6.852 Lecture 22

- Techniques for highly concurrent objects (continued)
  - “lazy” synchronization
  - illustrate on list-based sets, apply to other data structures
- Transactional memory
- Reading:
  - Herlihy-Shavit Chapter 8 (Chapter 9 in draft version)
  - Herlihy, Luchangco, Moir, Scherer paper
  - Dice, Shalev, Shavit paper

Review

- Techniques
  - coarse-grained locking
    - simple: works well for low contention
  - fine-grained locking
    - allows more concurrency, but also deadlock
    - greater time and space overhead (due to more locks)
    - simple two-phase policy guarantees atomicity (doesn't help list)
    - hand-over-hand locking
    - optimistic locking
  - lock-free techniques
    - separate “logical” and “physical” deletion
    - “announce” intention to facilitate helping (to guarantee progress)

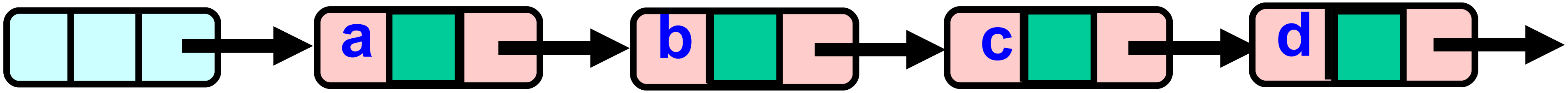
Review

- Optimistic locking
  - search down list without locking; lock appropriate nodes
  - verify that nodes are adjacent and in list (validation)
    - requires traversing the list again
    - retry if validation fails
  - good if validation typically succeeds
    - note that the list can have changed between locking and validation
  - traversal is wait-free, but
    - must traverse list twice (why?)
    - even contains must lock node (is this true?)
      - contains is typically by far the most common operation

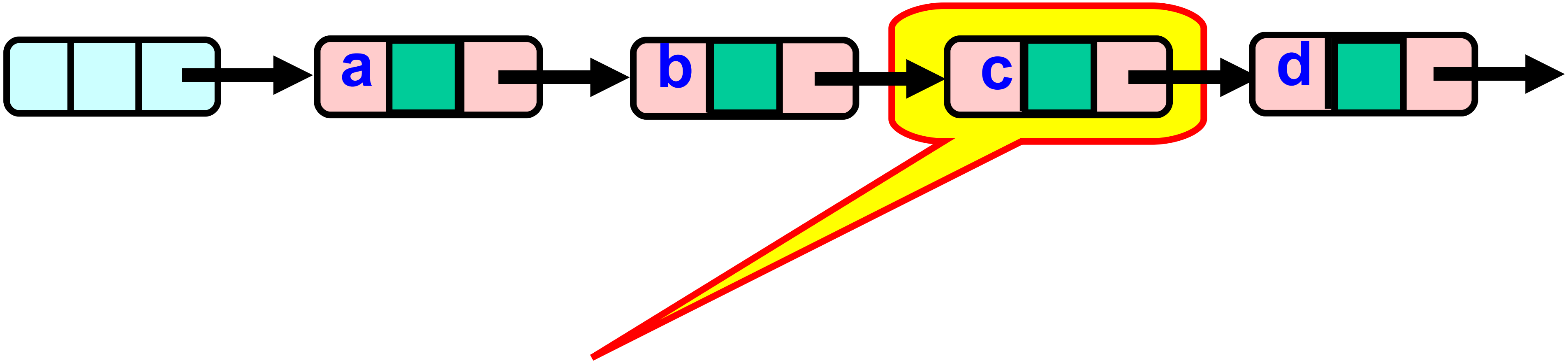
Lazy list algorithm

- Idea: use “mark” from lock-free list to avoid retraversal
  - “lazy” removal: first mark node, then splice around it
    - like lock-free list, except mark can be separate from next pointer
  - still locks node to be removed and predecessor
  - validation: check nodes are adjacent and unmarked
    - unmarked implies in list: no need to retraverse
    - much shorter critical section

Lazy Removal

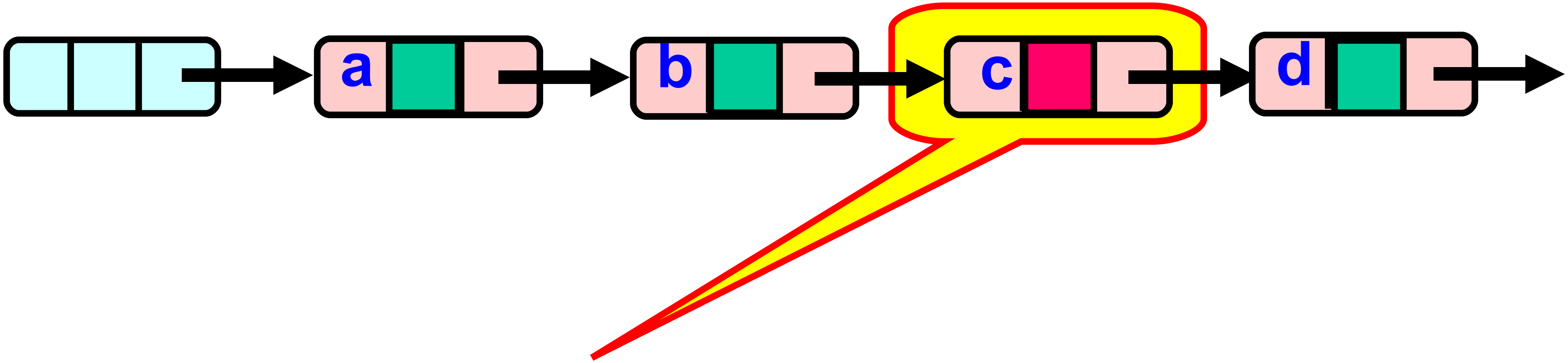


Lazy Removal



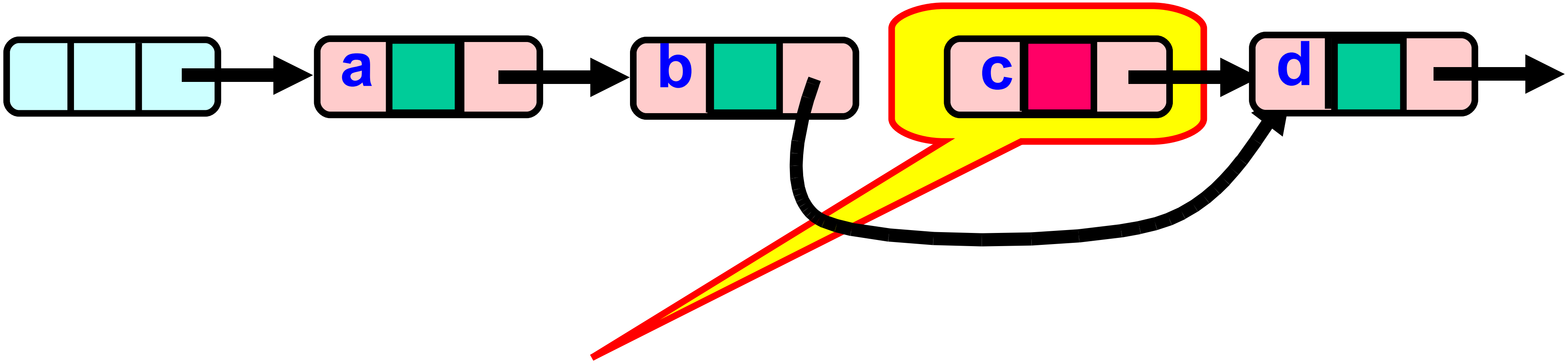
Present in list

Lazy Removal



Logically deleted

Lazy Removal

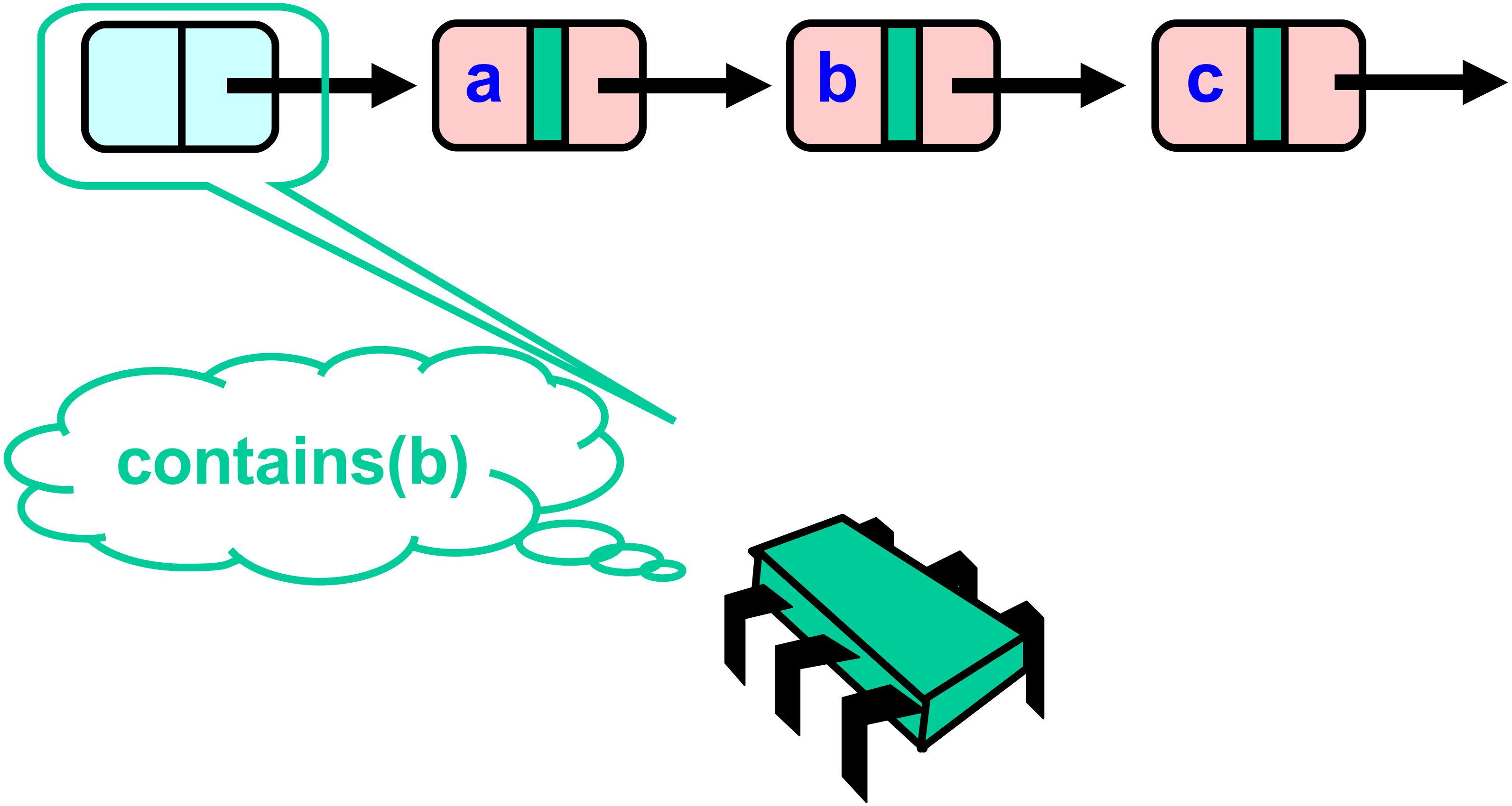


Physically deleted

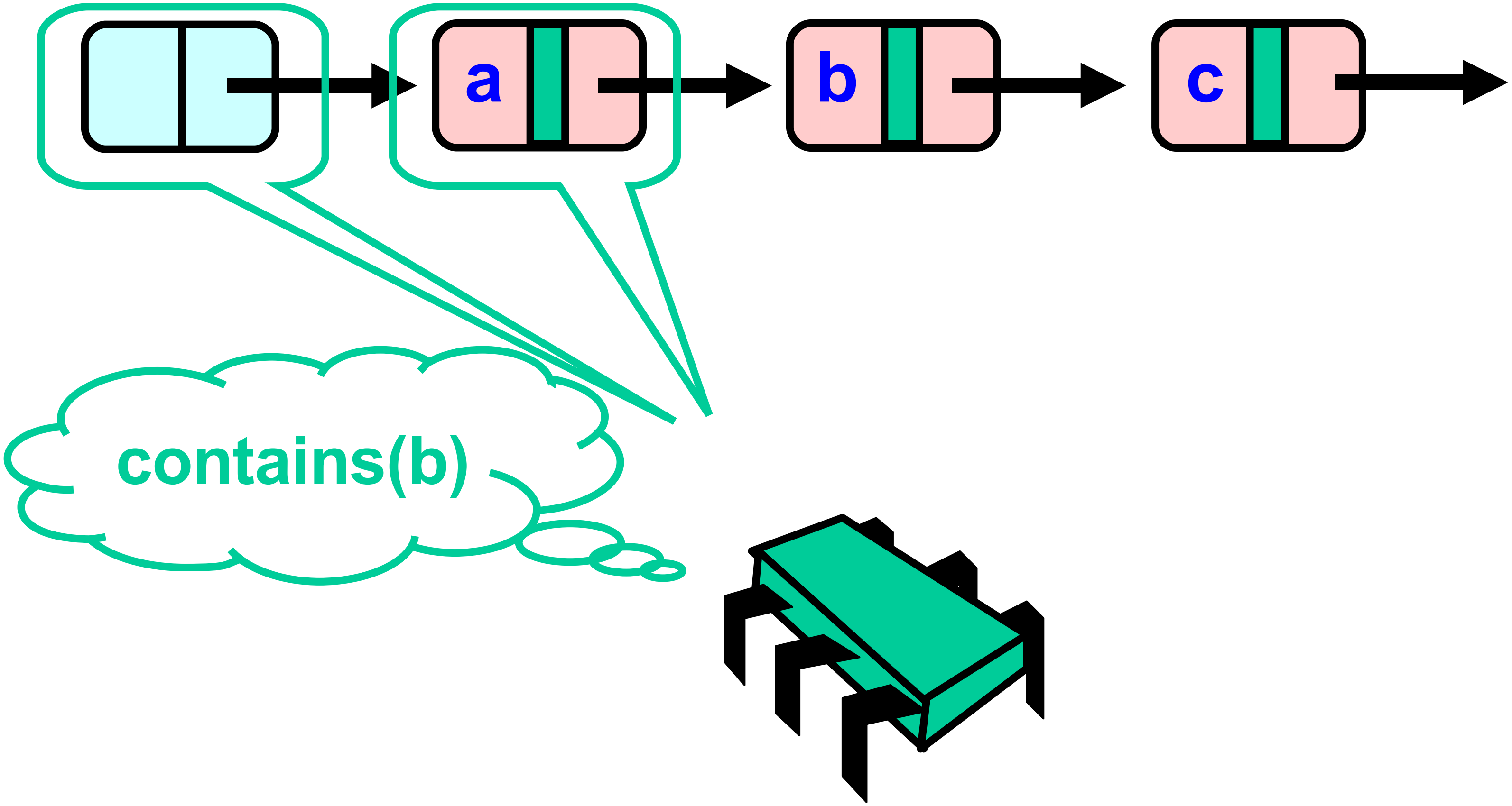
Lazy list algorithm

- Observation: contains(x) doesn't need to lock/validate
  - find first node with key ≥ x
  - return true iff unmarked and key = x
    - what if some other node with key = x is in the list?

