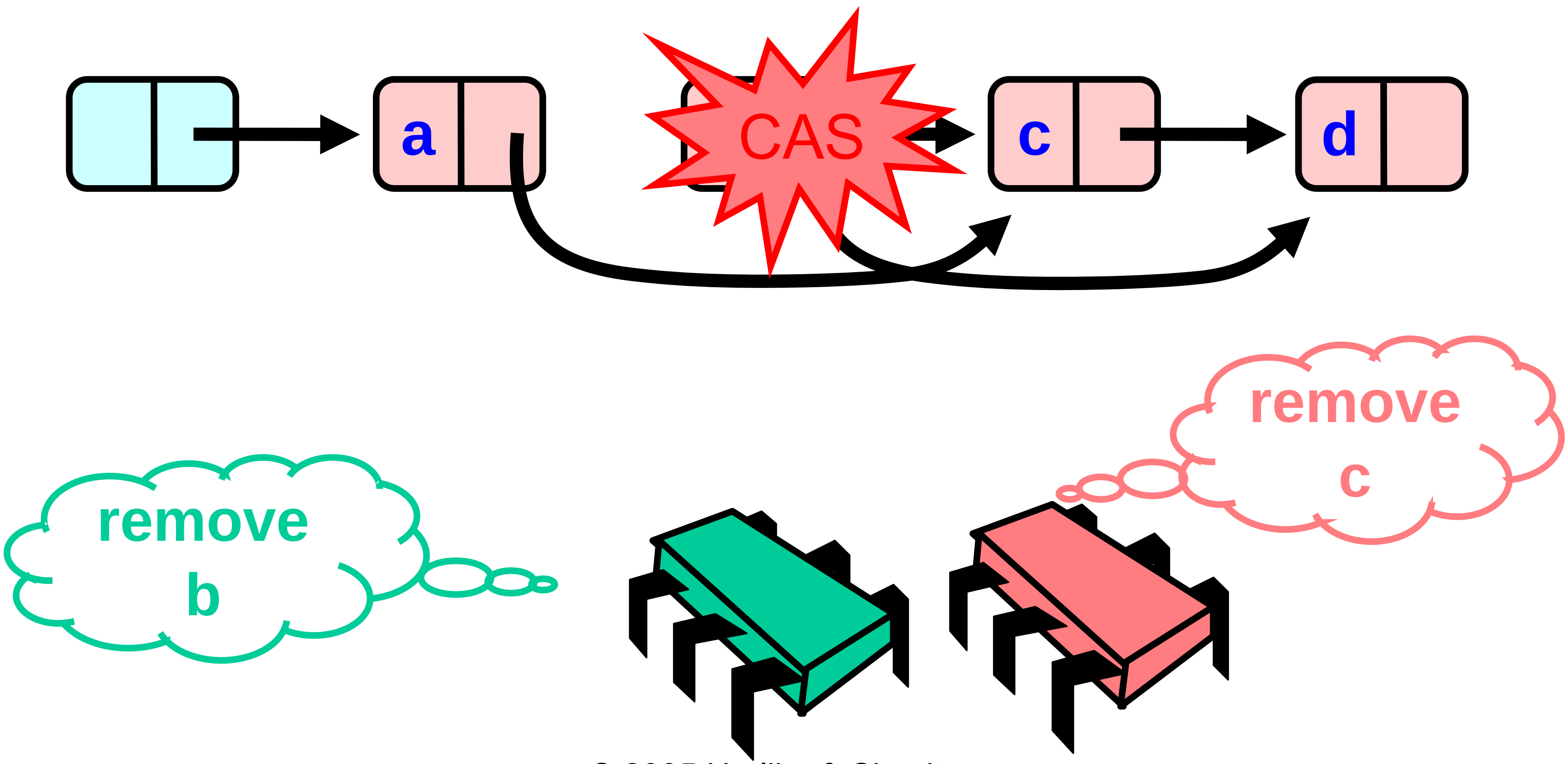
Adding a Node



Removing a Node

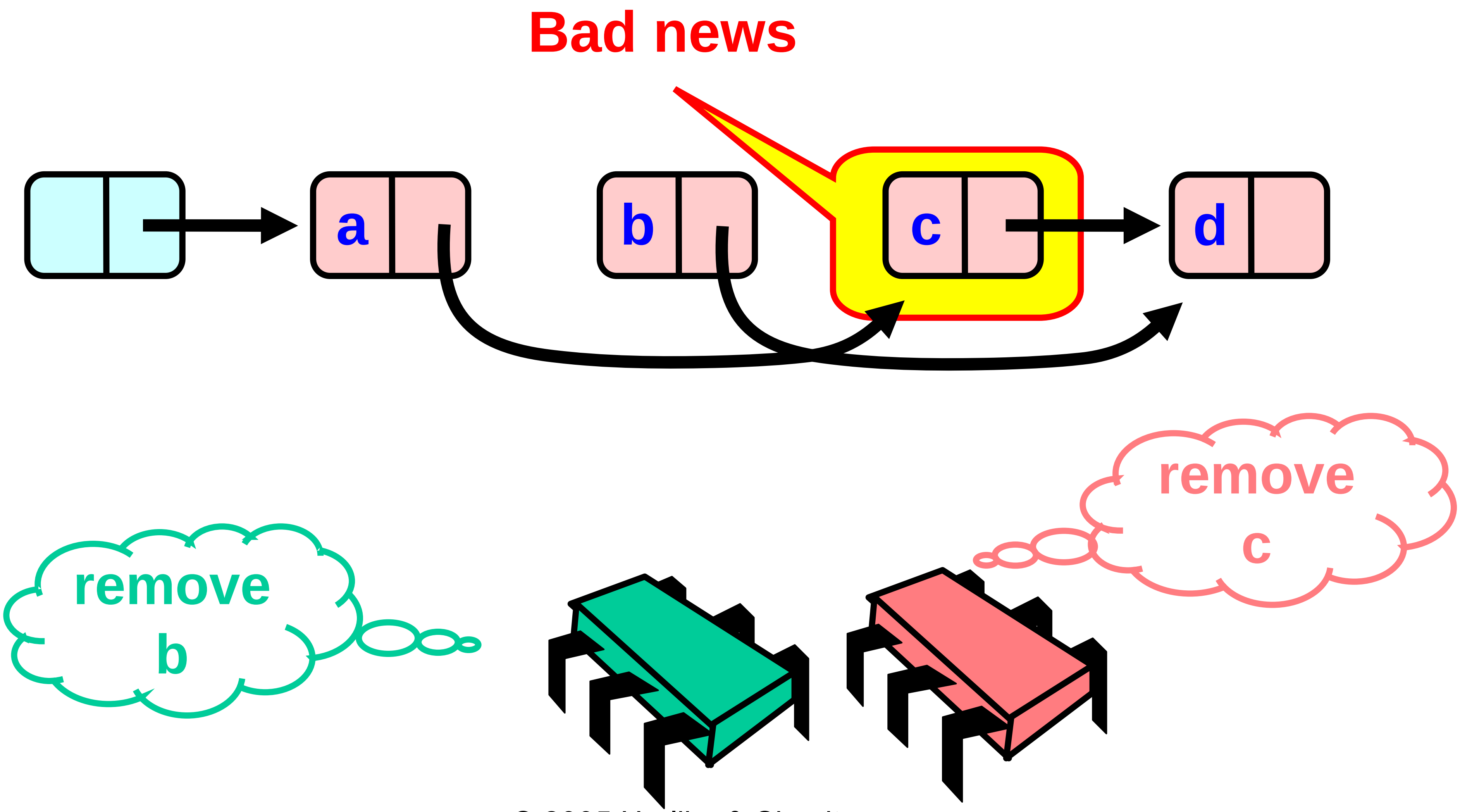


Removing a Node



From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Look Familiar?



Lock-free list-based set

- Idea: Add “mark” bit to a node to indicate whether its key has been removed from the abstract set S.
  - If mark = true, then node’s key is not in the set.
  - When a node is first added to the list, its mark = false.
  - Set mark := true before physically removing node from list by detaching its incoming pointer.
  - Setting the mark logically removes the node’s key from the set: It is the serialization point of a successful remove operation.

- Simulation relation:
  - S is the set of values in reachable nodes with mark = false.
- Don't change next pointer of a marked node.
  - Mark and next pointer must be in the same word, change atomically.
  - “Steal” a bit from pointers.
  - Jave class AtomicMarkableReference (in Java concurrency library) supports techniques like those in this algorithm.

Lock-free list-based set

- To perform any operation, traverse the list, through marked and unmarked nodes, to find needed nodes.
- If needed nodes are marked, retry the operation.
- If needed nodes are unmarked then operate as follows:
  - For contains(x) or unsuccessful add/remove(x), return appropriate value as usual based on whether curr.key = x
  - For successful add(x), CAS pred's (curr, false) to (node, false).
  - For successful remove(x),
    - Logical removal: CAS curr's (next, false) to (next, true)
    - Physical removal: CAS pred's (curr, false) to (curr.next, false)
  - If any CAS except for the physical remove fails, retry the operation.

Helping

- Whenever an operation encounters marked nodes during traversal, it helps:
- If curr is marked:
  - CAS pred's (curr, false) to (curr.next, false).
  - If this CAS fails (because next is no longer curr or mark is now true), then retry the operation.

- Such helping is characteristic of lock-free and wait-free algorithms (not all have it, but most do).

- See HS Section 9.8.

Lock-free list: Find subroutine

Returns (pred, curr) such that at some point during execution, the following held simultaneously: pred.next = (curr, false), curr.next.mark = false, and pred.key < x ≤ curr.key.

```
S.find(x)
retry:
  pred := S.head; curr := pred.next.ref
  while (curr.key < x or curr.next.mark) do
    if curr.next.mark then
      if CAS(pred.next, (curr, false), (curr.next.ref, false)) then curr := pred.next.ref
      else
        if pred.next.mark then goto retry
        else curr := pred.next.ref
    else // It must be that curr.key < x.
      pred : = curr; curr := pred.next.ref
  return (pred, curr)
```

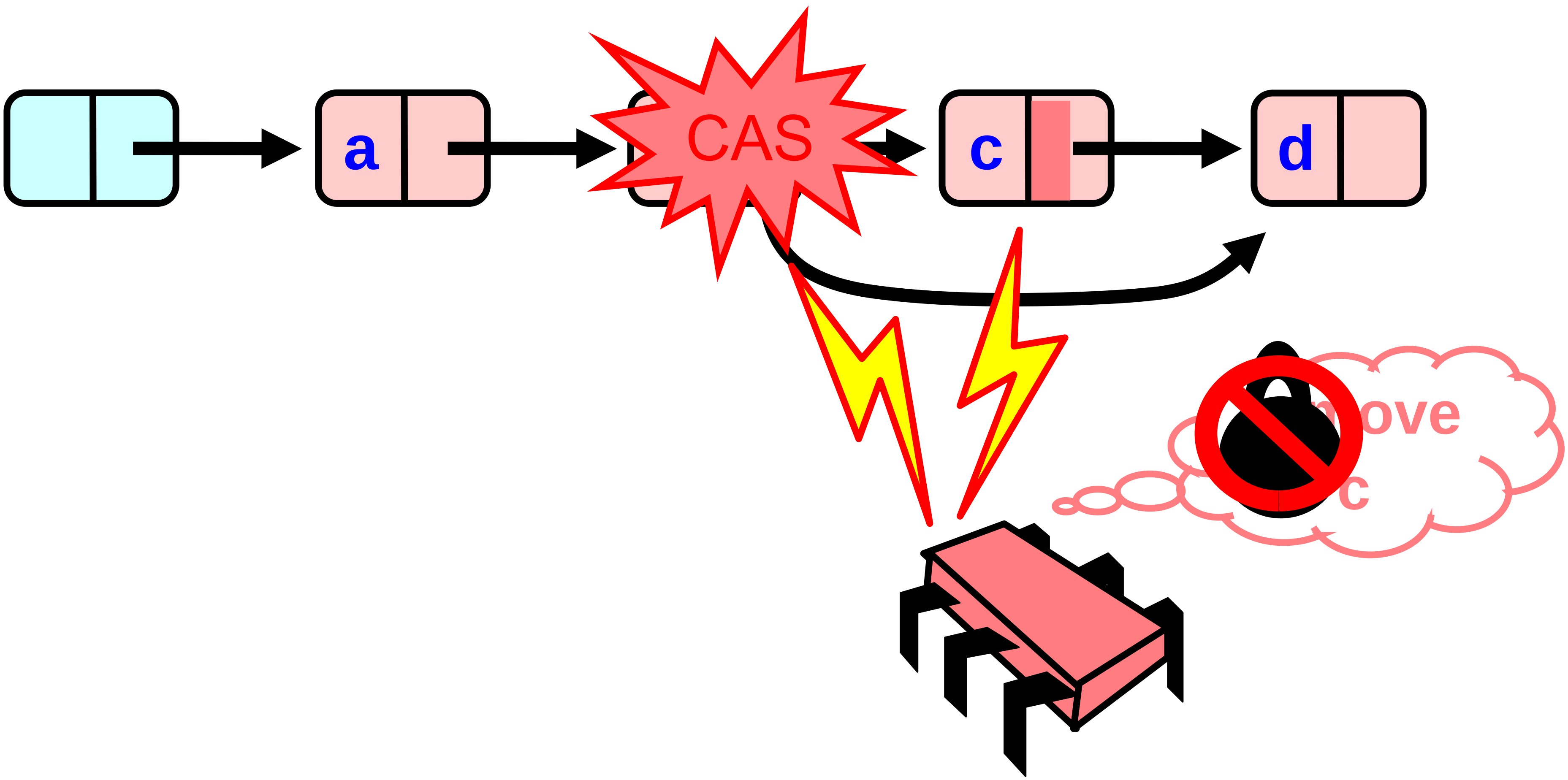
Lock-free list: Add

```
S.add(x)
retry:
    (pred, curr) := S.find(x)
    if curr.key = x then return false
    else
        node := new Node(x)
        node.next.ref := curr
        if CAS(pred.next, (curr, false), (node, false)) then return true
        else goto retry
```