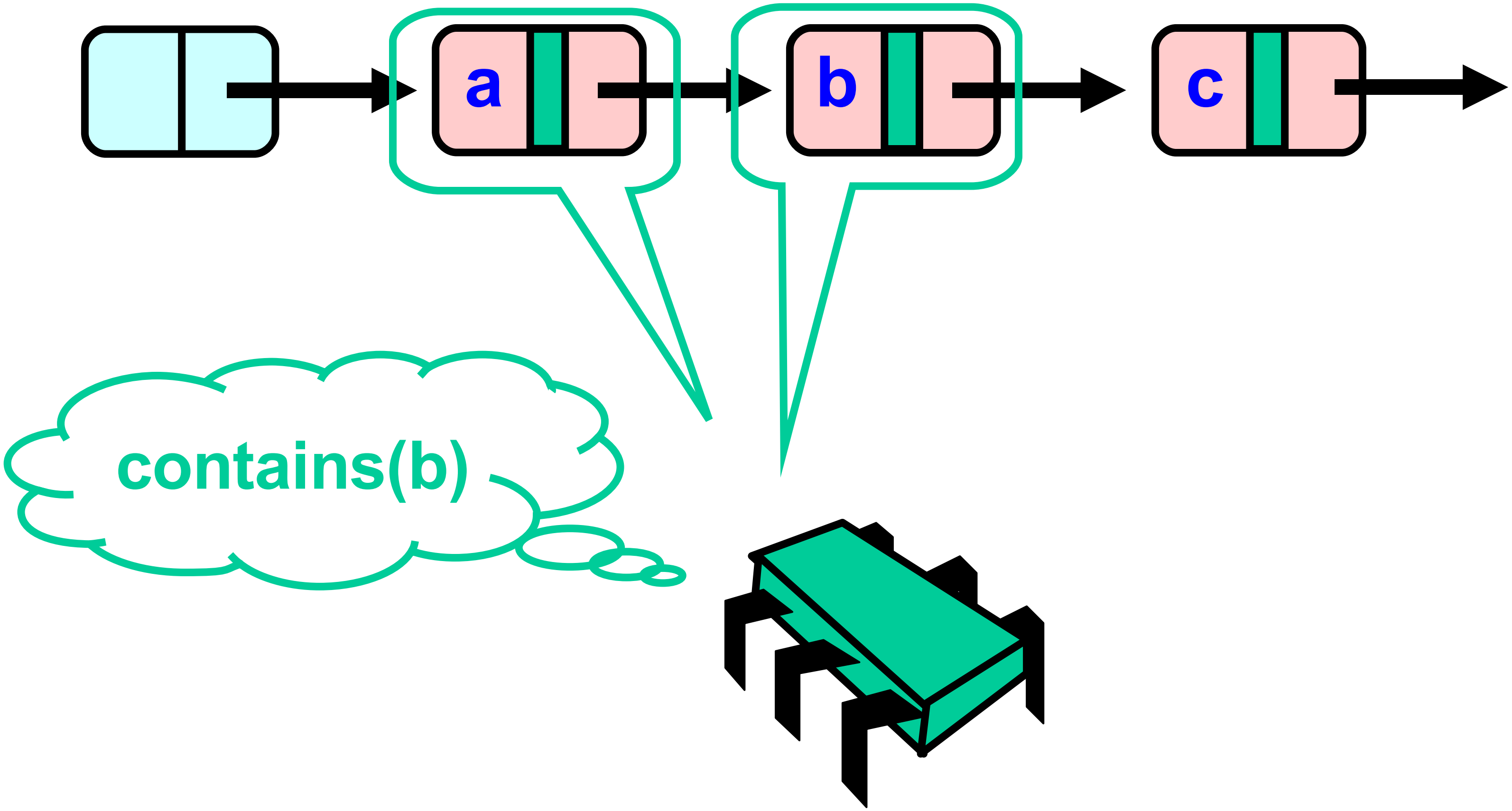
Lazy list algorithm



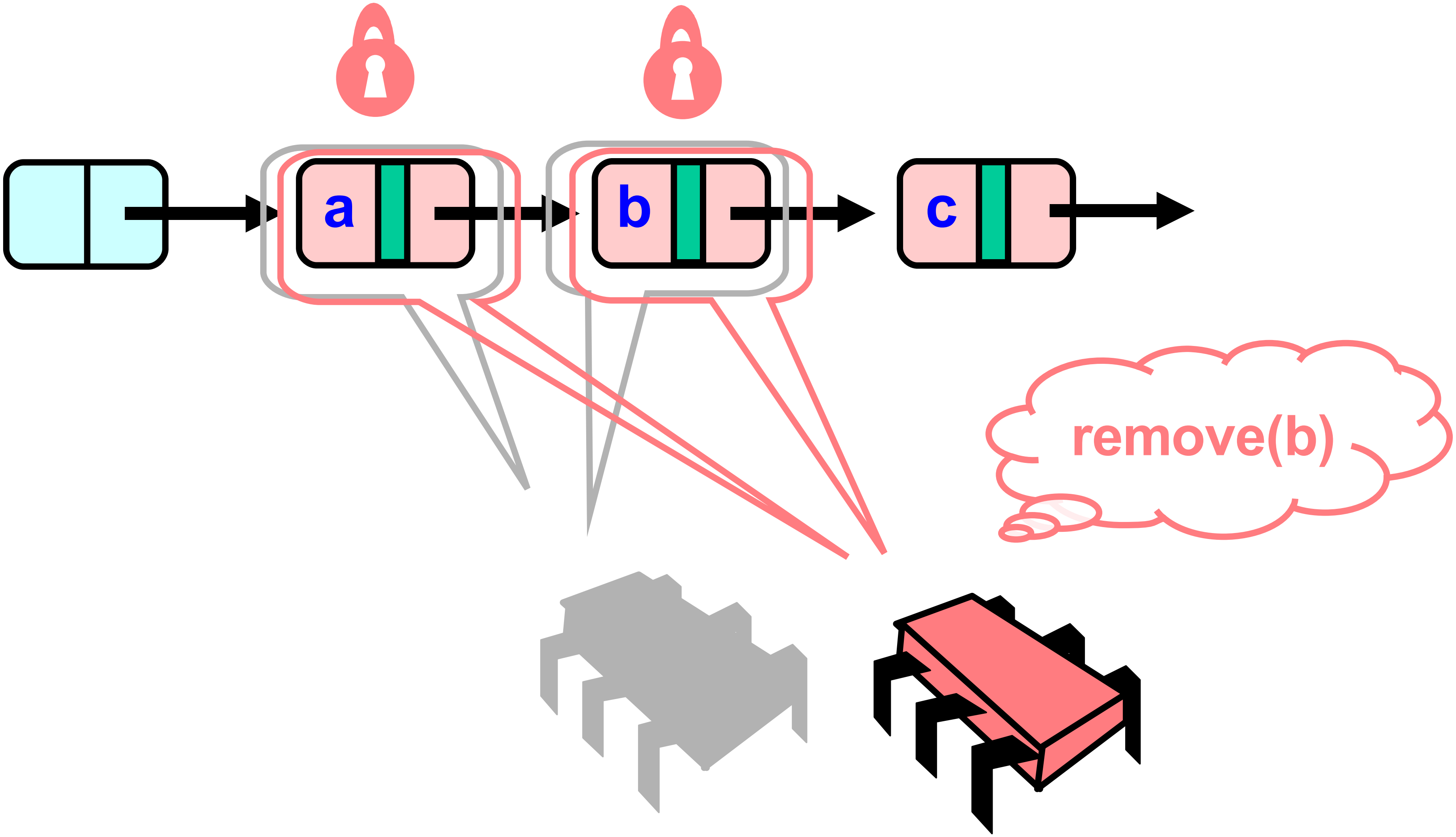
Lazy list algorithm



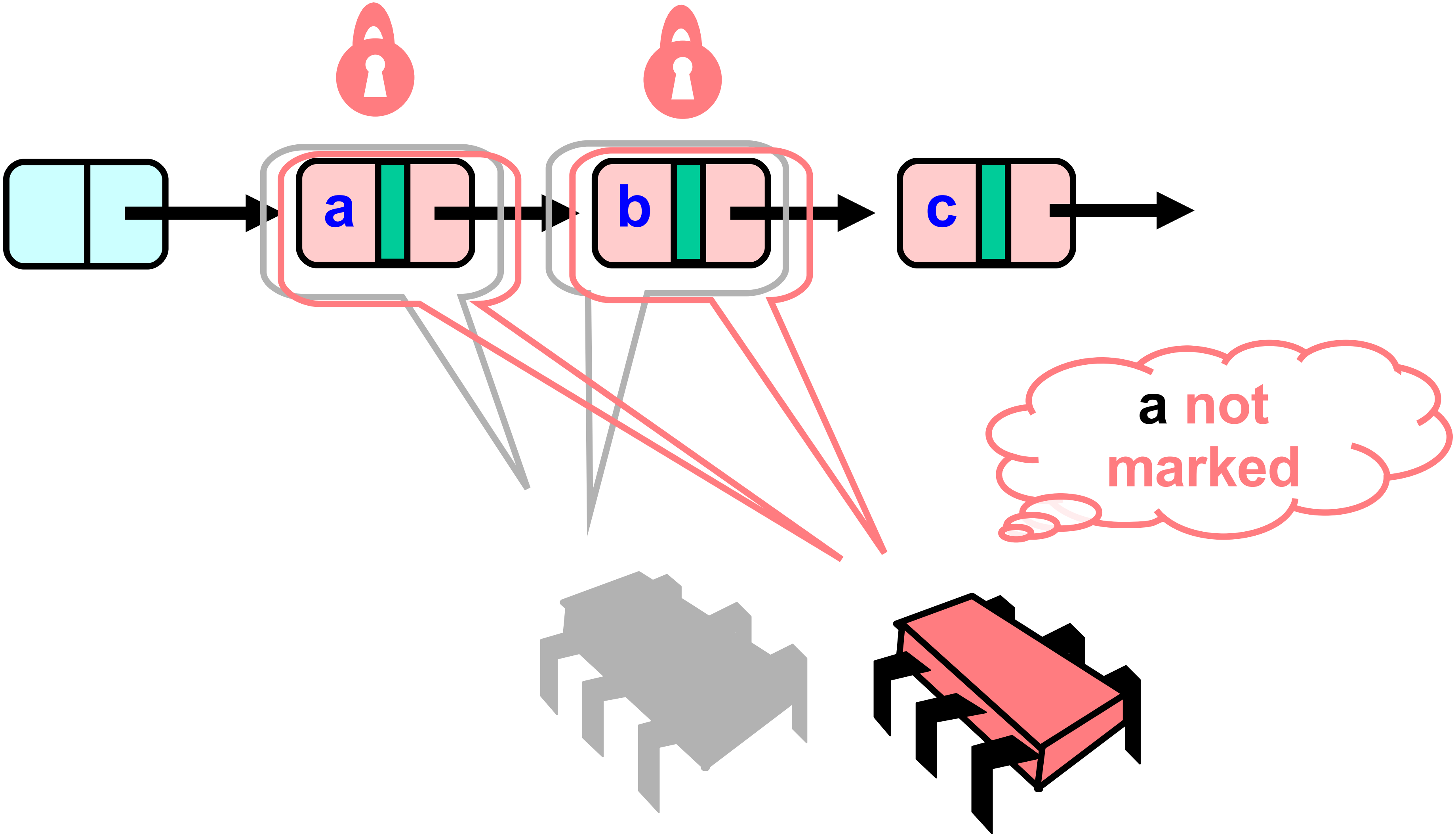
Lazy list algorithm



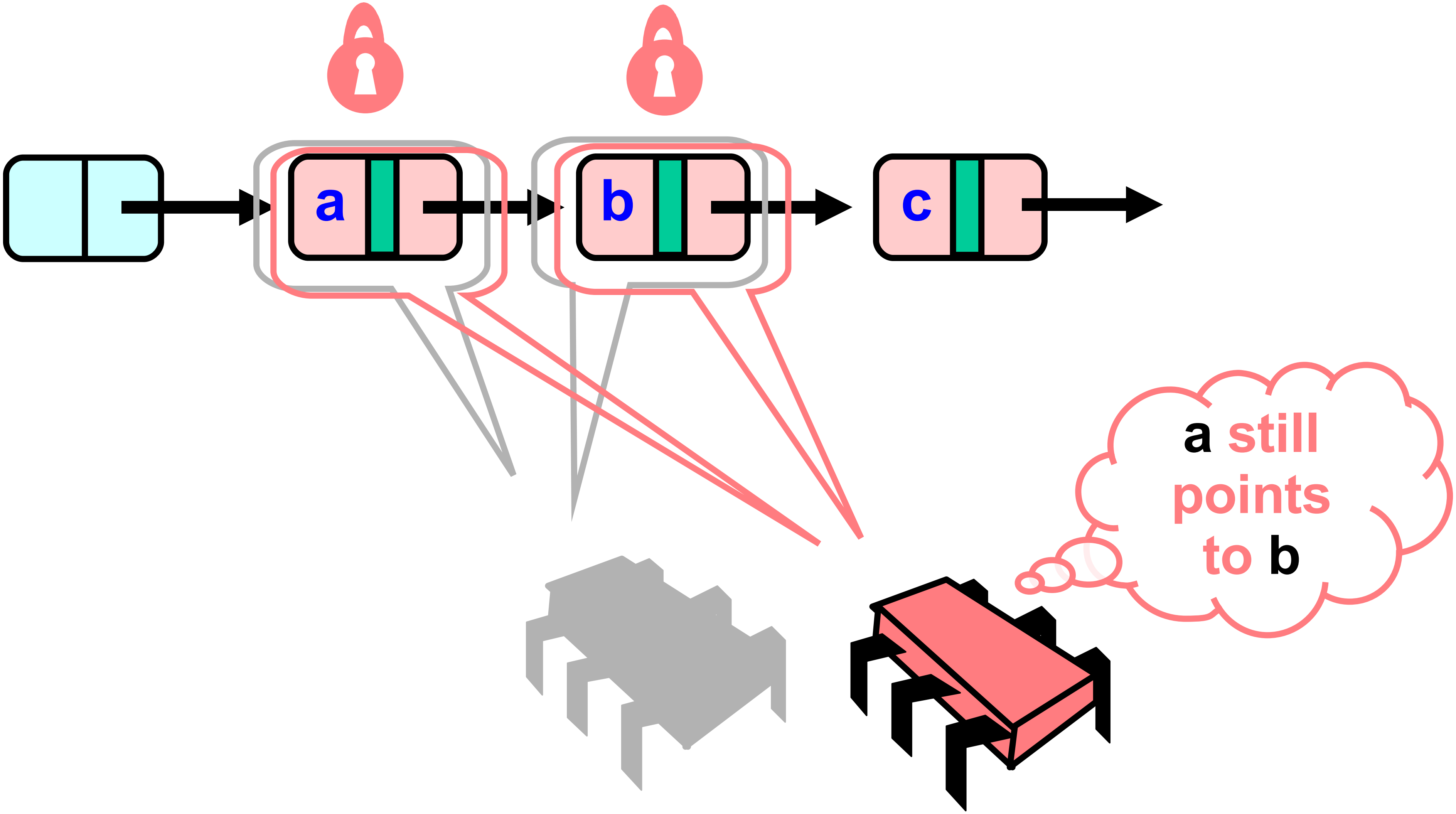
Lazy list algorithm



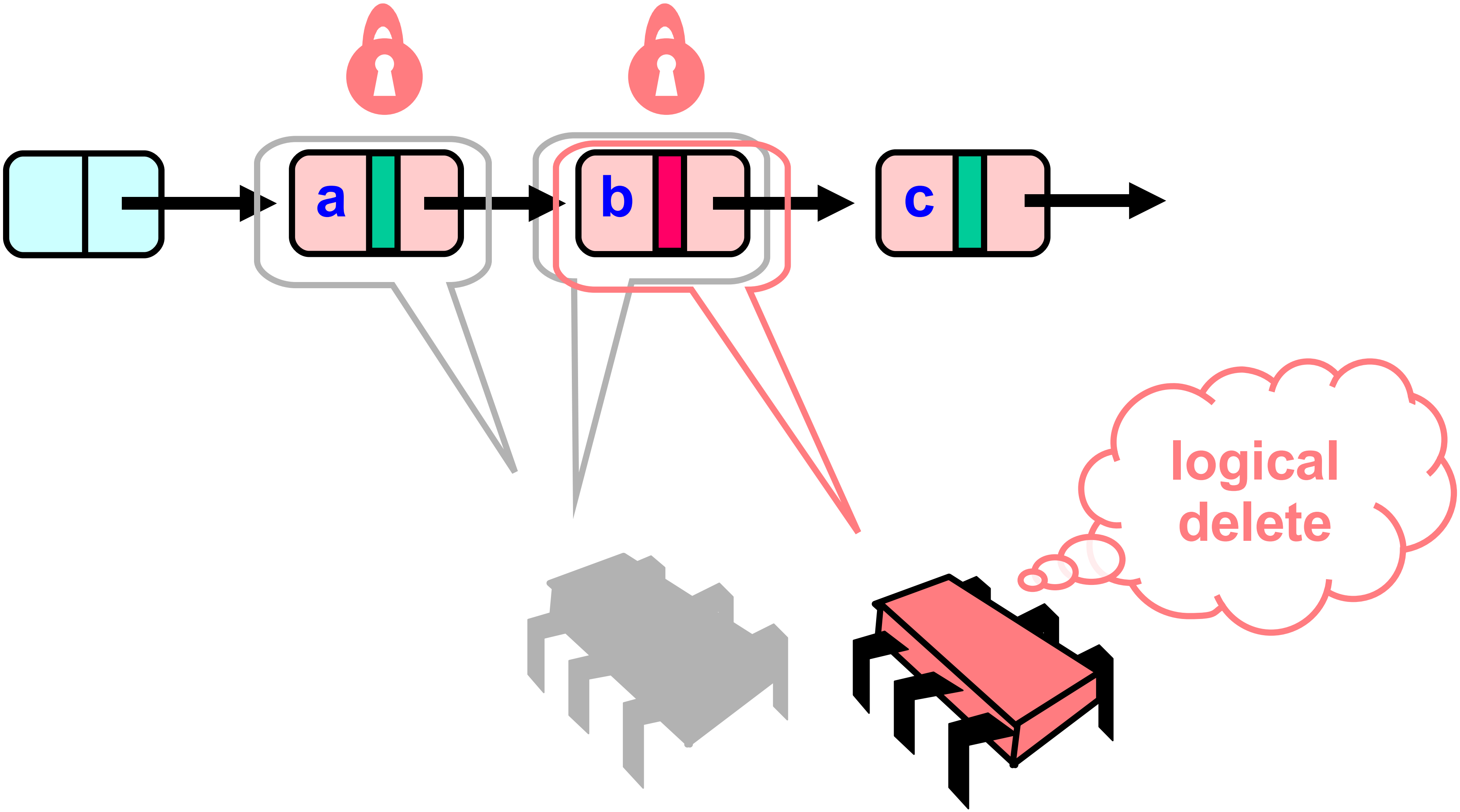
Lazy list algorithm



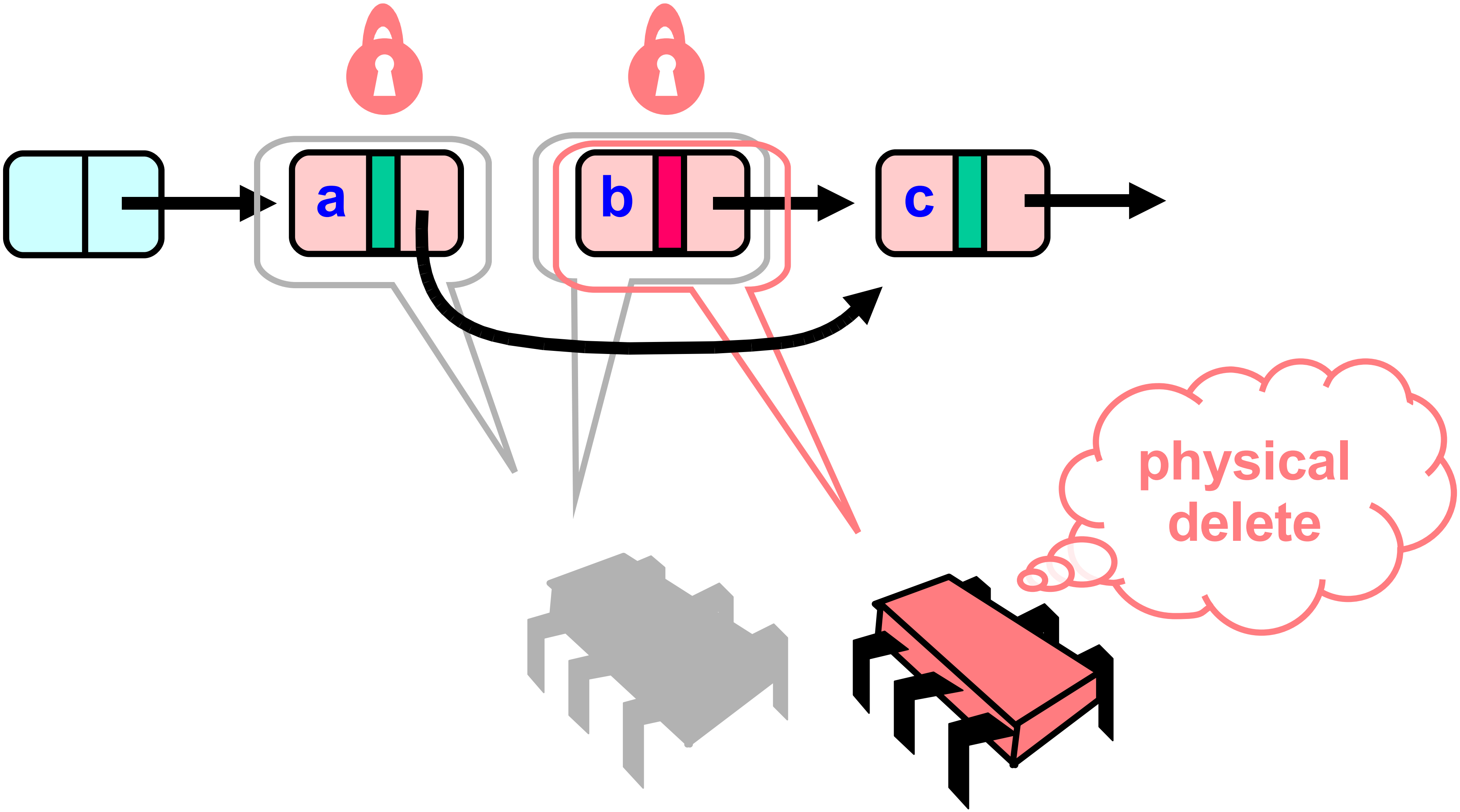
Lazy list algorithm



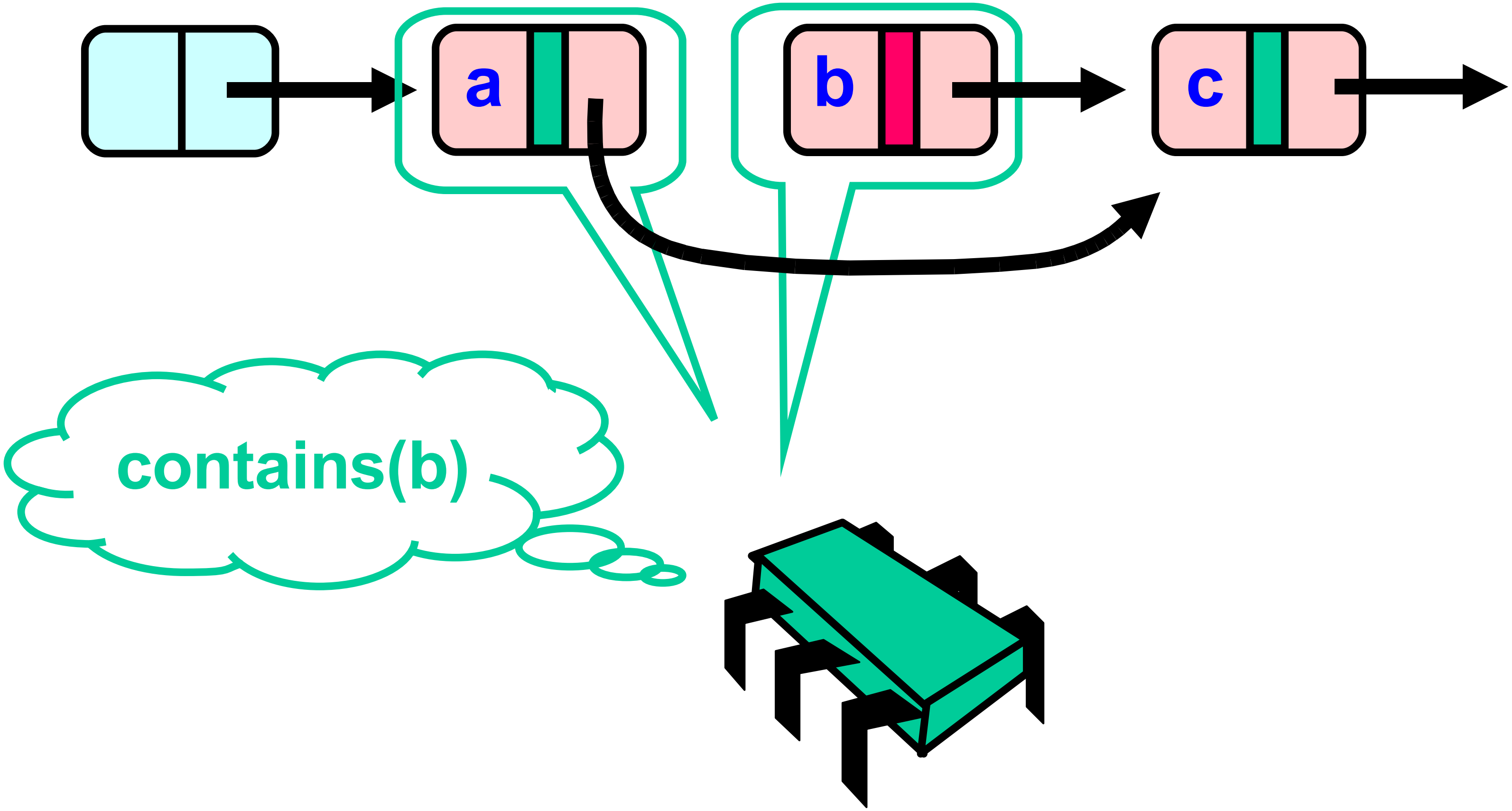
Lazy list algorithm



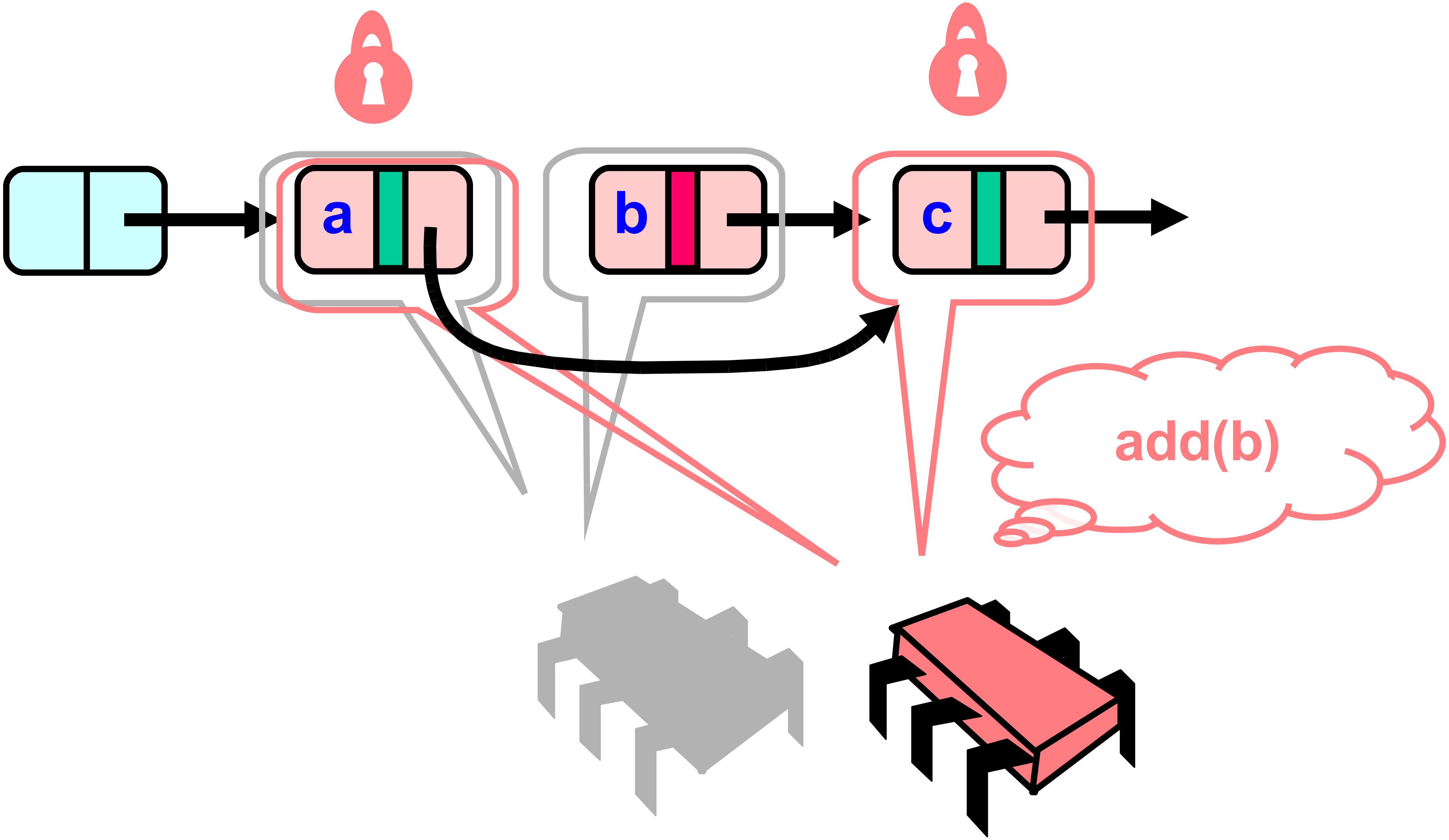
Lazy list algorithm



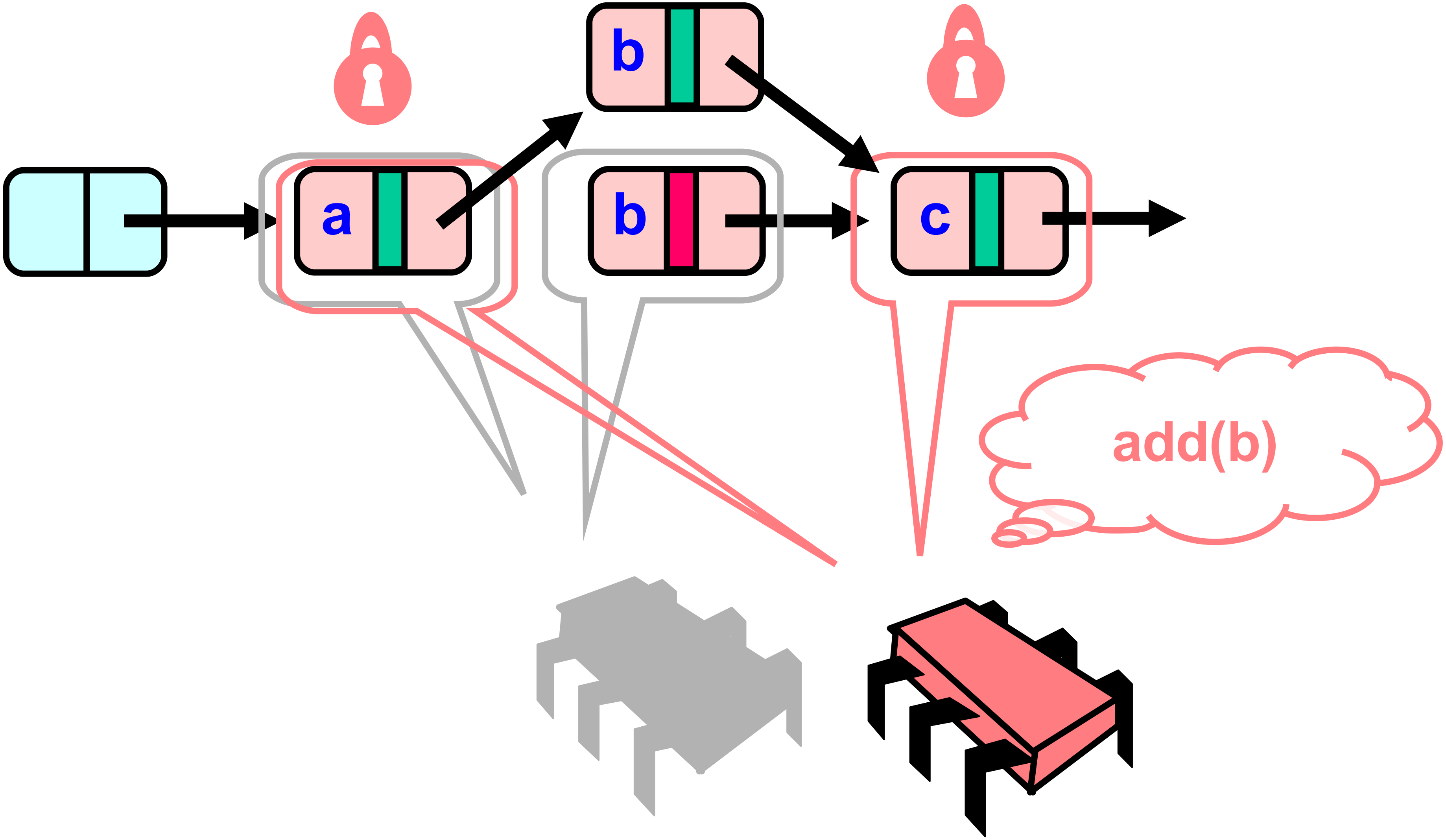
Lazy list algorithm



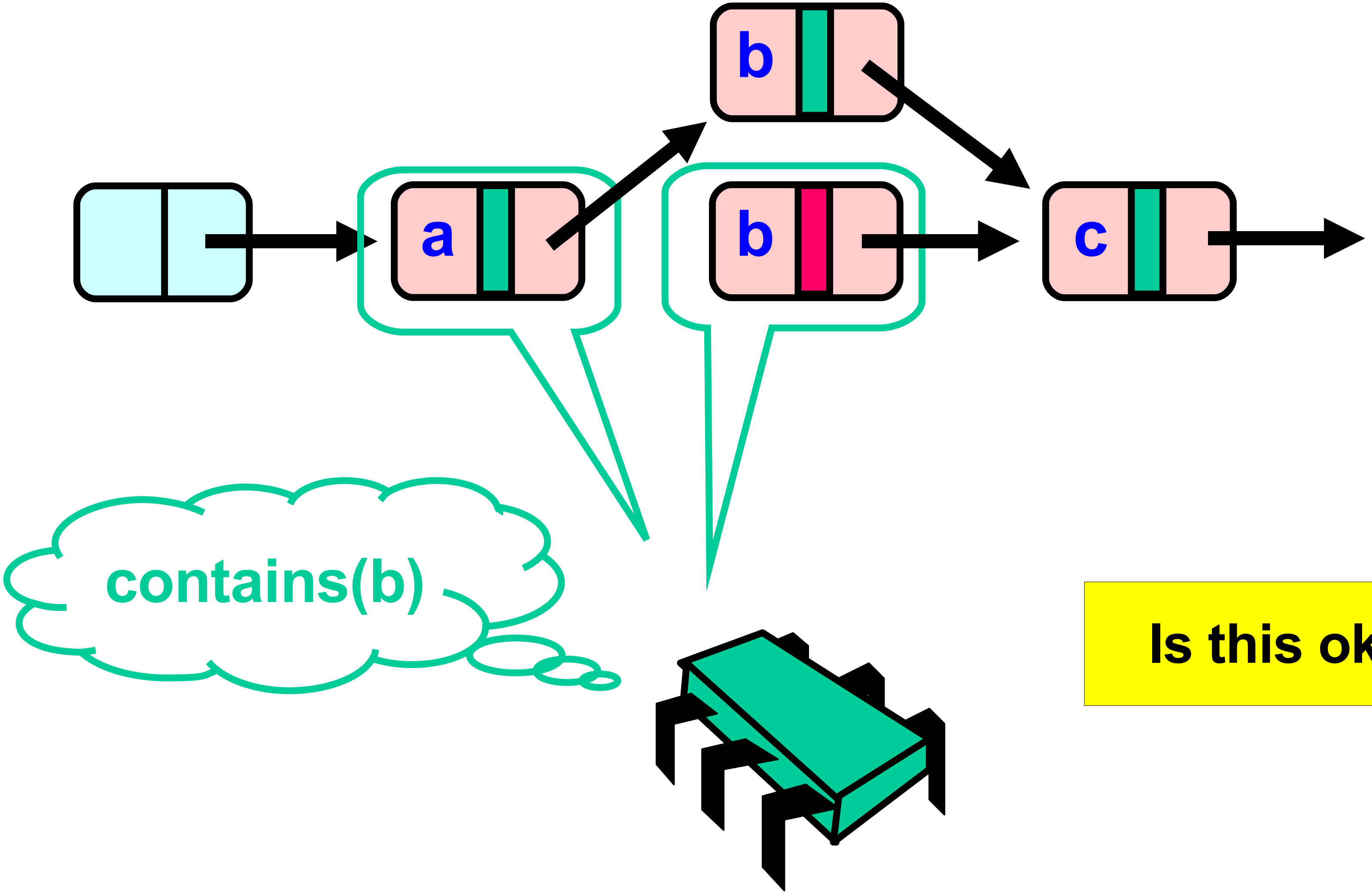
Lazy list algorithm



Lazy list algorithm

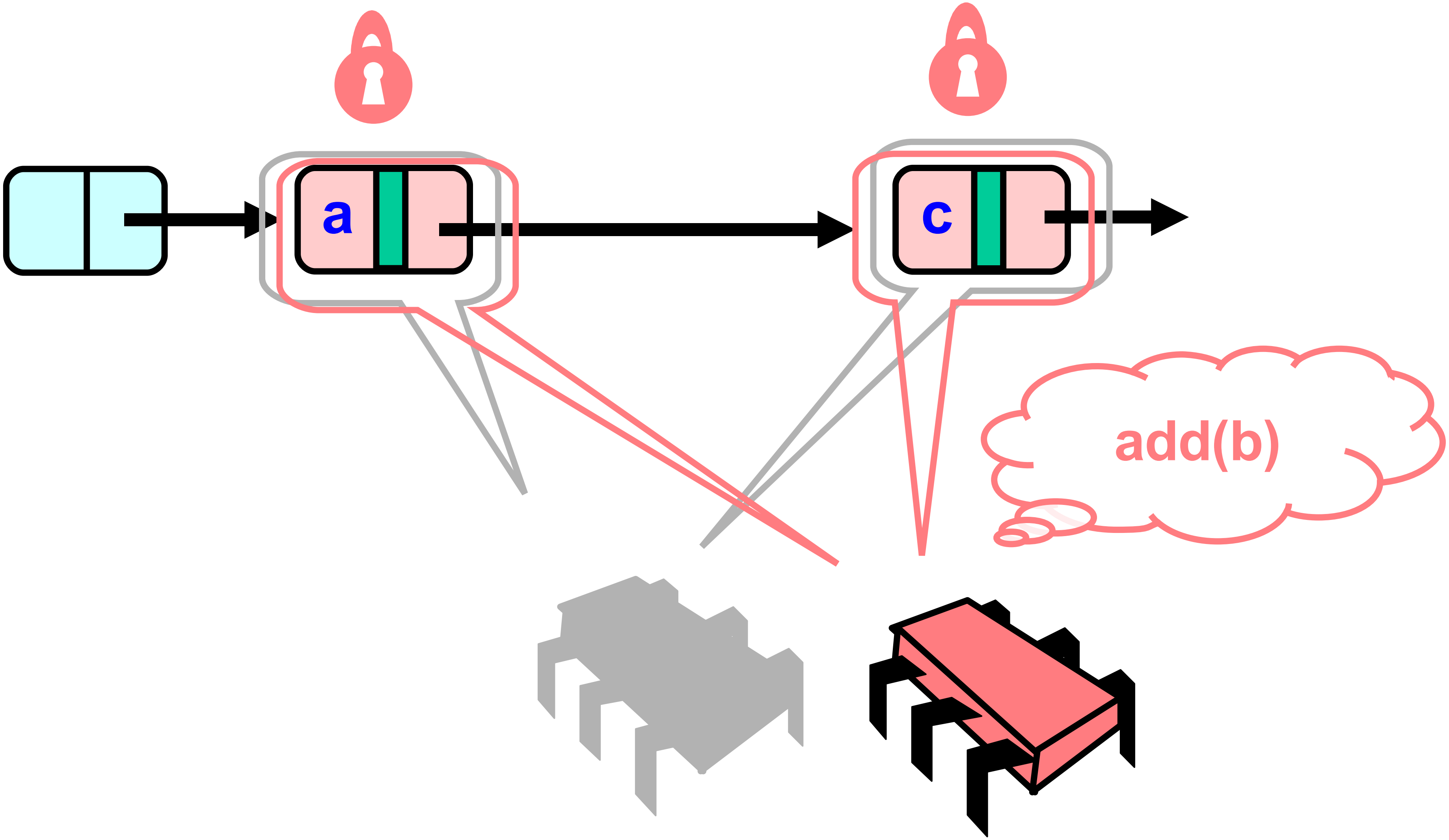


Lazy list algorithm

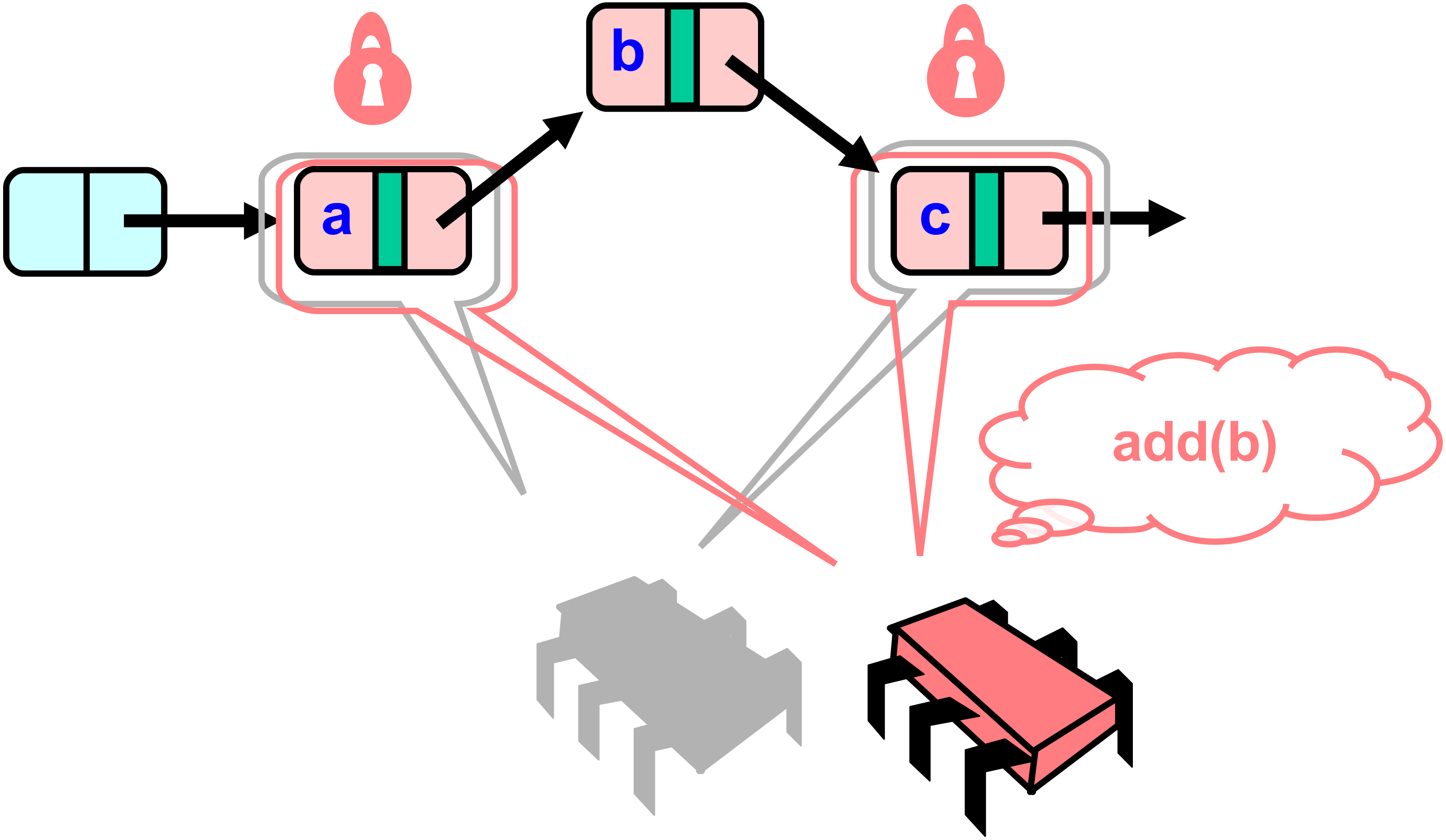


Is this okay?

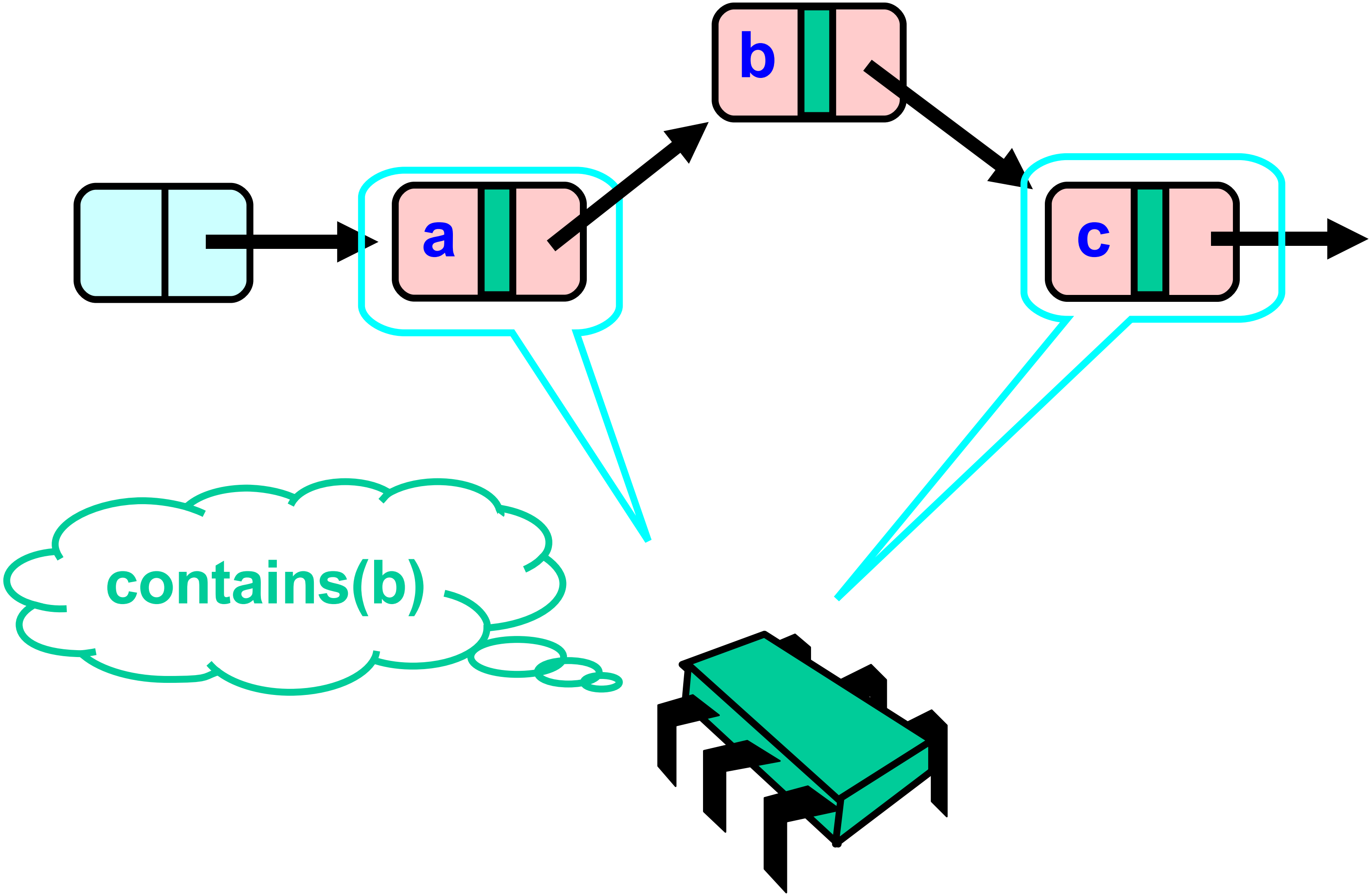
Lazy list algorithm