Lock-free list: Remove and Contains

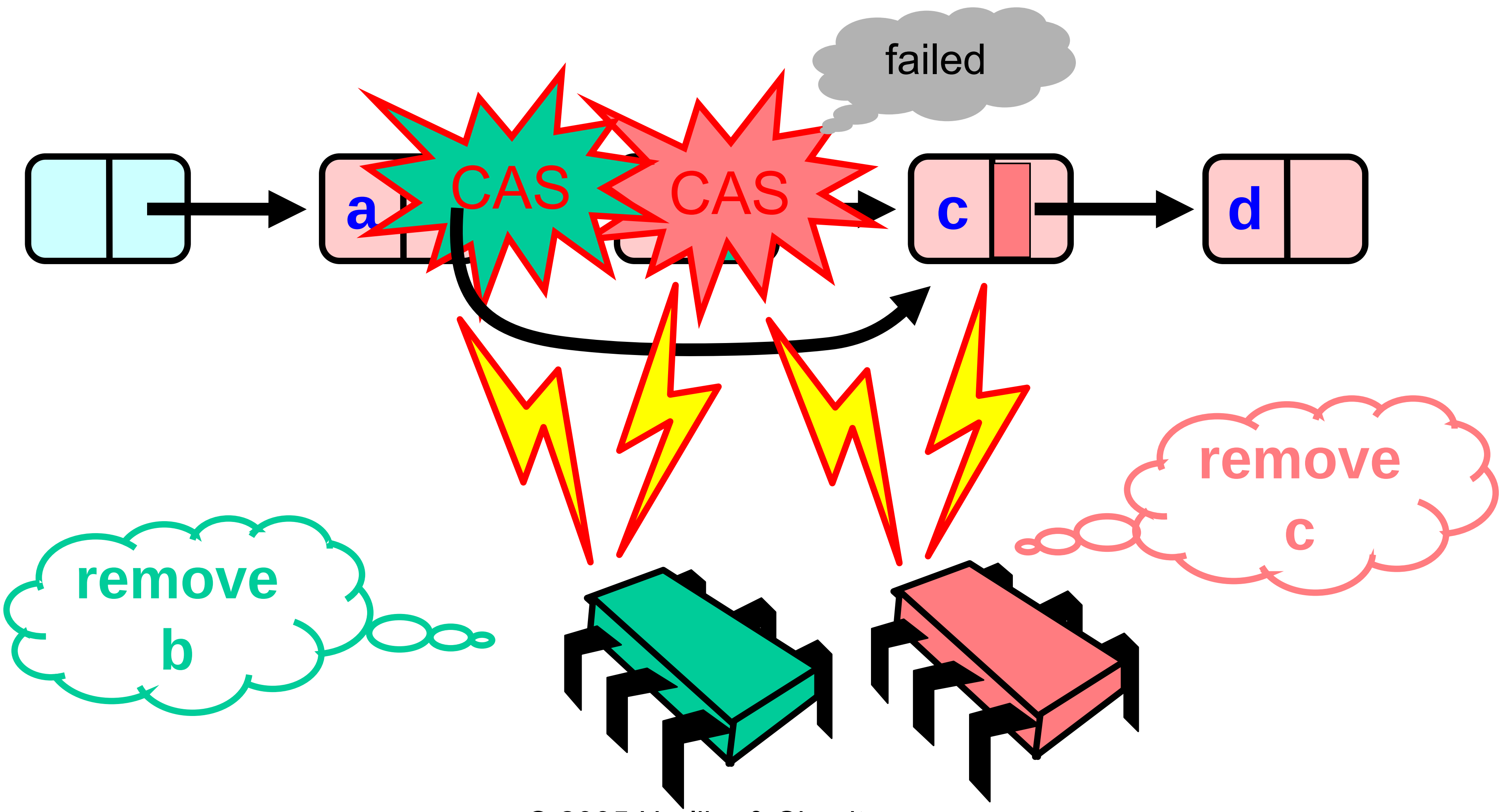
```
S.remove(x)
retry:
      (pred, curr) := S.find(x)
      if curr.key = x then
          next := curr.next.ref
          if CAS(curr.next, (next, false), (next, true)) then
              CAS(pred.next, (curr, false), (curr.next.ref, false))
              return true
          else goto retry
      else return false

S.contains(x)
    (pred, curr) := S.find(x)
    if curr.key = x then return true
    else return false
```

Removing a Node

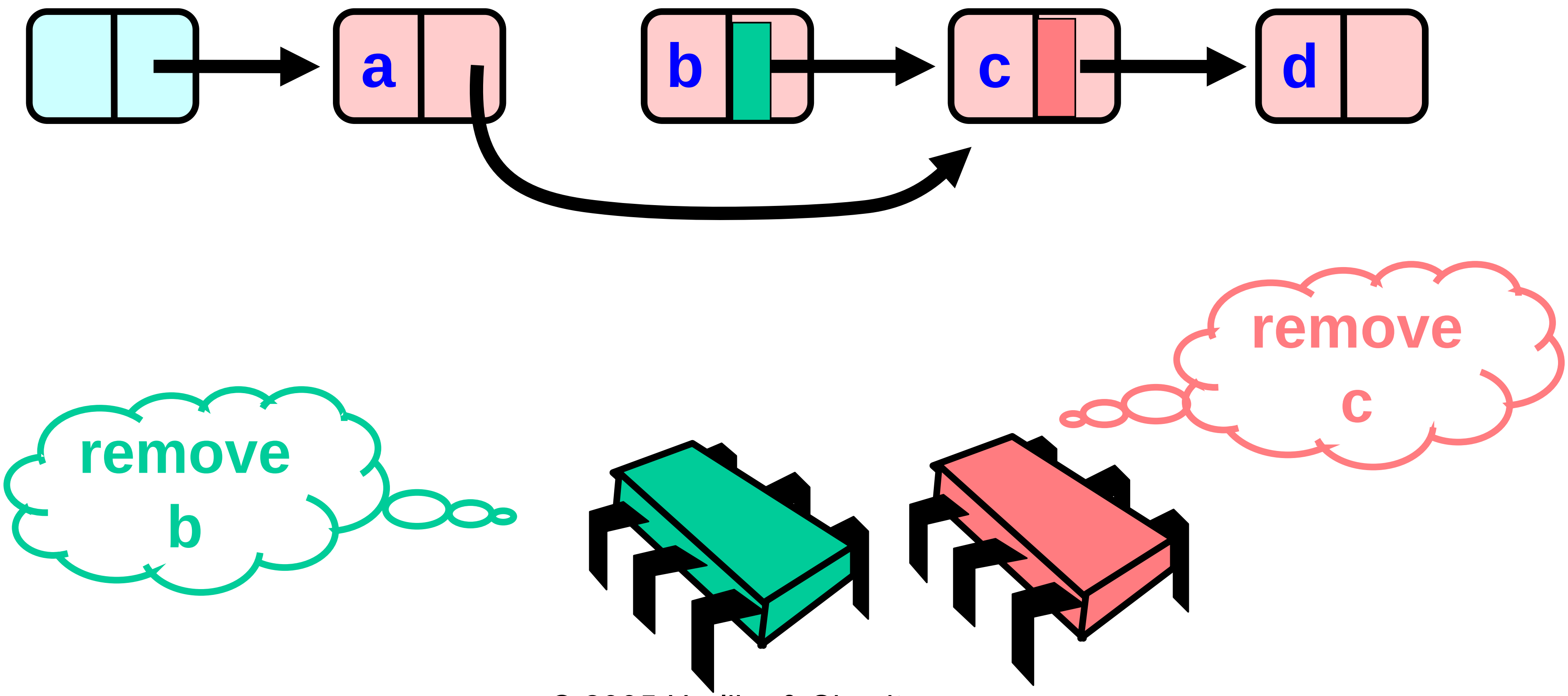


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

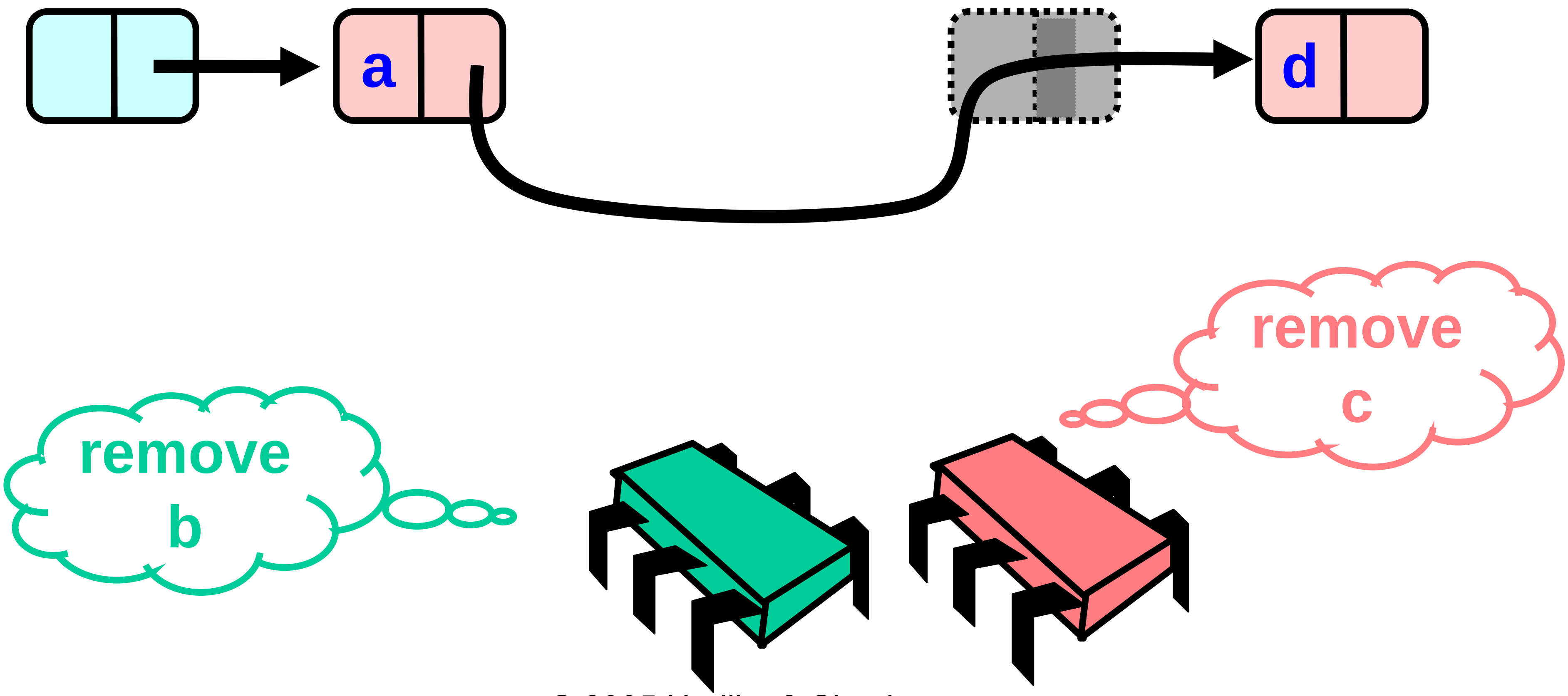
Removing a Node



Removing a Node



Removing a Node



Correctness

- Atomicity:
  - Forward simulation to canonical atomic set:
    - S = values in unmarked nodes that are reachable from the head via list pointers (through marked and unmarked nodes).
  - "perform" steps:
    - contains(x) or unsuccessful add(x) or remove(x): When curr is read from pred.next.
    - Successful add(x): When successful CAS sets pred.next := node.
    - Successful remove(x): When successful CAS marks node x (sets curr.mark := true).
  - Invariant: Any unmarked node encountered while traversing the list is reachable in the list.
- Liveness:
  - Nonblocking: lock-free
    - Operations may retry, but some must succeed.
  - Allows starvation (not lockout-free).