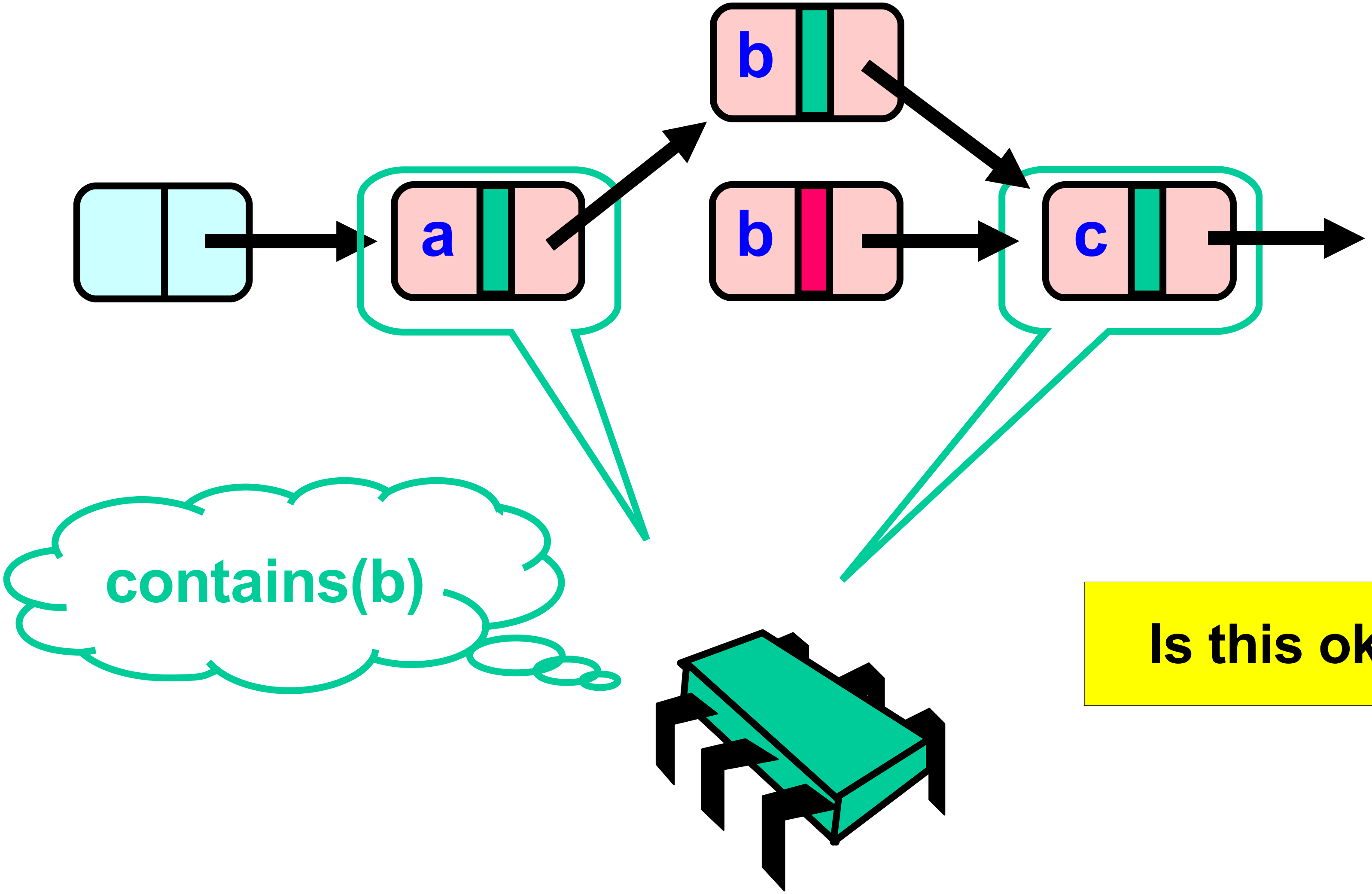
Lazy list algorithm



Lazy list algorithm



Lazy list algorithm



Is this okay?

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?

Review

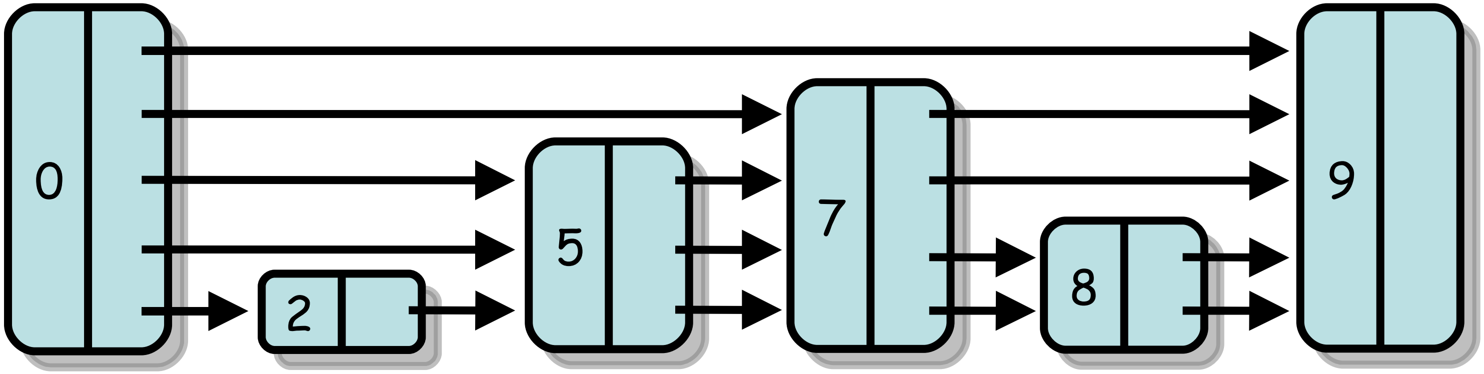
- Lock-free algorithm
  - “mark” nodes before removing from the list
    - marking is logical deletion
  - don't modify marked nodes
    - use CAS, mark and next pointer in same word
  - if encounter a marked node, help
    - physically delete node from list
  - if CAS fails, retry operation (except if you marked the node)

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

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



Summary

- Reduce granularity
- Two-phase locking
- Avoid deadlock by ordering locks
- Optimistic techniques
- Separate “logical” and “physical” changes