Evaluation

- No locks!
- Nonblocking, lock-free algorithm.
- But: Overhead for CAS and for helping.

Lazy algorithm

- Uses the marking trick as in the lock-free algorithm, removing nodes in two stages.
- Avoids CAS and helping.
- Instead, uses short-duration locks.

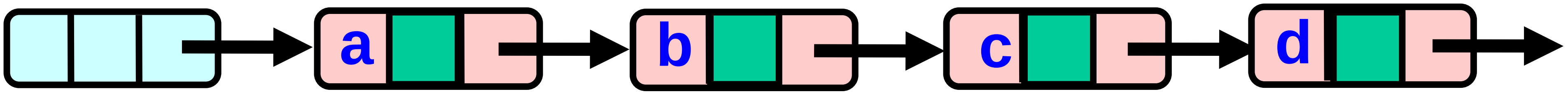
Lazy list algorithm

- Idea: Use mark as in lock-free list.
- "Lazy" removal: First mark node, then splice around it.
- Now mark can be separate from next pointer.
- No helping---assume each remove operation completes its own physical removal.

- Locks curr and pred nodes, with short-duration locks.
- Validation: Check locally that nodes are adjacent and unmarked; if not, retry the operation.

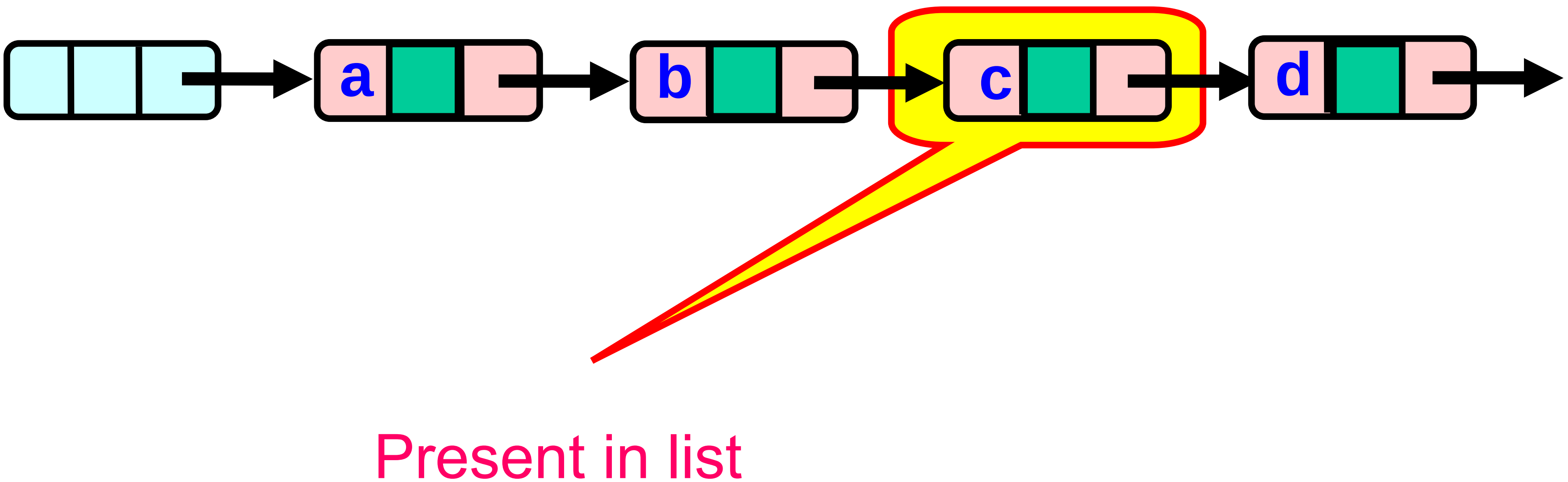
- See HS, Section 9.7.