- Enable helping (by “announcing” intention)
- Optimize for the common case (usually reading)
  - analyze read-only operations separately
- Maintain invariants
- Weaken requirements: progress, invariants

Other techniques/issues

- Pointer swinging
  - maintain extra level of indirection
  - current version of object is never modified
  - to modify object: copy, modify copy, then “swing pointer”
  - okay for small objects
  - vary granularity to trade between efficiency and simplicity
  - beware of ABA problem (garbage collection helps here)
  - only lets you change one object at a time
- Keep copies
  - maintain an indicator of which is copy is “current”
  - like pointer swinging with pointer in reverse direction

Other techniques/issues

- Revocable locks/ownership records
  - like locks, but others can take away locks
    - they may undo your changes (aka rollback), or else help you finish
    - must leave undo or announcement info
  - contention can lead to “thrashing”
- Keep logs
  - remember operations done, derive state
    - keep recent version to reduce overhead
    - can roll back by truncating log
  - like universal construction from consensus

Other techniques/issues

- Contention management
  - queuing
  - backoff
  - priorities
- Composability
  - build algorithms/systems hierarchically
  - very hard with locks
- Weaker progress guarantees
  - obstruction-freedom
- Adaptive algorithms
  - overhead depends on actual rather than potential contention

Problems with locks

- Reduce concurrency
- Possibility of deadlock
- Convoying
- Priority inversion
- Difficult to manage
  - everyone must follow locking convention; hard to enforce/check
- Not composable