Lazy Removal

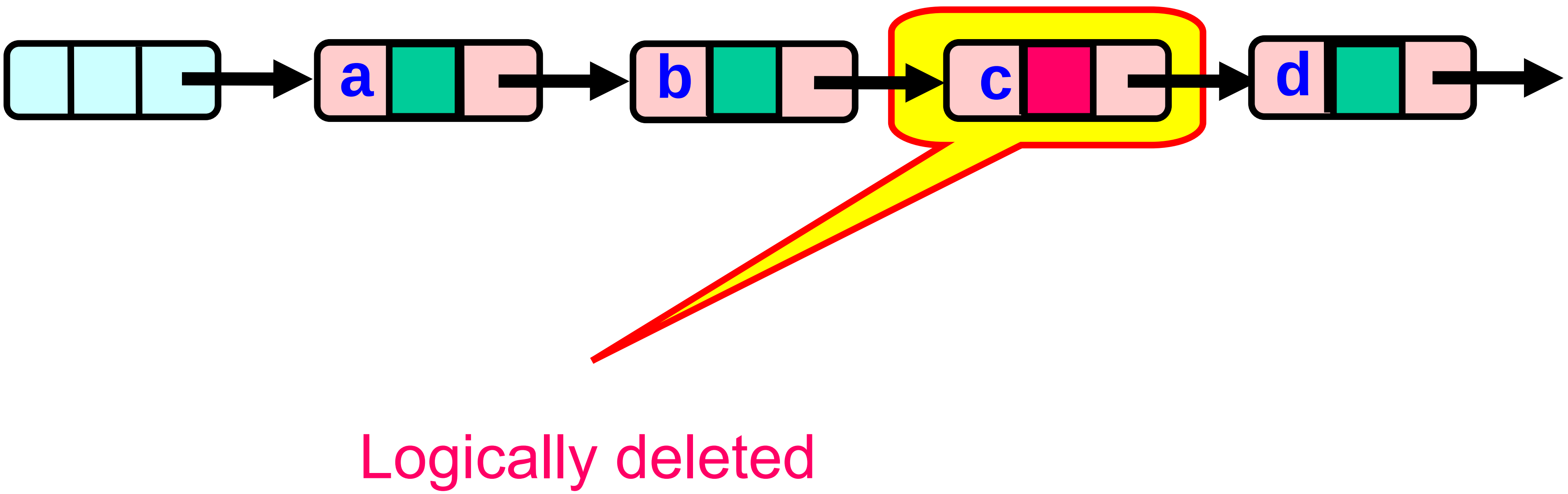


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

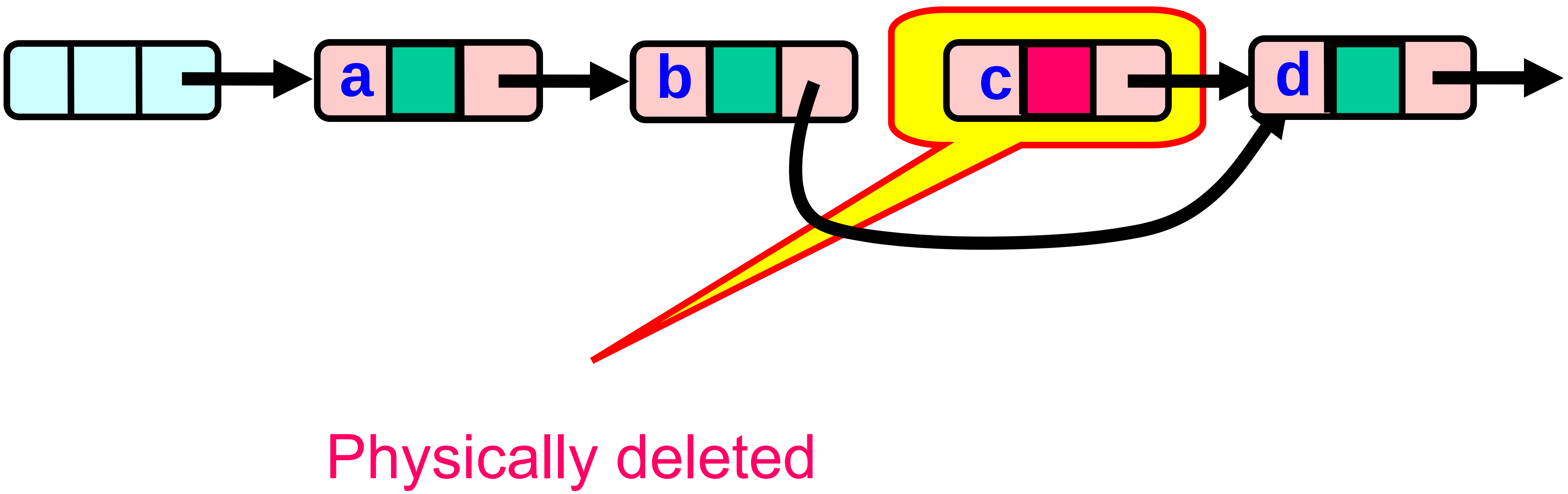
Lazy Removal



Lazy Removal



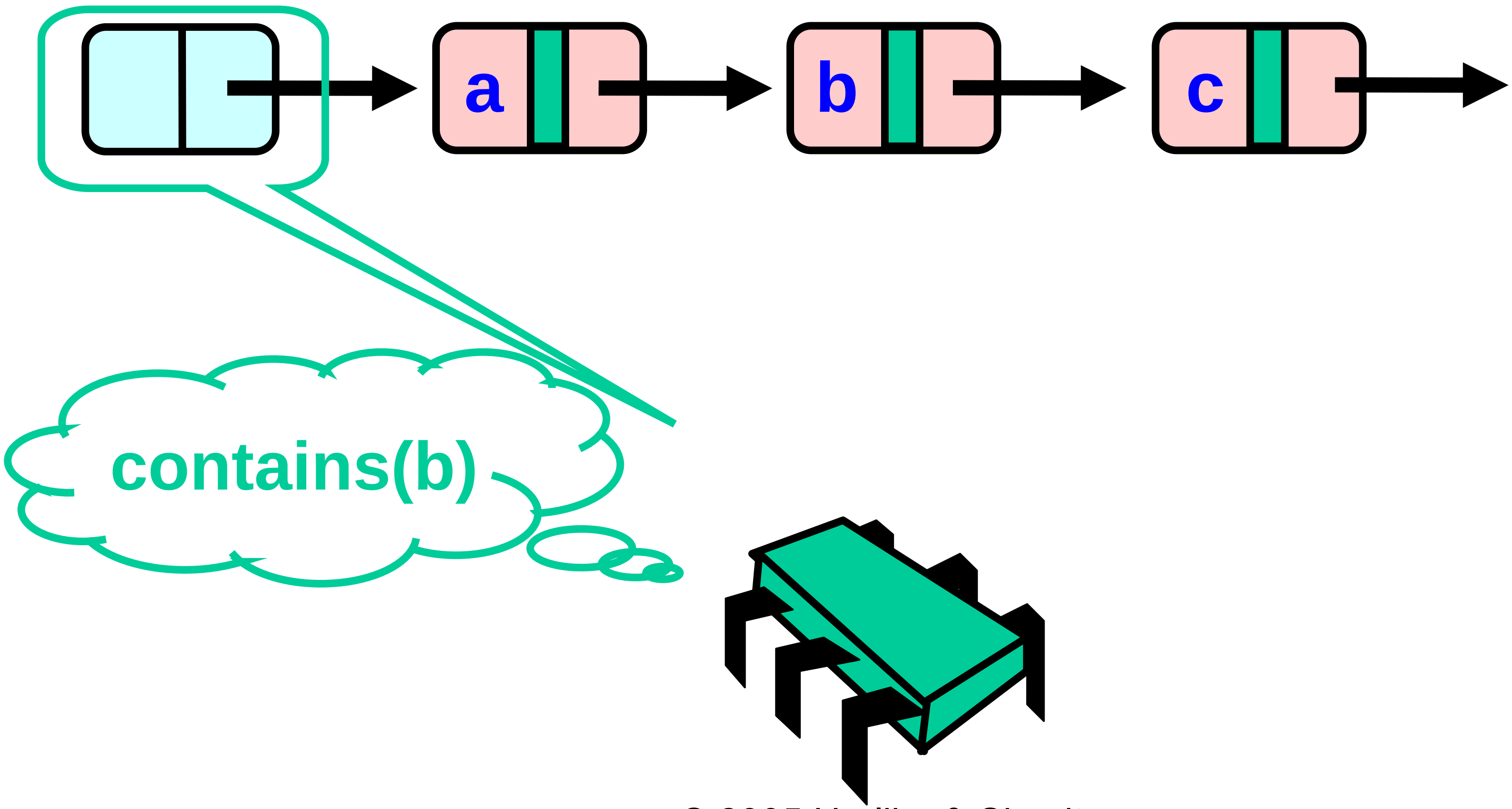
Lazy Removal



Lazy list algorithm

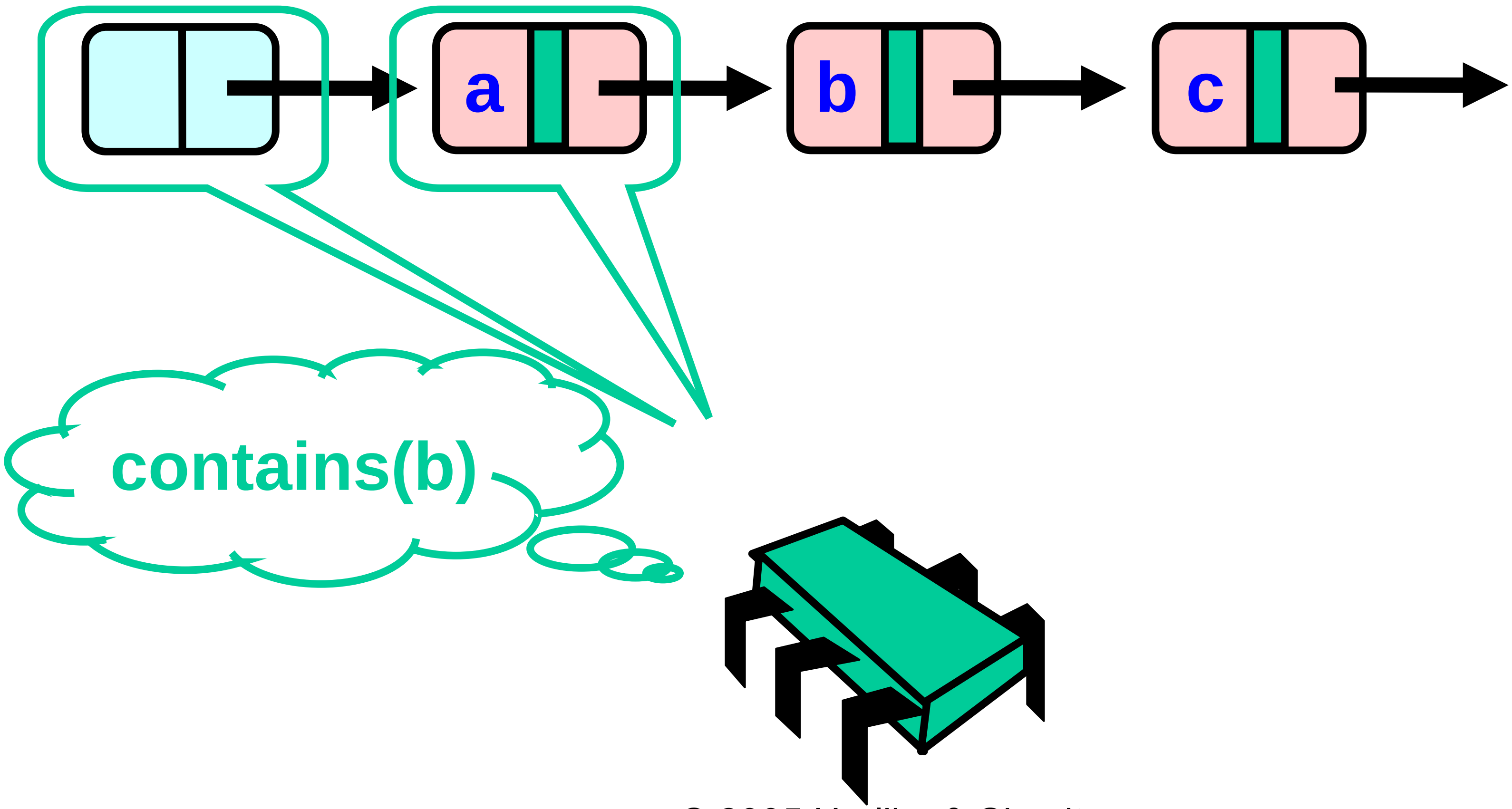
- Observation: contains(x) doesn't need to lock/validate.
- Just find first node with key ≥ x, return true iff key = x and unmarked.