Problems with CAS or LL/SC

- Access only single location
- ABA problem (for CAS)
- Spurious failures (for LL/SC)
- Typically complex algorithms
- Helping interacts badly caching
- Contention management can break progress guarantees
- Difficult to compose (because of single-location limit)
  - bank transfer example

Transactional memory

- Raise level of abstraction
  - programmer specifies atomicity boundaries: transactions
  - system guarantees atomicity
    - commits if it can
    - aborts if not (roll back any changes)
    - possibly retry on abort
  - system manages contention (possibly separable functionality)
  - nested transactions compose
    - but large transactions may not commit

Transactional memory

- begin transaction
- commit
- “acquire”/“open” objects
  - differentiate reading and writing
- validate
- maintain roll-back functionality to support abort
- detect conflicts
  - contention manager resolves conflict
- retry policy

Transactional memory

- Herlihy and Moss (1993) proposed hardware TM
  - hardware; exploits cache coherence protocol
  - platform-dependent limits
- Shavit/Touitou (1995) proposed software TM
  - lock-free, not adaptive, very expensive (not practical)
- Revisited by many in early 2000s
  - DSTM (PODC 2003), OSTM (OOPSLA 2003), then lots more
  - SLE (2001), TCC (2004), then lots more
  - very active area (e.g., several new workshops)

Transactional memory

- Object-based vs word-based
- Hardware vs software (or combination)
- Blocking vs nonblocking
  - “user” blocking vs “system” blocking
  - obstruction-freedom vs lock-freedom
- Contention management
- Encounter-time vs commit-time acquire
- Eager vs lazy conflict detection (“zombie” transactions)
- Undo log vs write set
- Visible vs invisible vs “semivisible” readers
- Feature interaction: i/o, exceptions, conditional waiting, privatization, strong vs weak atomicity

Using transactional memory

- Simplified interface: atomic blocks
  - atomic { code }
  - automatic retry, obstruction-free progress guarantee

```
Q.enqueue(x)
  node = new Node(x)
  node.next := null
  atomic{
    oldtail = Q.tail
    Q.tail := node
    if oldtail = null then
      Q.head := node
    else
      oldtail.next := node
  }
```

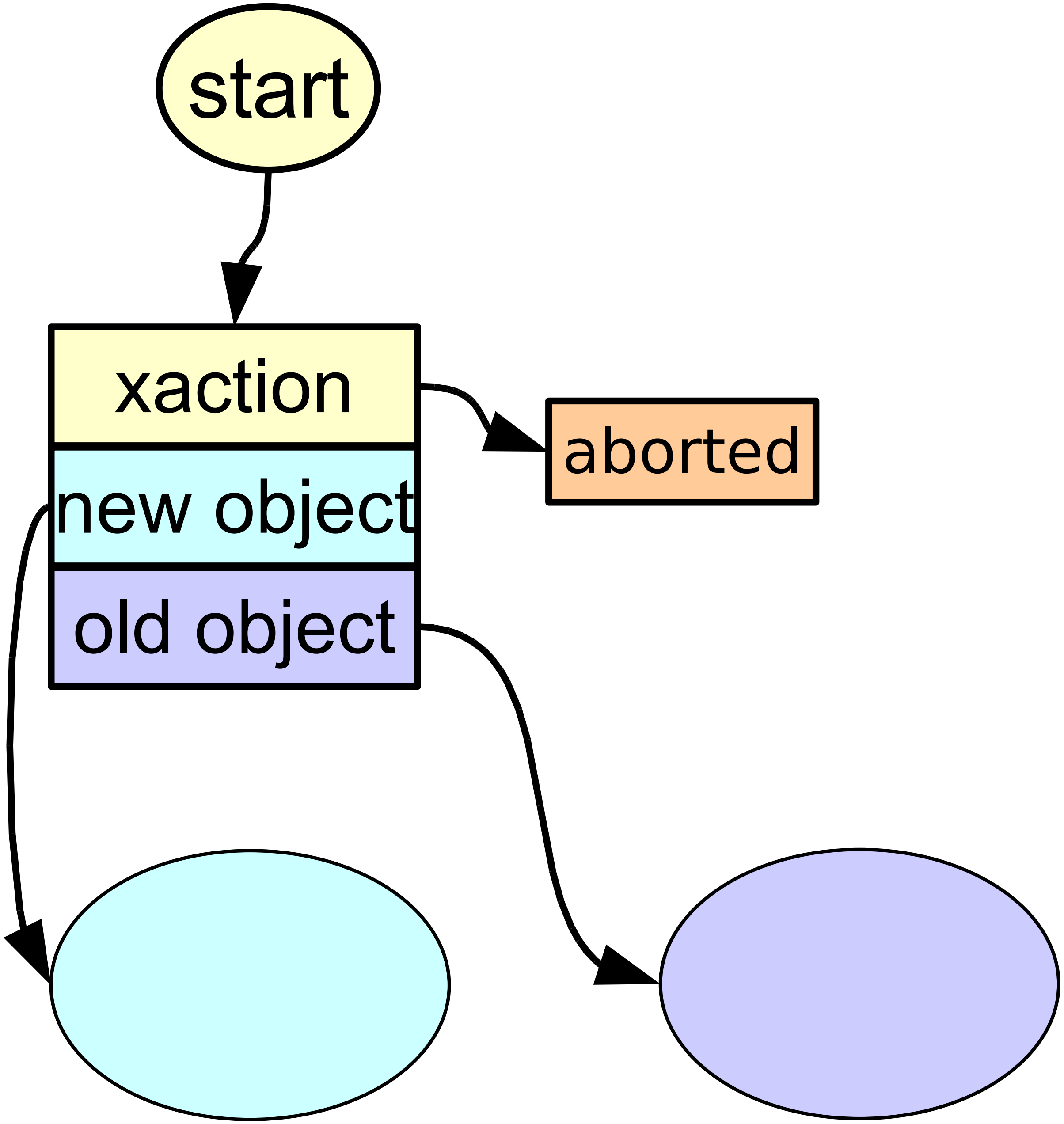
```
Q.dequeue()
  atomic{
    if Q.head = null then
      return null
    else
      node := Q.head
      Q.head := node.next
      if node.next = null then
        Q.tail := null
      return node.item
  }
```

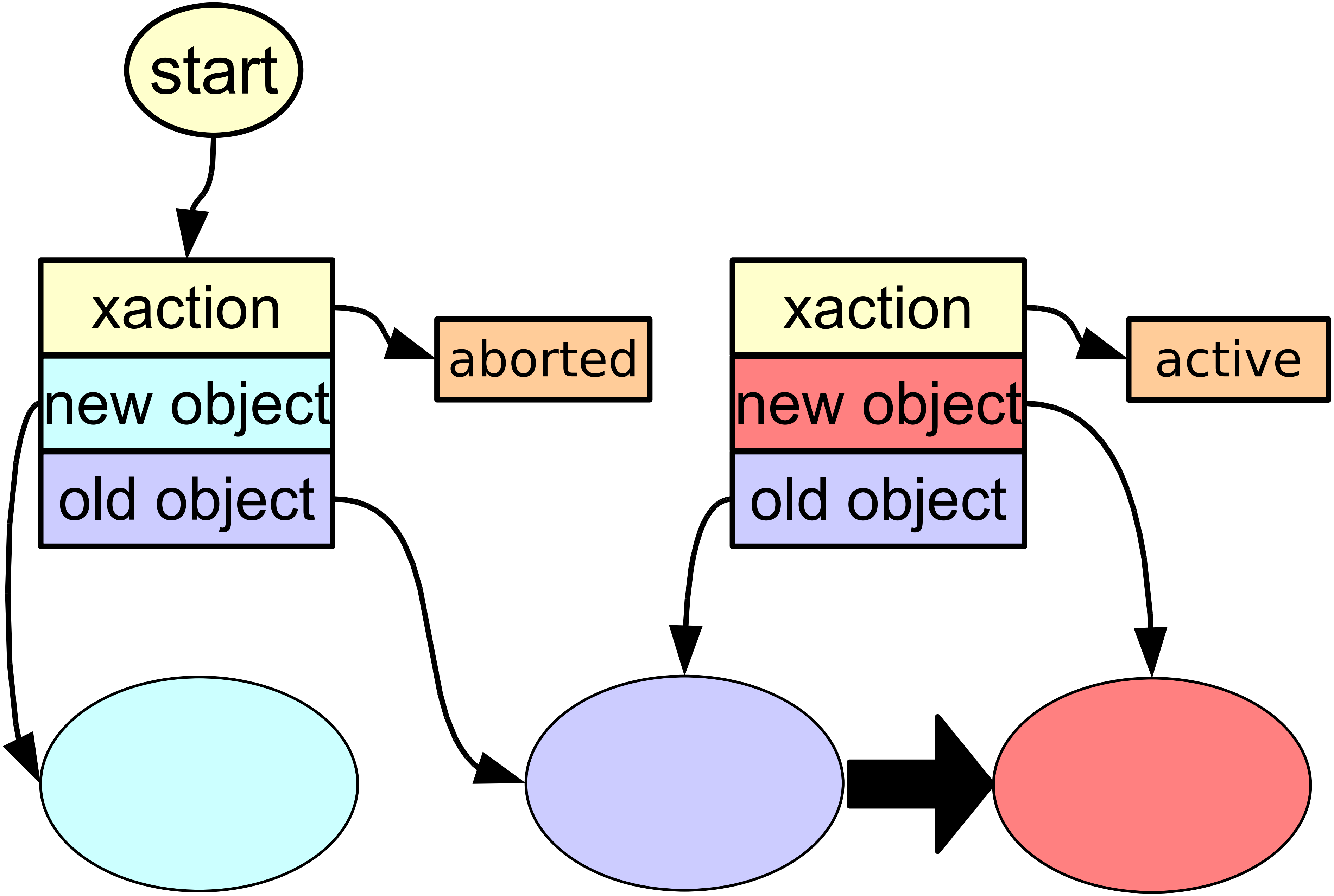
Implementing transactional memory

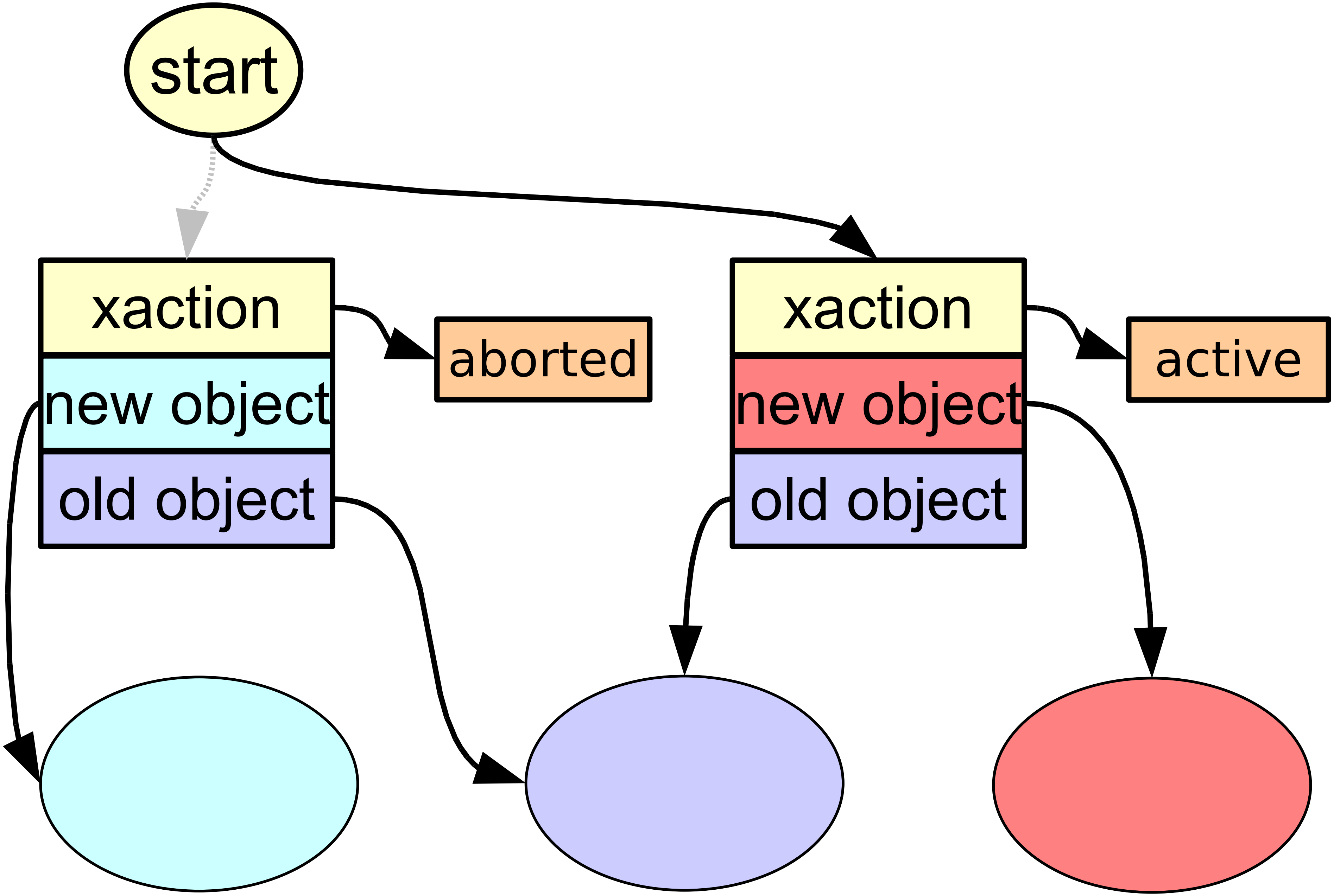
- Assume we can intercept access to objects
  - for object-based TM, exploit object infrastructure
  - for word-based TM, need compiler (or run-time) help
- TM implementation maintains shared metadata
  - with object for object-based TM, plus an additional small word for each transaction (could be just for active transactions)
  - for word-based TM, read sets and write sets
- No hardware support (use CAS)
- Different progress conditions