Lazy list algorithm

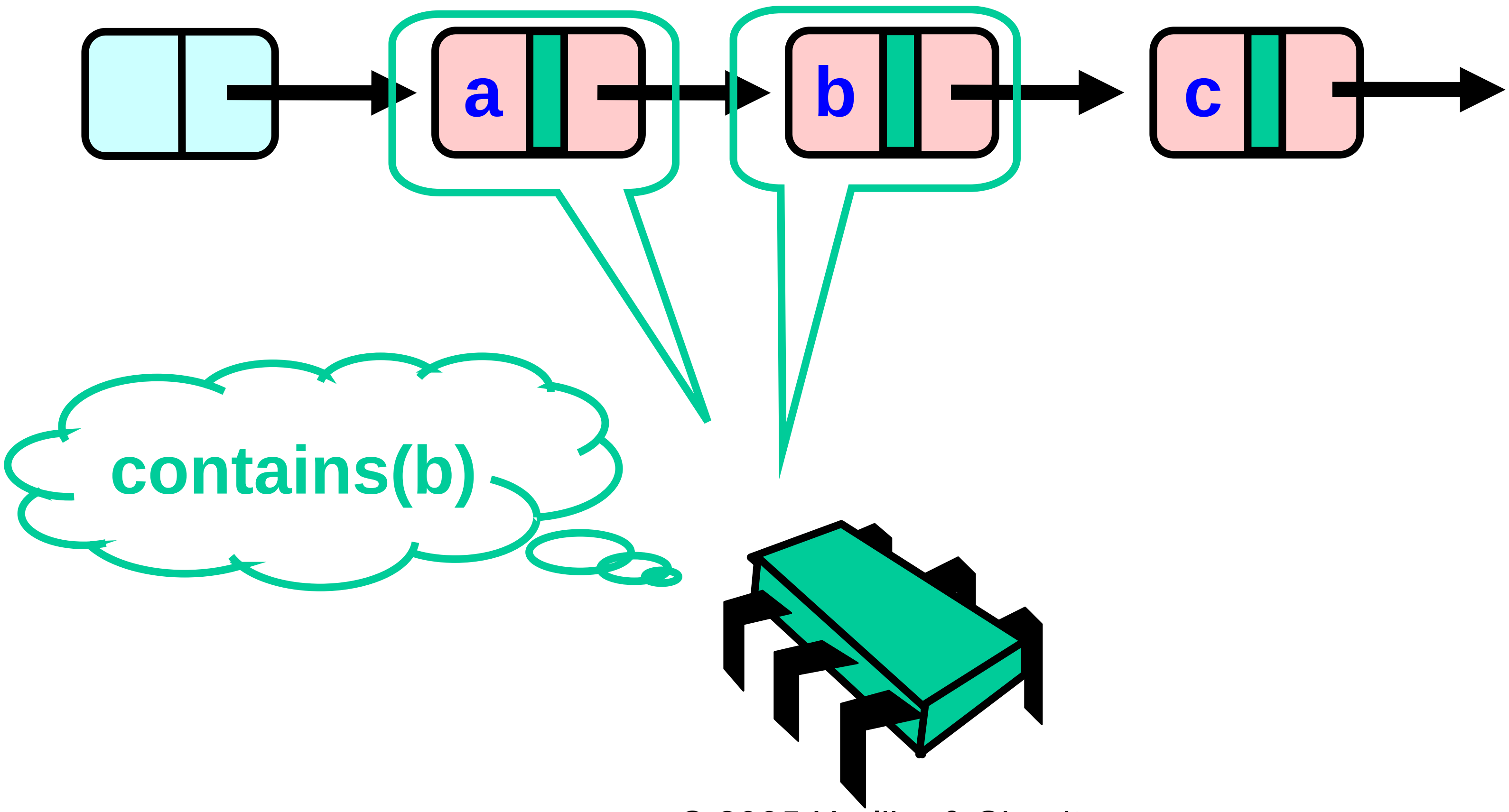


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

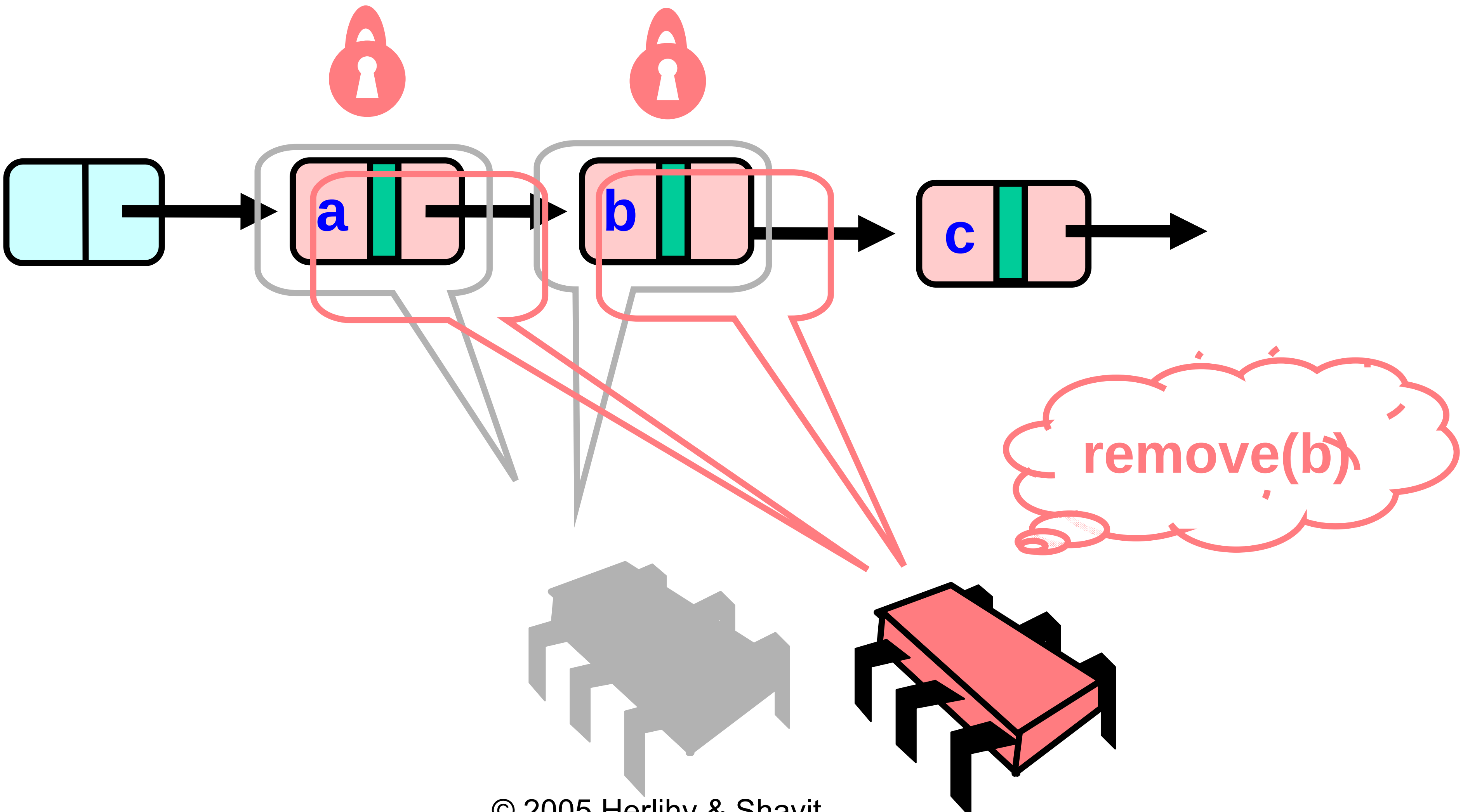
Lazy list algorithm



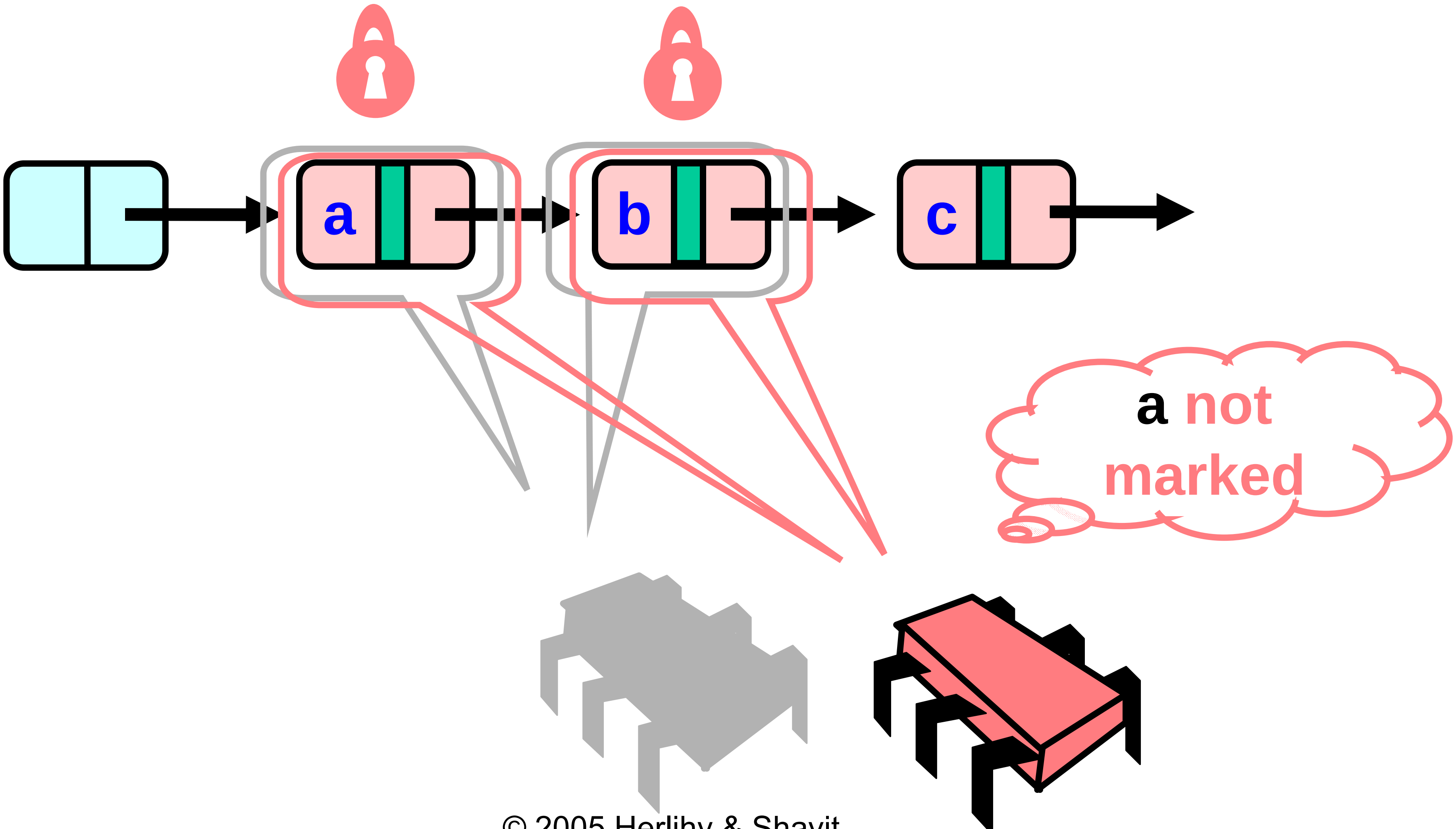
Lazy list algorithm