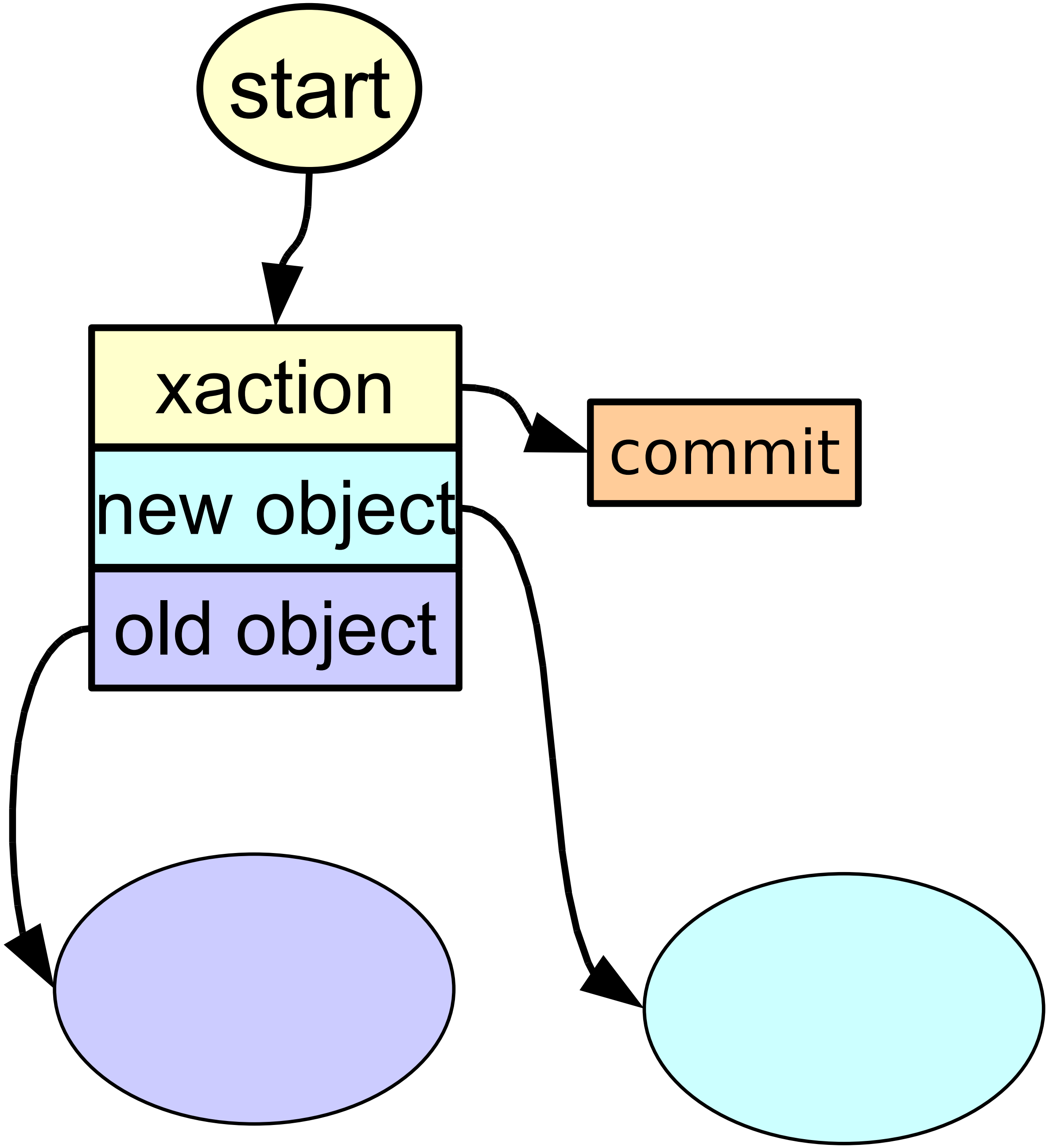
Dynamic STM (DSTM)

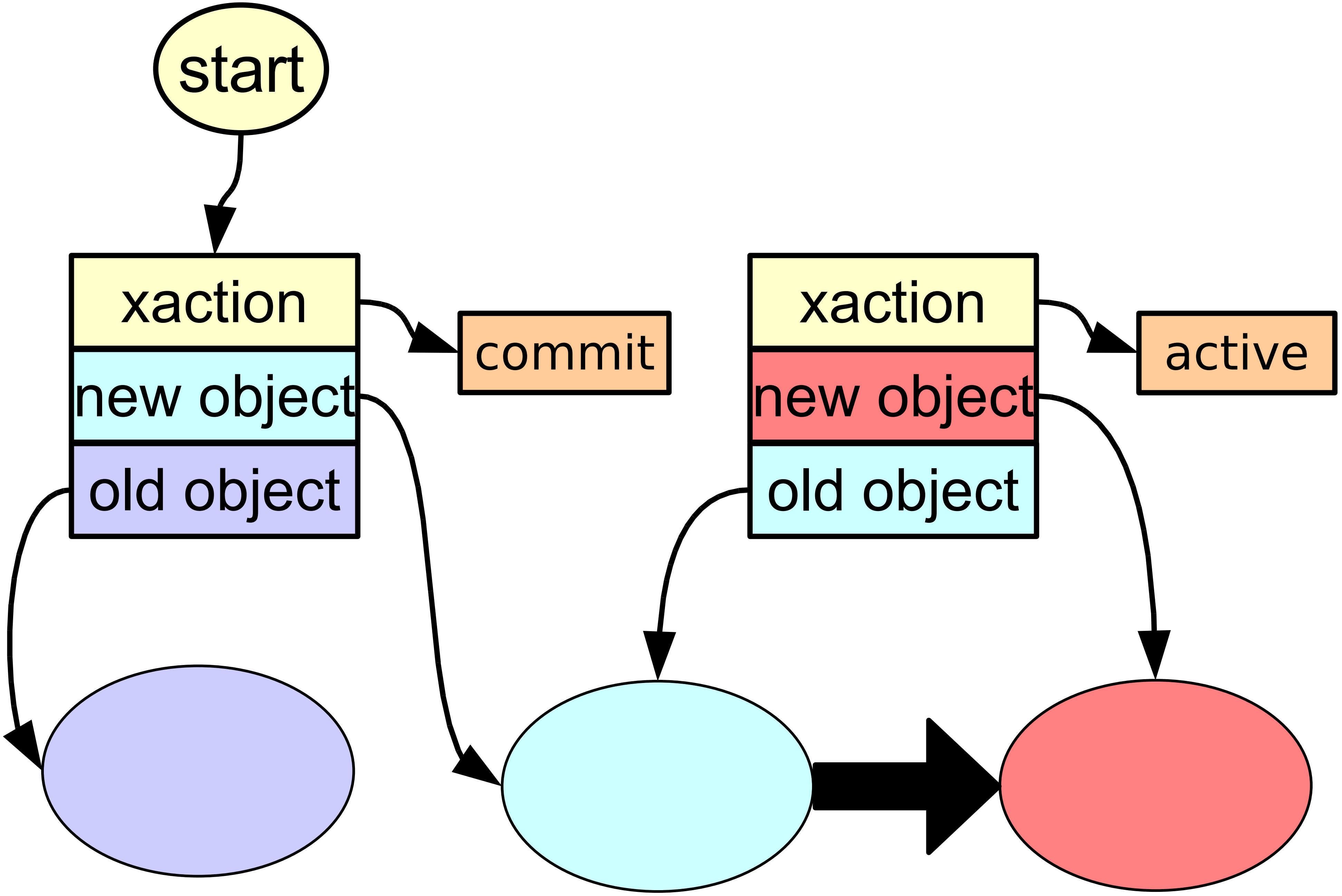
- object-based (Java^{TM} library)
- no locks (obstruction-free)
- supports dynamic allocation and access
- separable contention management
- pluggable implementations (2nd release)