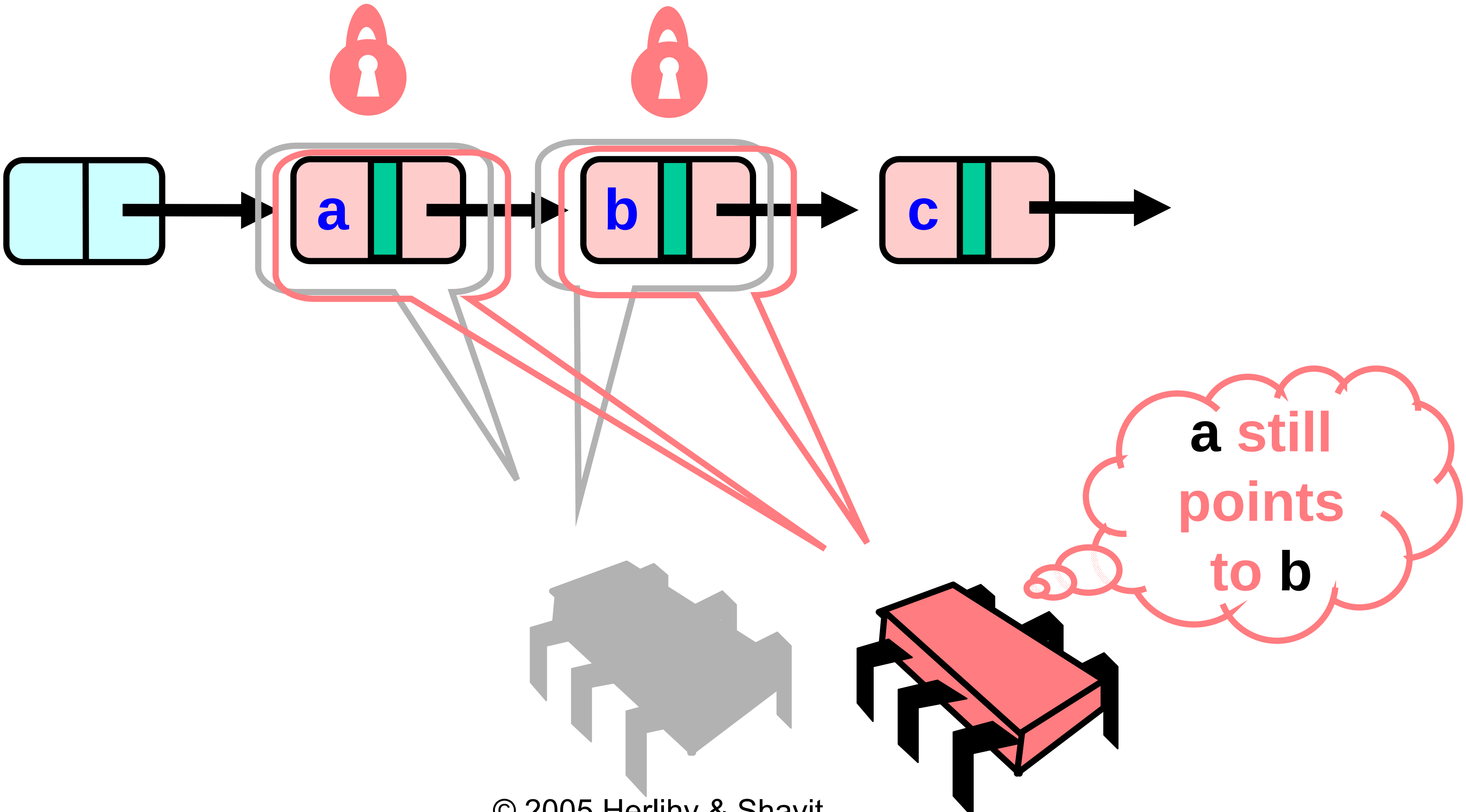
Lazy list algorithm



Lazy list algorithm

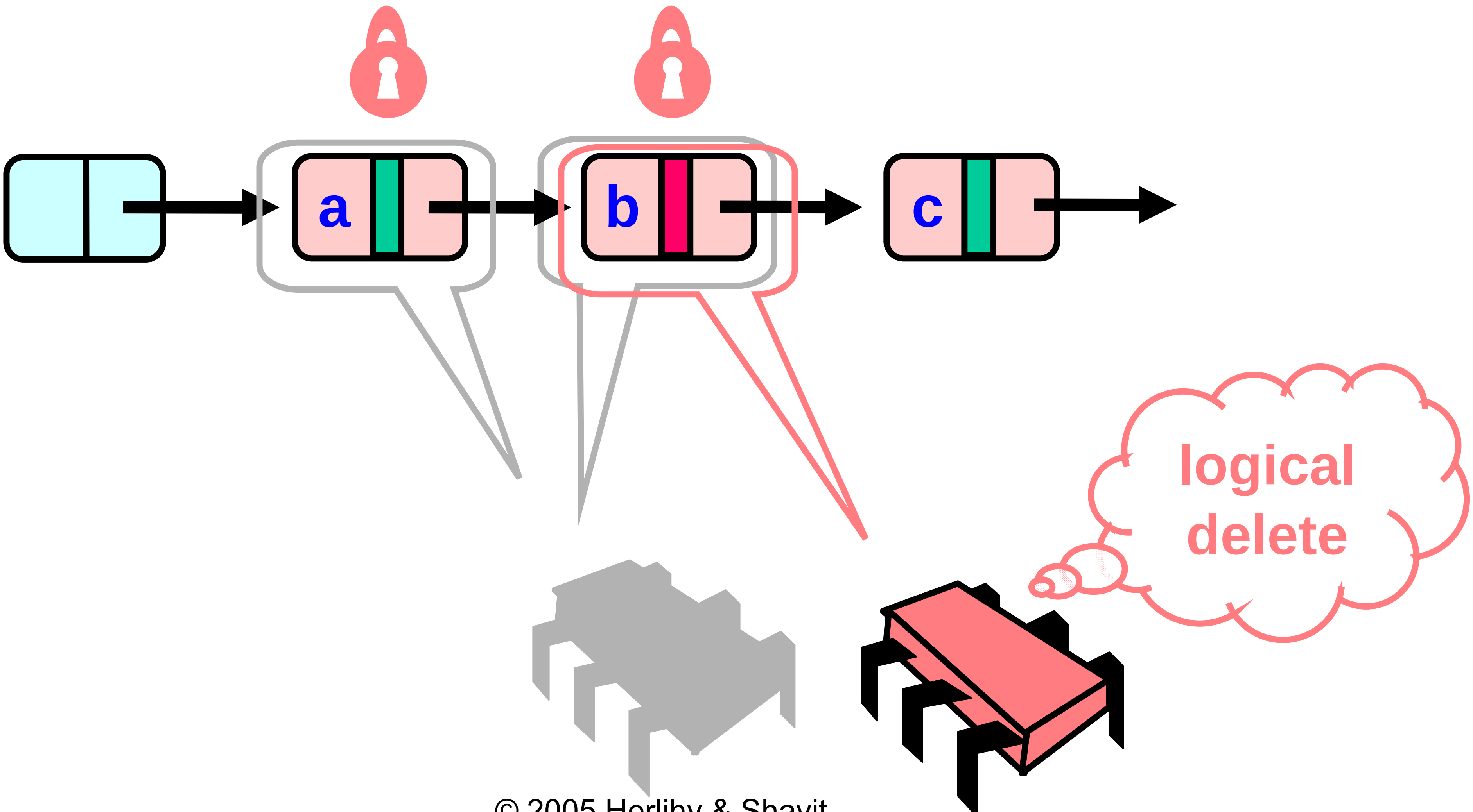


Lazy list algorithm



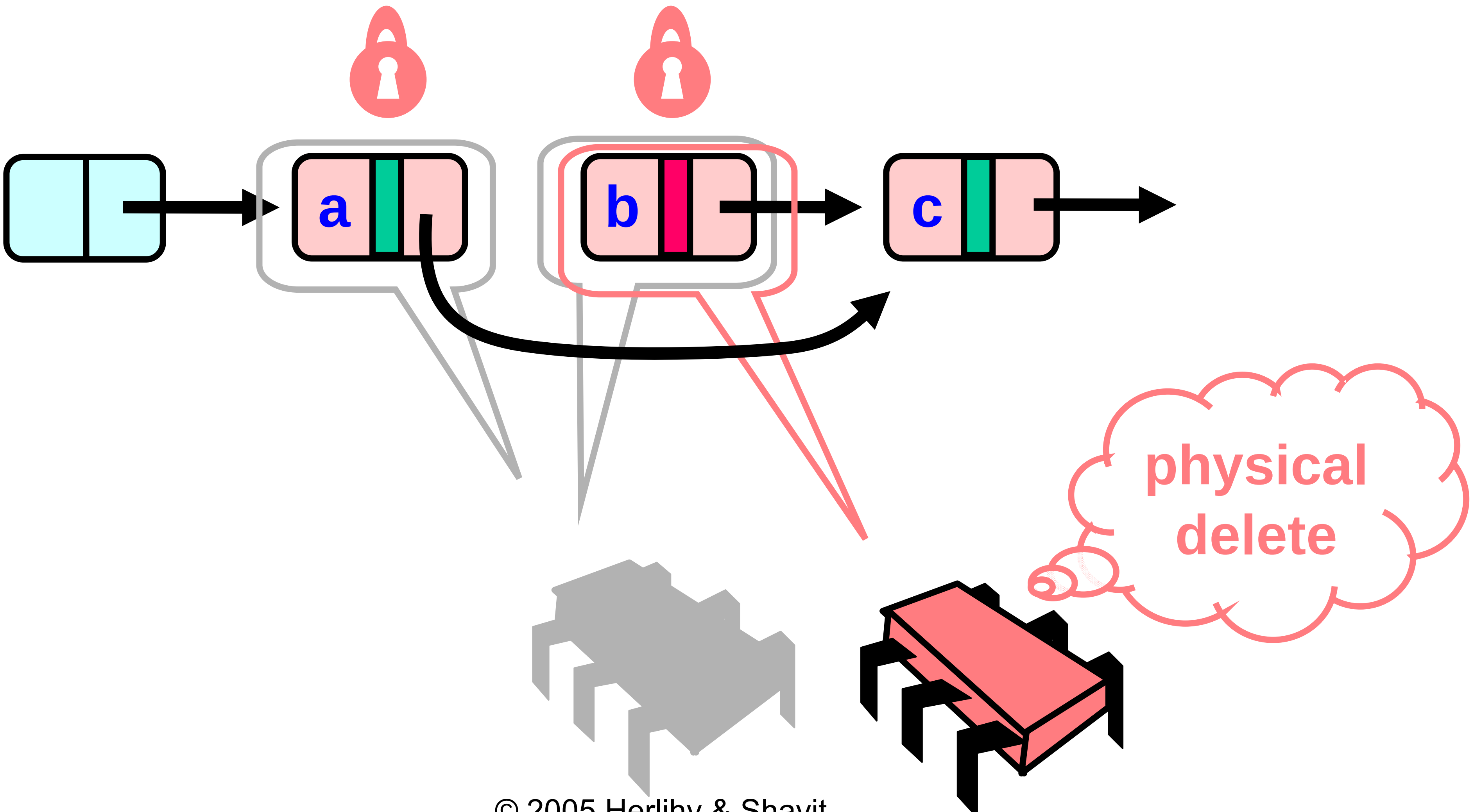
From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Lazy list algorithm

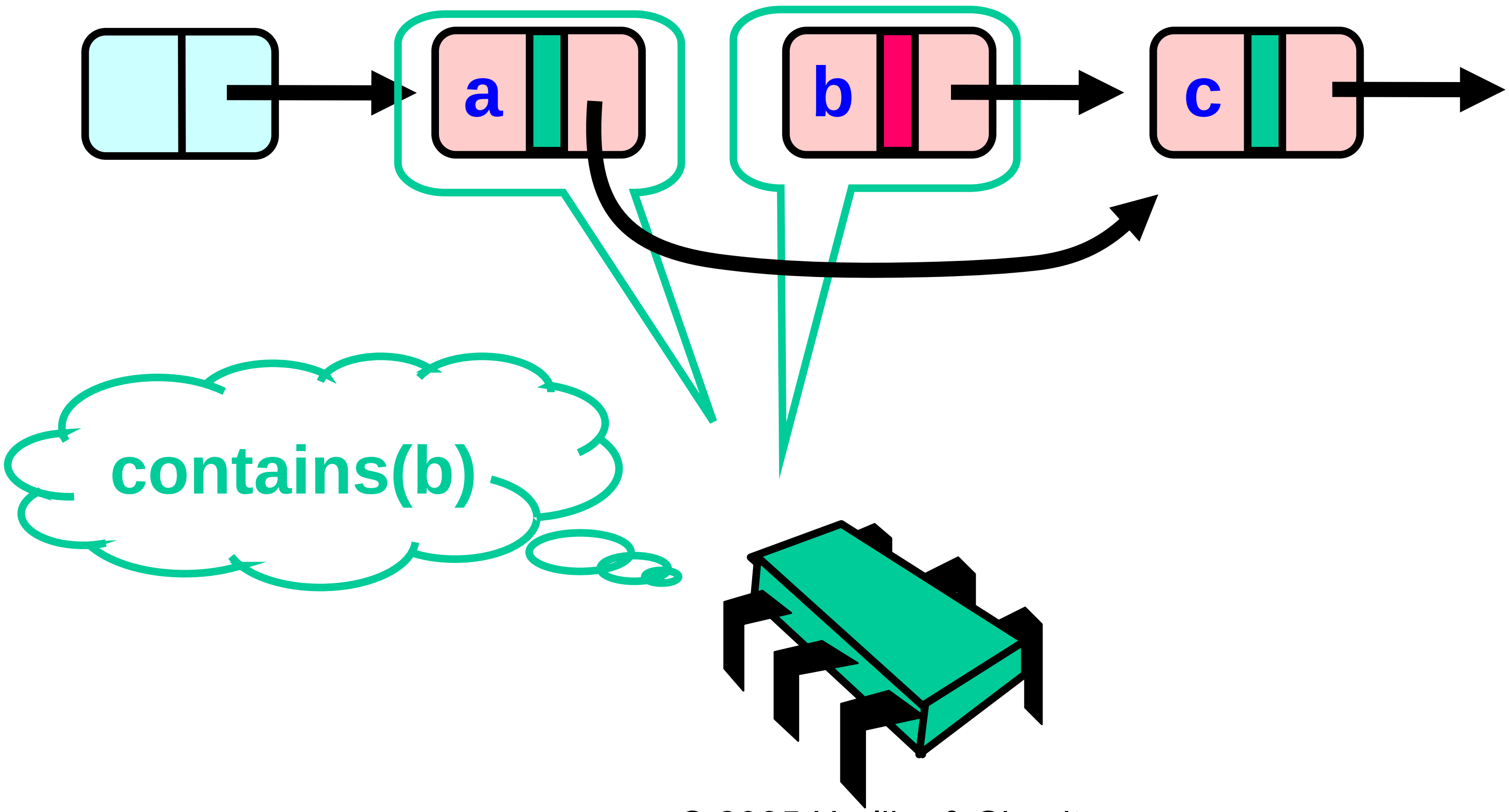


From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

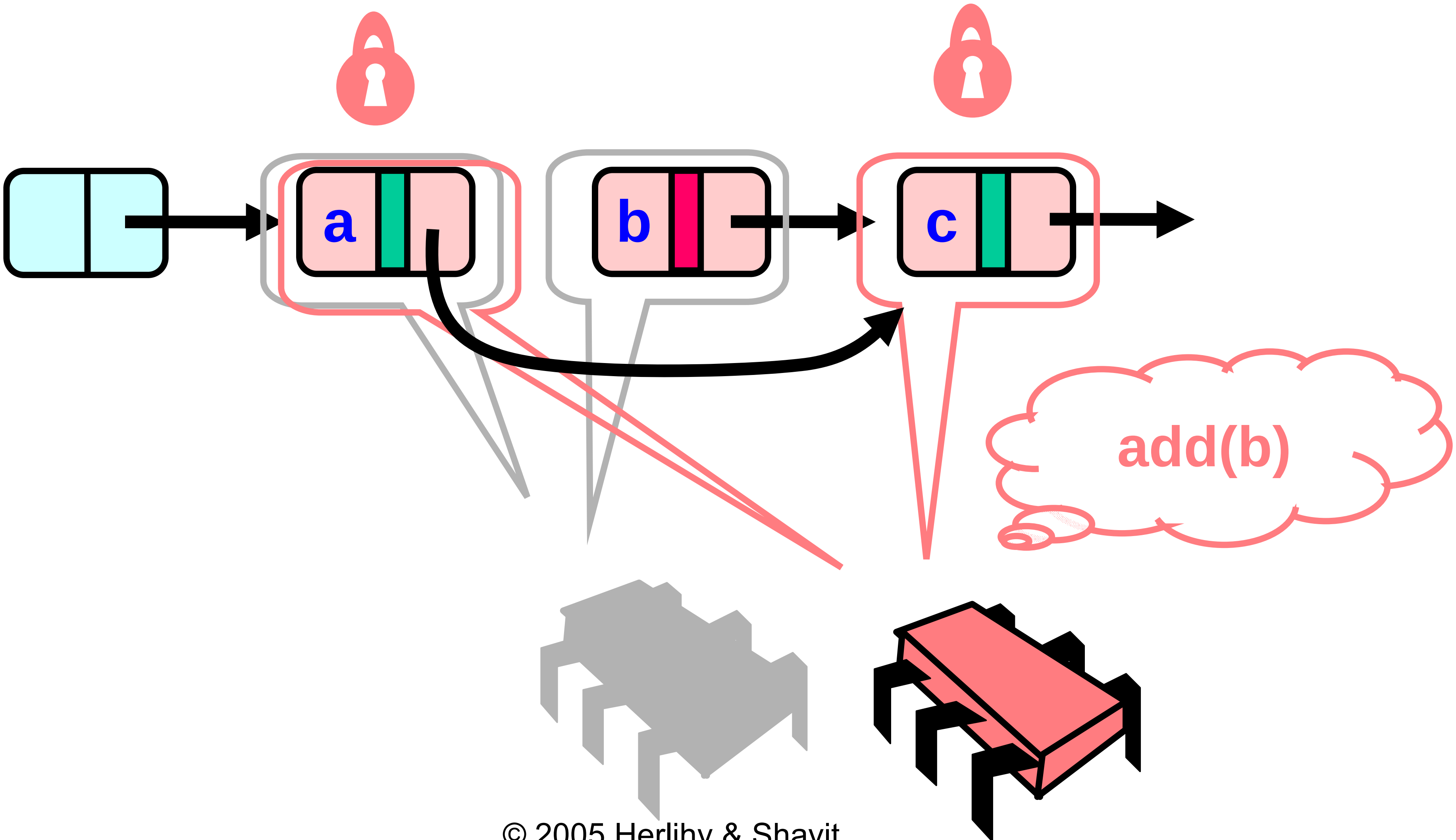
Lazy list algorithm



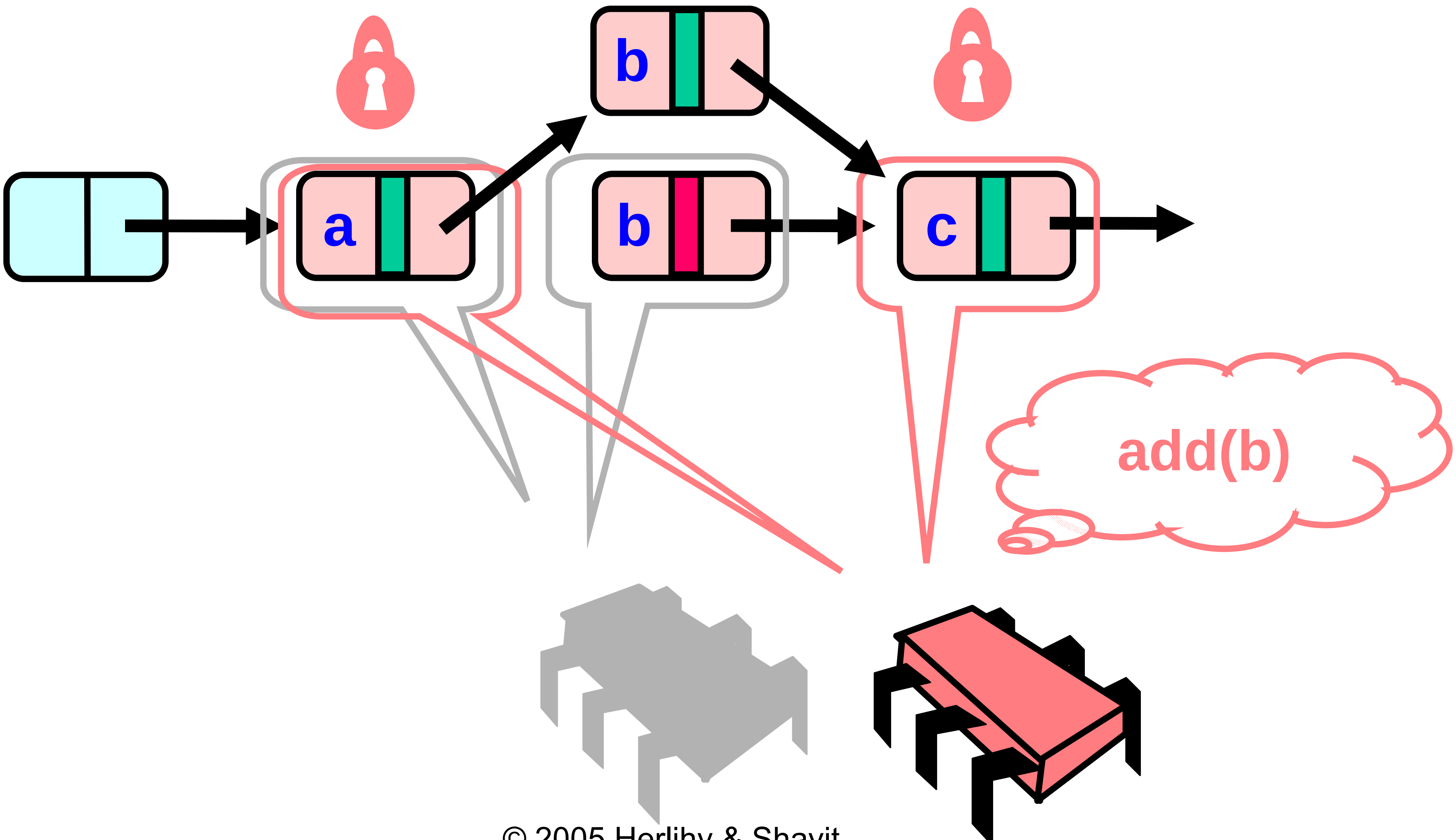
Lazy list algorithm



Lazy list algorithm

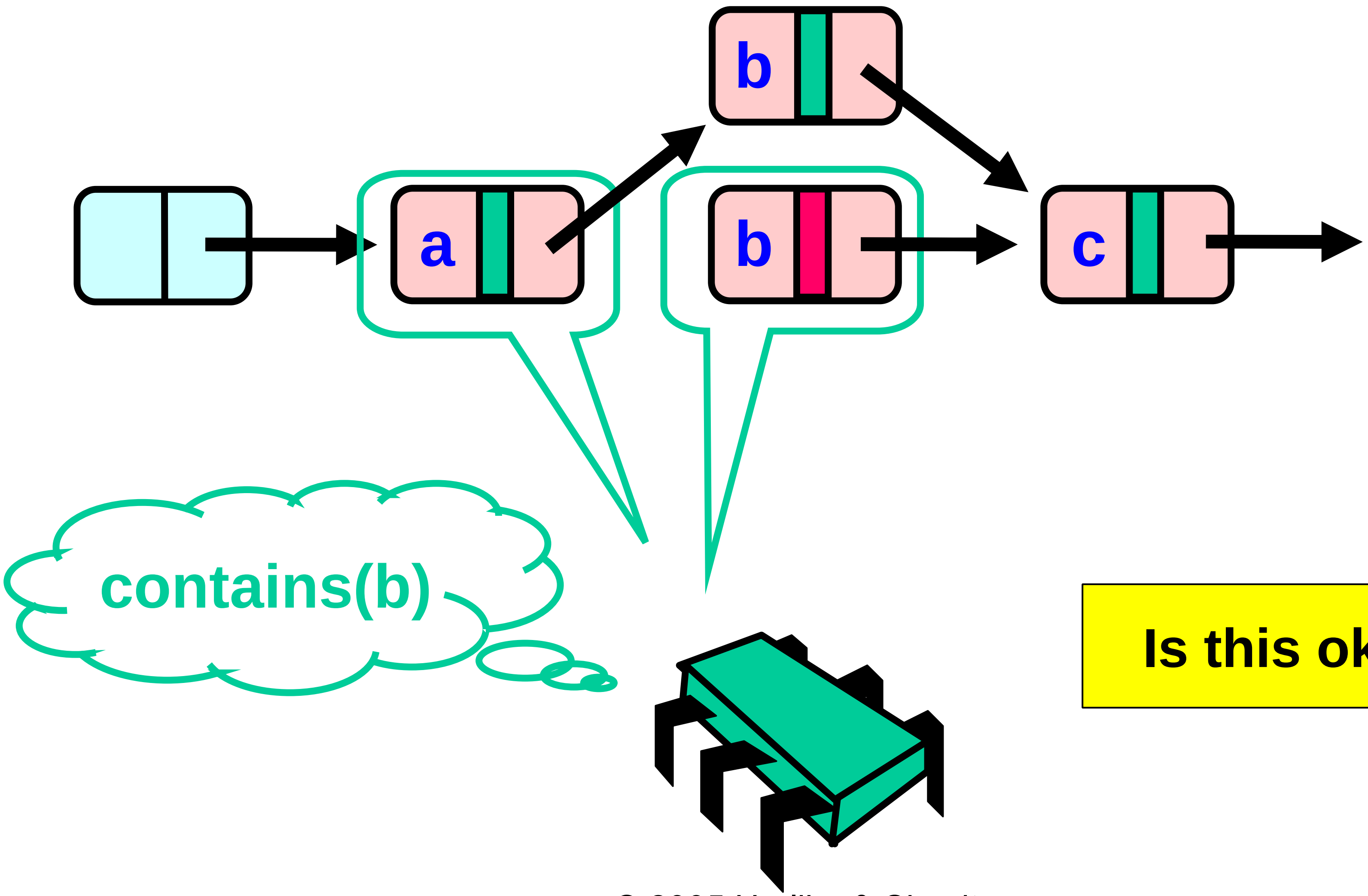


Lazy list algorithm



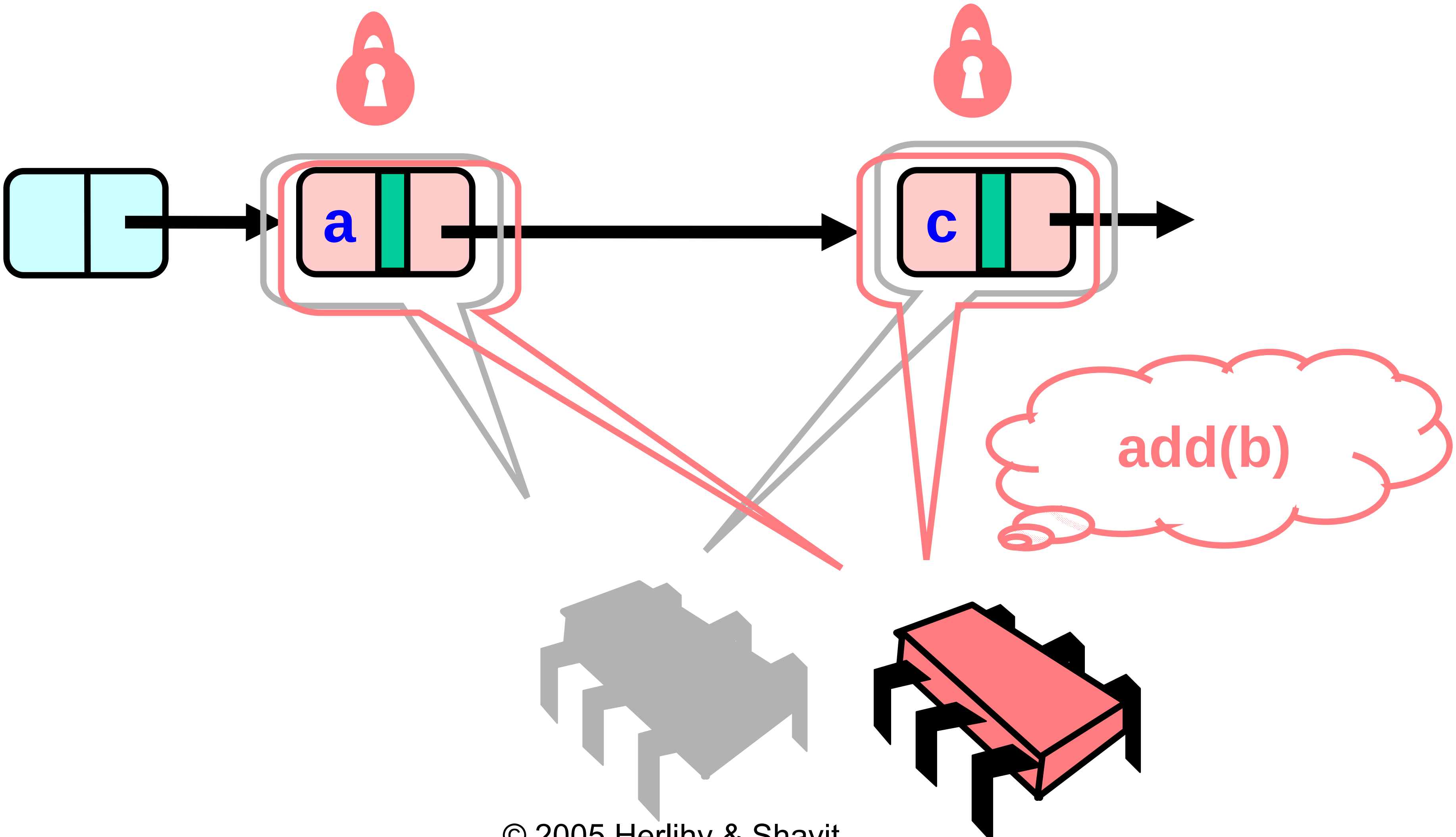
From *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit.