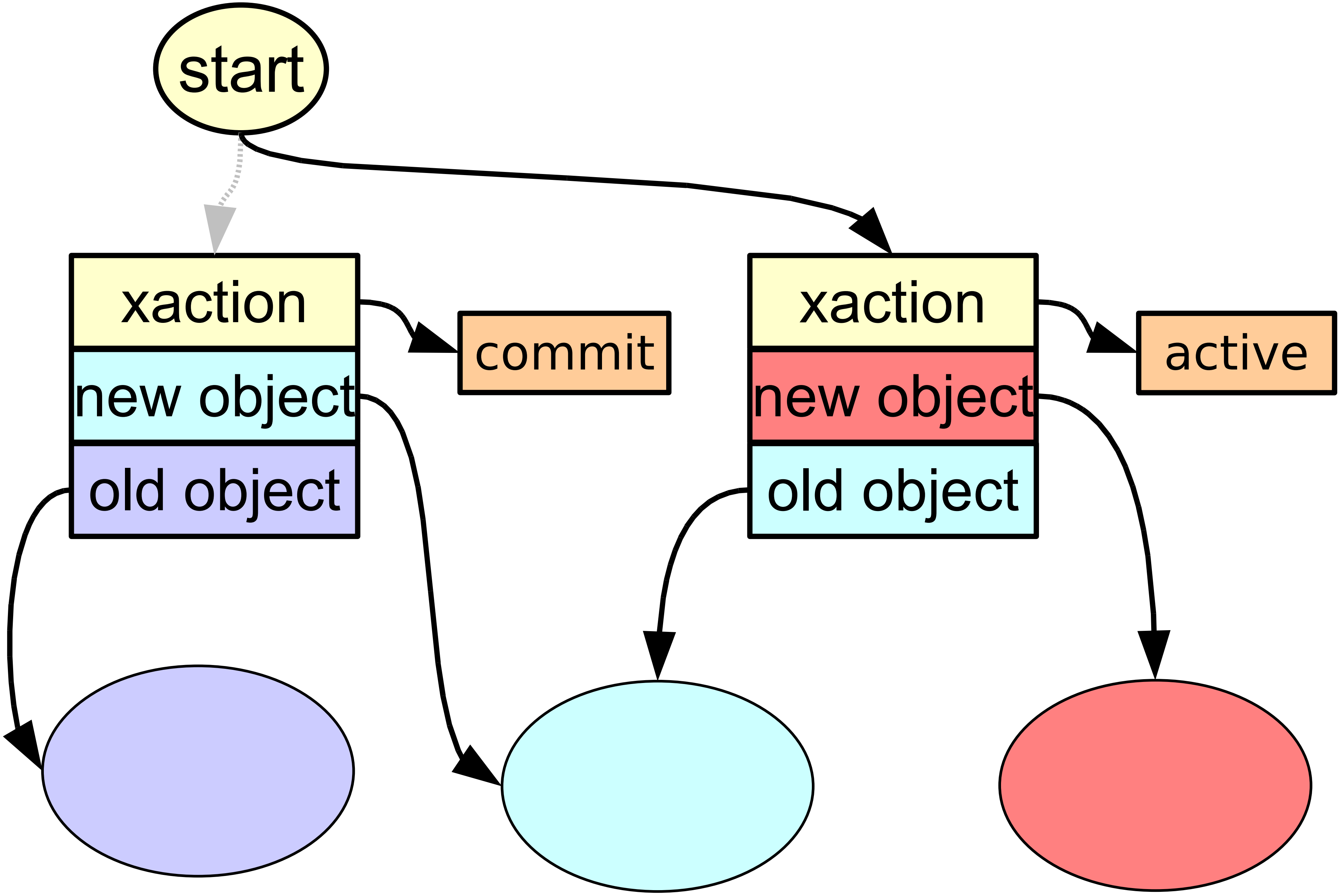

start
xaction
new object
old object
aborted

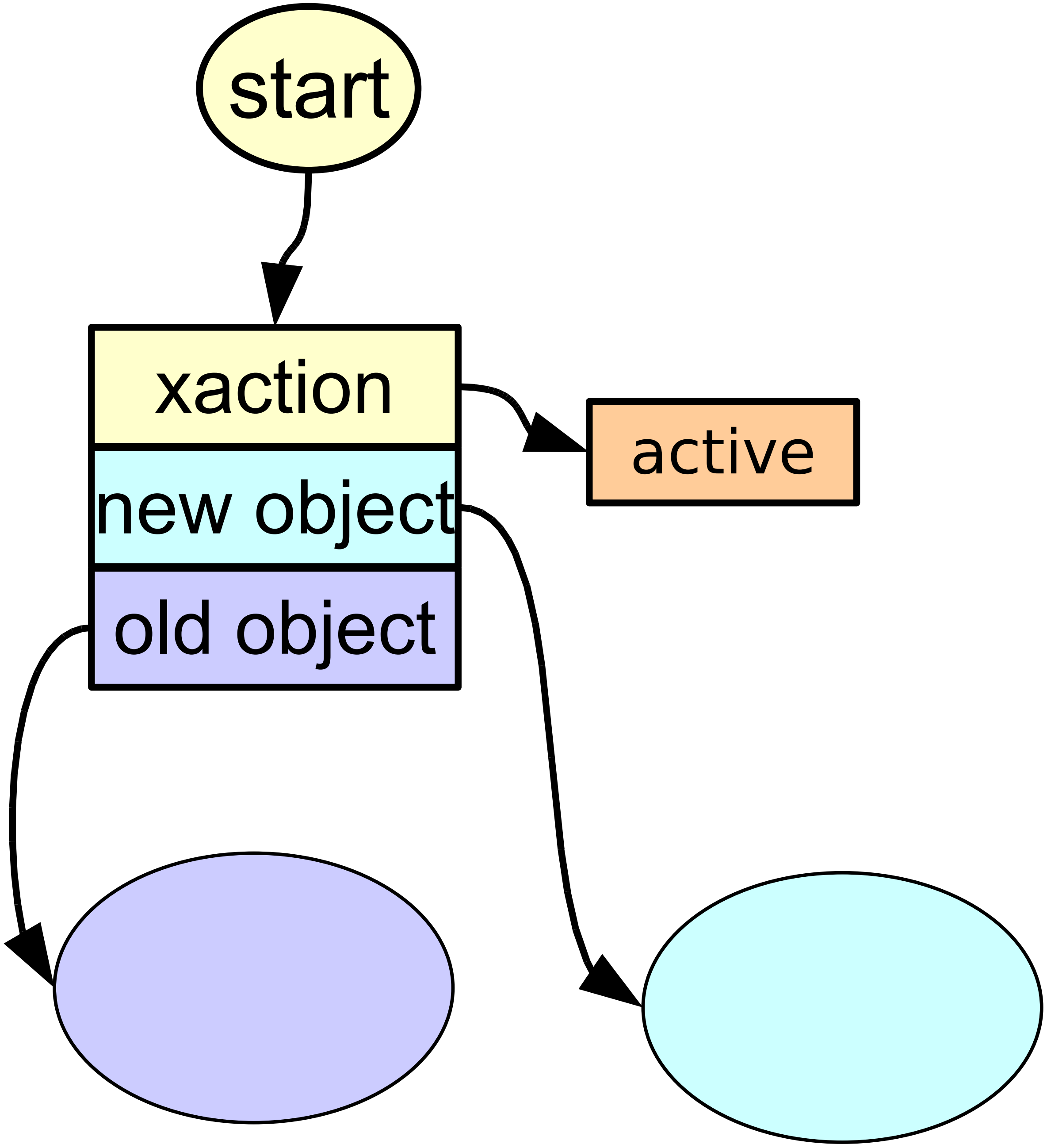

start
xaction
new object
old object
aborted
xaction
new object
old object
active

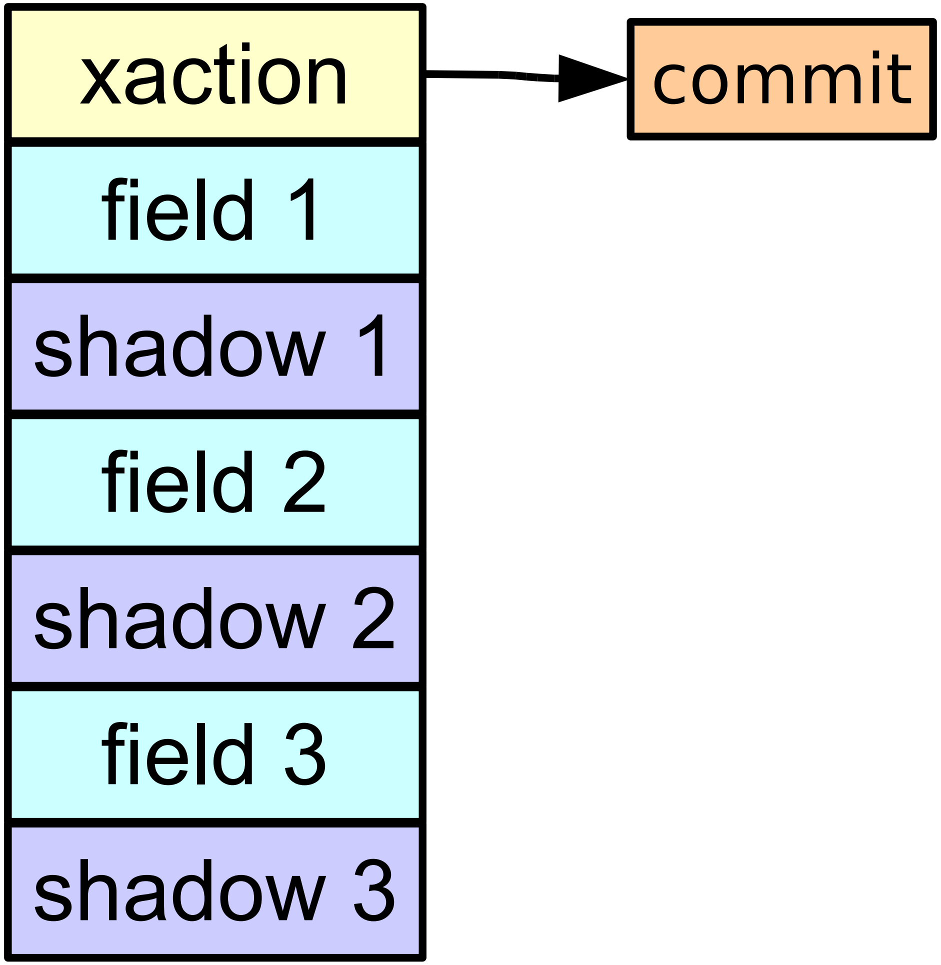

start
xaction
new object
old object
aborted
xaction
new object
old object
active

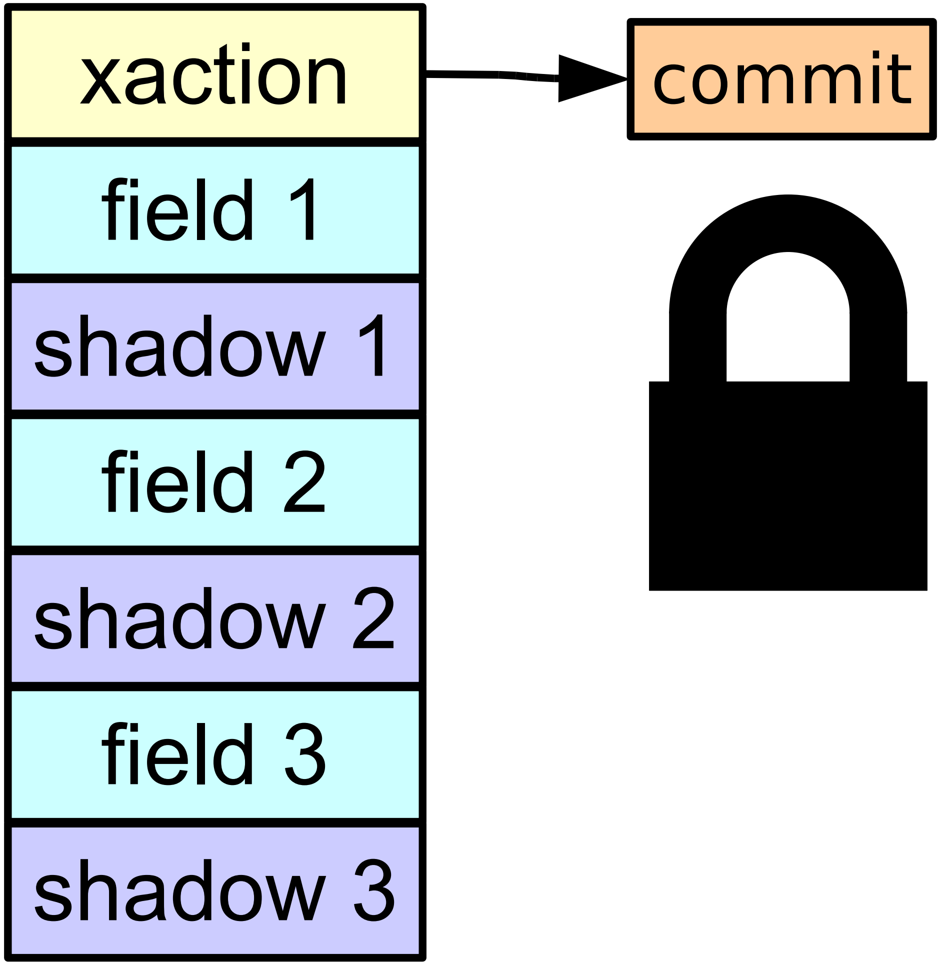

start
xaction
new object
old object
commit


start
xaction
new object
old object
commit
xaction
new object
old object
active


start
xaction
new object
old object
commit
xaction
new object
old object
active


start
xaction
active
new object
old object


xaction
commit
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3



xaction

field 1

shadow 1

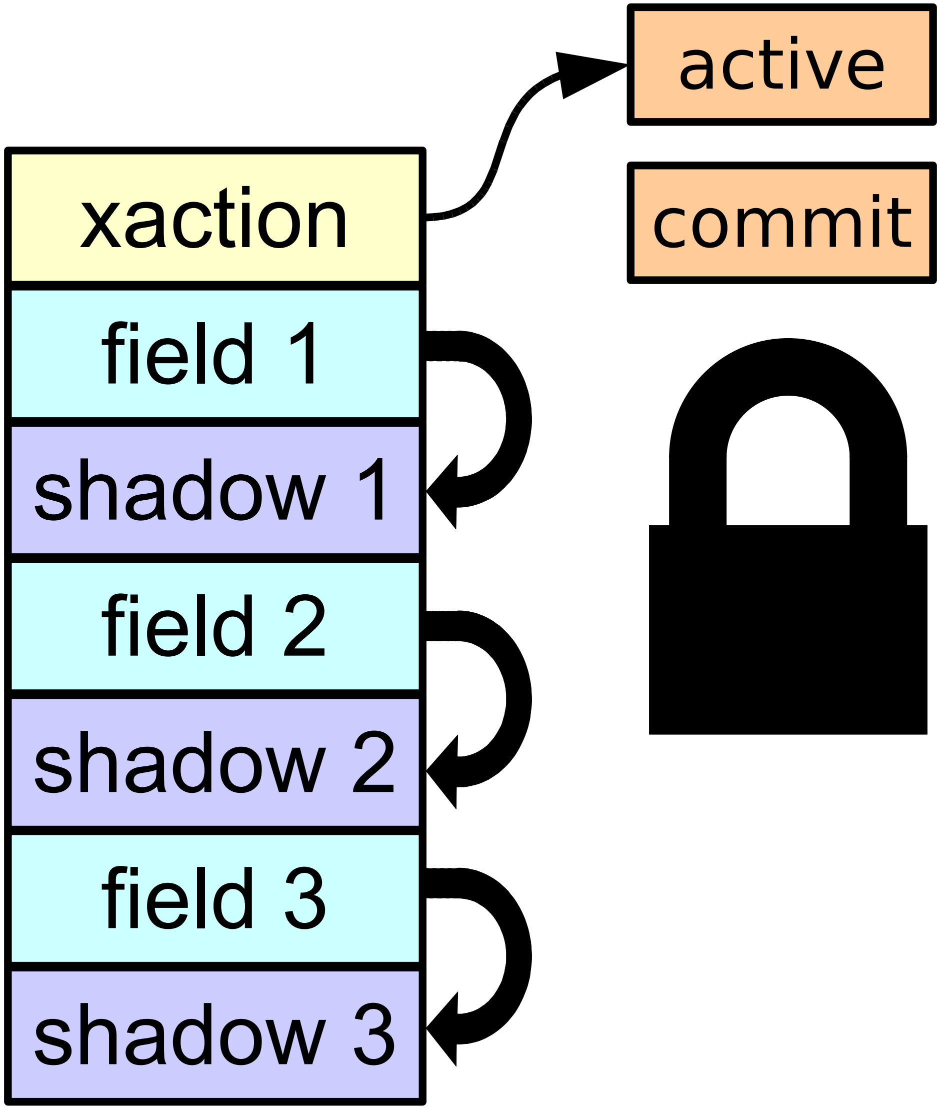
field 2

shadow 2

field 3

shadow 3

commit



xaction

field 1

shadow 1

field 2

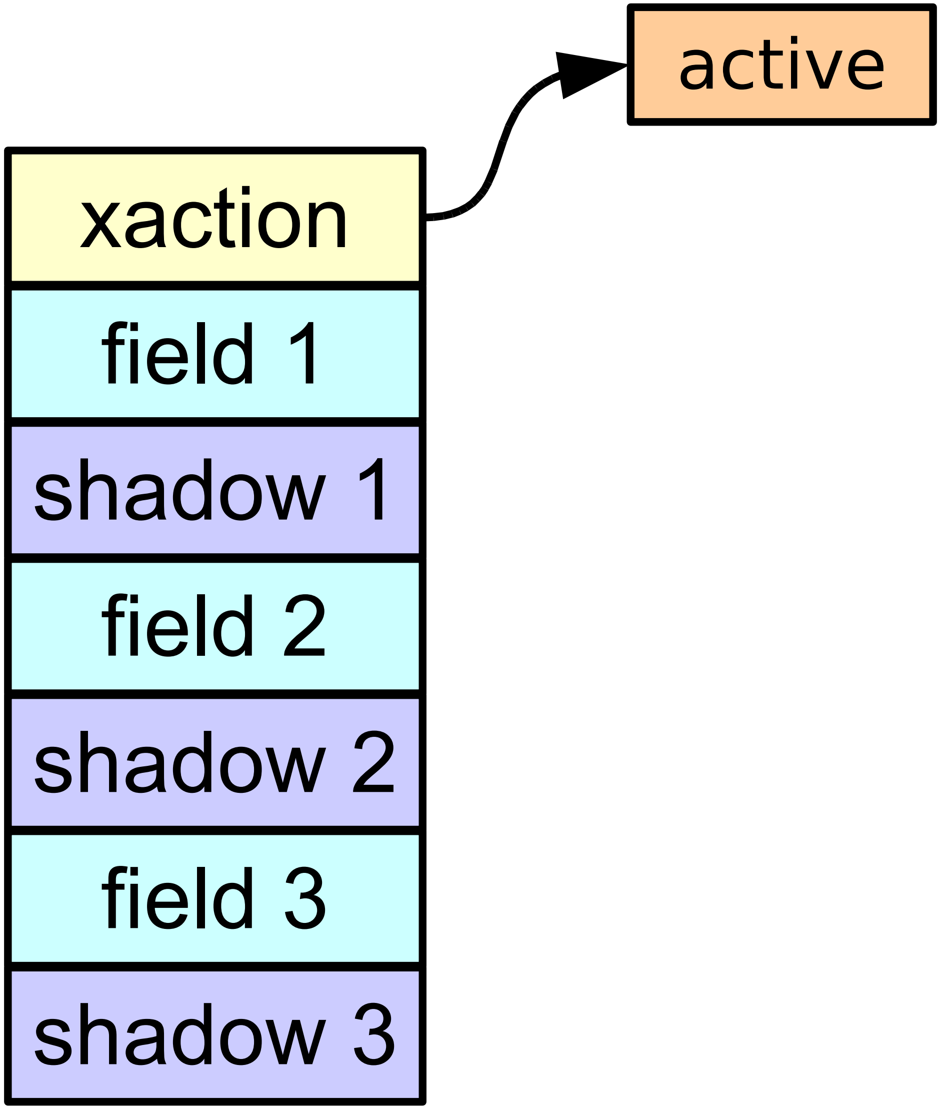
shadow 2

field 3

shadow 3

active

commit


active
xaction
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3

Transactional Locking II (TL2)

- Best (or one of the best) performing STM
  - also other nice properties beyond the scope of this lecture
- Lock-based, word-based STM
  - locks only held during commit phase (not executing user code)
- Uses global version number (potential bottleneck)
  - updated by each writing transaction (but could relax this)
- Every location also has version number
  - transaction that last wrote it

Transactional Locking II (TL2)

- Read global version counter
  - store locally: rv (for read version)
- Run transaction “speculatively”
  - track which locations are read and written
  - when location first accessed, check version counter
  - write values into write set
  - read values into read set (so transaction gets consistent reads)
    - if value was written by transaction, get value from write set
- At commit
  - lock write set
  - increment global version counter
  - validate read set
  - write-back values
  - release locks

Combining hardware and software

- Hardware-assisted transactional memory
  - new required hardware
  - hardware support can accelerate implementation
- Hybrid transactional memory
  - STM that can work with HTM
    - hardware transactions and software transactions must “play nicely”
    - can be used now (with no hardware support)
    - can exploit HTM support with little change
  - phased transactional memory
    - can switch dynamically between using STM and HTM

Next time

- Asynchronous networks vs asynchronous shared memory
- Agreement in asynchronous networks
  - Paxos algorithm
- Reading:
  - Lamport paper: The Part-Time Parliament
  - Lynch, Chapter 17