Lazy list algorithm

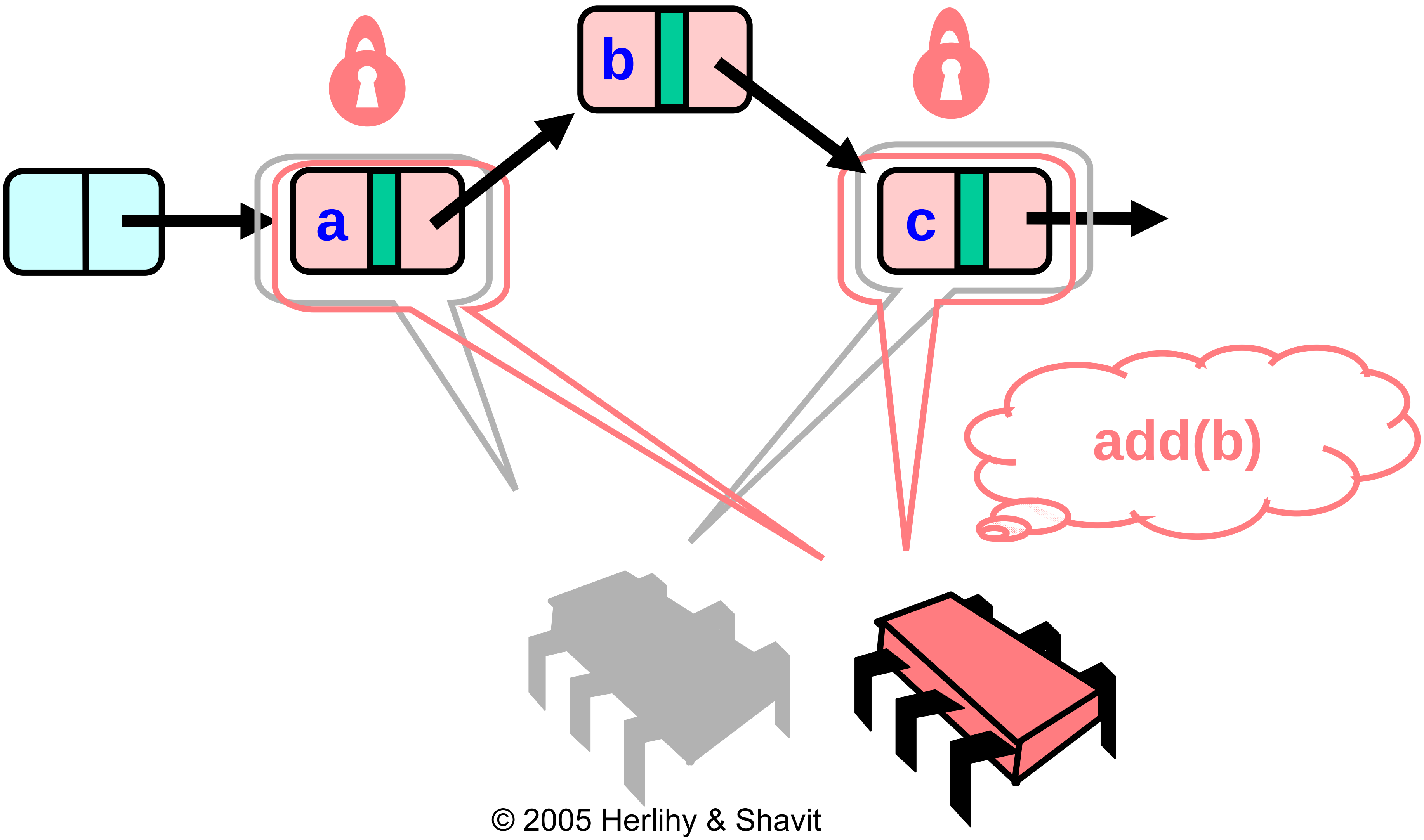


Is this okay?

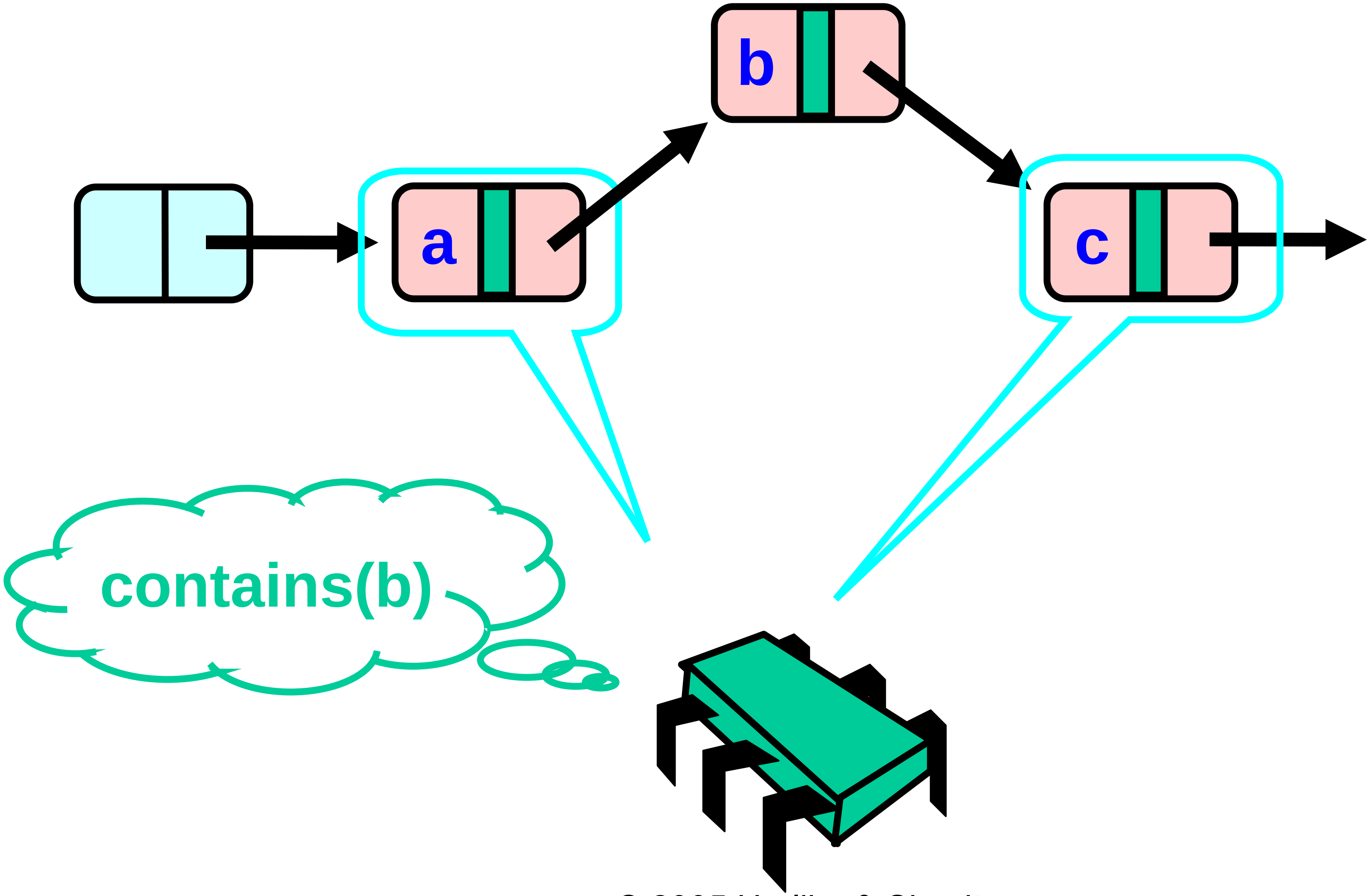
Lazy list algorithm



Lazy list algorithm



Lazy list algorithm



Lazy List: Add

Nodes have fields: key, next, mark.

```
S.add(x)
retry:
   pred := S.head; curr := pred.next
   while (curr.key < x) do pred := curr; curr := curr.next
   if (curr.key = x and curr.mark = false) then return false
   else
       pred.lock()
       if (pred.mark = false and pred.next = curr) then
         node := new Node(x)
         node.next := curr
         pred.next := node
         pred.unlock()
         return true
      else
         pred.unlock()
         goto retry
```

Lazy List: Remove

```
S.remove(x)
retry:
   pred := S.head; curr := pred.next
   while (curr.key < x) do pred := curr; curr := curr.next
   if (curr.key > x or curr.mark = true) then return false
   else
       pred.lock(); curr.lock()
       if (pred.mark = curr.mark = false and pred.next = curr) then
          curr.mark := true
          pred.next := curr.next
          pred.unlock(); curr.unlock()
          return true
       else
        pred.unlock(); curr.unlock()
        goto retry
```

Lazy List: Contains

```
S.contains(x)
curr := S.head.next
while (curr.key < x) do curr := curr.next
if (curr.key = x and curr.mark = false) then return true
else return false
```

Lazy list algorithm

- Serializing contains(x) that returns false
  - if node found has key > x
    - when node.key is read?
    - when pred.next is read?
    - when pred is marked (if it is marked)?
  - if node with key = x is marked
    - when mark is read?
    - when pred.next is read?
    - when mark is set?

Lazy list algorithm

- Serializing contains(x) that returns false
  - if node found has key > x
    - when node.key is read?
    - when pred.next is read?
    - when pred is marked (if it is marked)?
  - if node with key = x is marked
    - when mark is read?
    - when pred.next is read?
    - when mark is set?

Can we do this for the optimistic list?

Correctness

- Atomicity:
  - Forward simulation to canonical atomic set:
    - S = values in reachable unmarked nodes.
  - "perform" steps:
    - contains(x) or unsuccessful add(x) or remove(x): LTTR, based on some technical cases.
    - Successful add(x): When pred.next := node.
    - Successful remove(x): When curr.mark := true.
- Liveness:
  - contains is wait-free.
  - add, remove are blocking.
  - add, remove satisfy progress, but not lockout-freedom.

Lock-free list with wait-free contains()

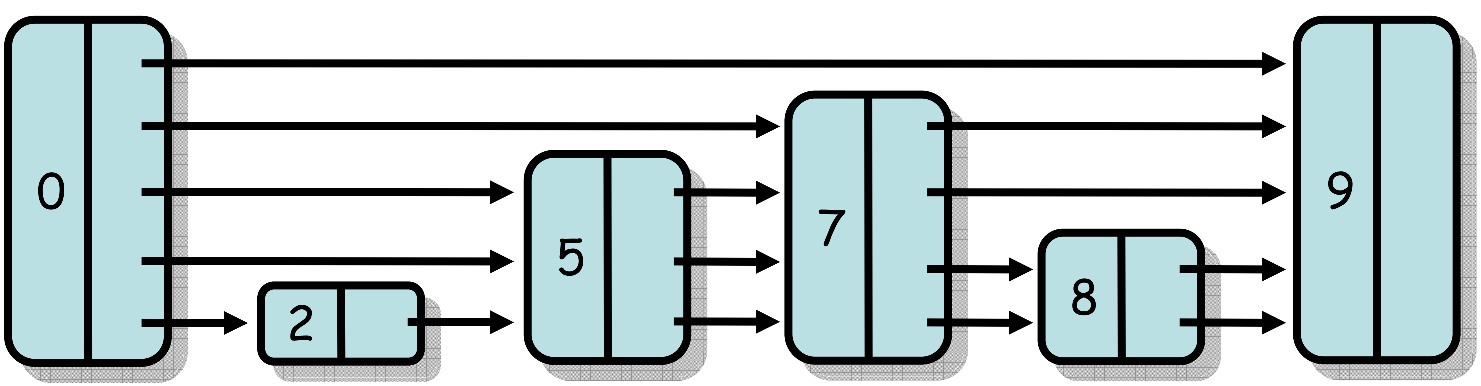
- Add and remove just like lock-free list.
- Contains() does not help, does not retry, just like in lazy list.

Evaluation/Comparison

- Lock-free list with wait-free contains():
  - contains() is wait-free
  - add() and remove() are nonblocking (lock-free)
  - Incurs overhead of CAS and of cleanup.
- Lazy list:
  - contains() is wait-free
  - add() and remove() are blocking, but use short lock durations.
  - Low overhead.

Application of list techniques

- Trees
- Skip lists
  - multiple layers of links
  - list at each layer is sublist of layer below
  - logarithmic expected search time if each list has half elements of next lower level
    - probabilistic guarantees



Next time

- Transactional memory
- Reading:
  - HS, Chapter 9
  - Guerraoui, Kapalka