# Concepts and Models of Parallel and Data-centric Programming

Shared Memory XII

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

# Outline

0. Organization
1. Foundations
2. **Shared Memory**
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

o. Lock-free Synchronization
p. SIMD / Vectorization
q. Intrinsics for SIMD
r. Parallel STL for SIMD and Parallelism

# Review

# Set Interface

- Unordered collection of items
  - No duplicates
- Methods
  - **`add(x)`** put **`x`** in set
  - **`remove(x)`** take **`x`** out of set
  - **`contains(x)`** tests if **`x`** in set

```
template<typename T> class Set {
public:
  bool add(T x);
  bool remove(T x);
  bool contains(T x);
};
```

# List-based Set

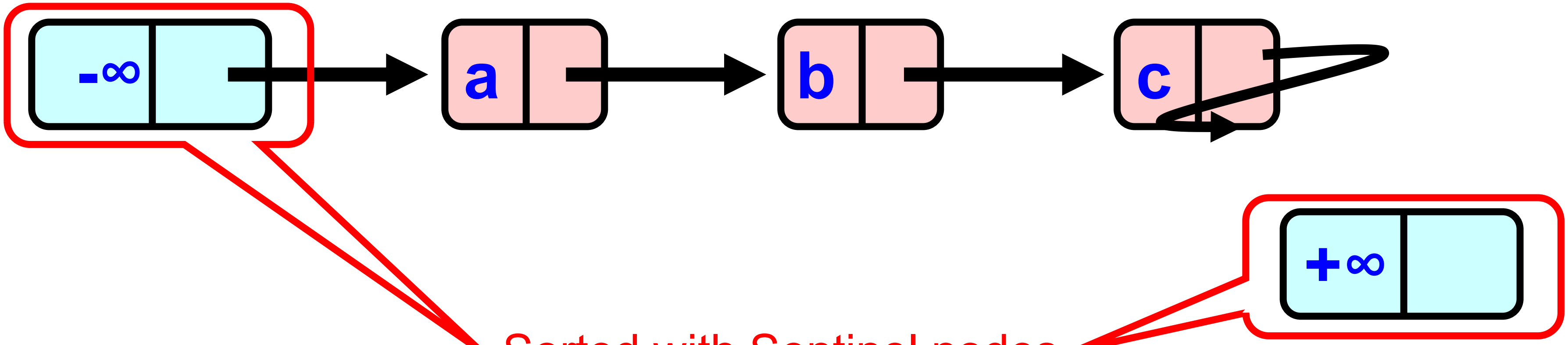
- List node:

```
template<typename T> class Node {
public:
      T item;
      int key;
      Node* next;
};
```

Item of interest

Hash

Reference to next node



Sorted with Sentinel nodes
(min & max possible keys)

# Comparison

| | add() | remove() | contains() |
|---|---|---|---|
| Coarse-grained Sync. | whole object locked | whole object locked | whole object locked |
| Fine-grained Sync. | chain of pair-wise acquire and release | chain of pair-wise acquire and release | no lock |
| Optimistic Sync. | lock targets only, but validate with traversal | lock targets only, but validate with traversal | chain of acquire and release |
| Lazy Sync. | mark and lock targets only, retry if conflict | mark and lock targets only, retry if conflict | no lock (check for marker) |
| Lock-free Sync. | no lock | no lock | no lock |

# Lock-free Synchronization

# Goal of lock-free data structure

- 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

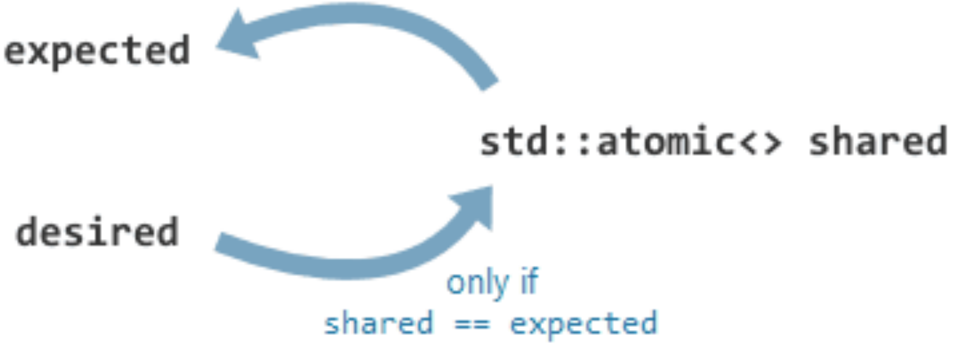
- Implies that implementation can’t use locks

- Lock-free List
  - Next logical step
    - Wait-free contains()
    - lock-free add() and remove()
  - Use only `atomic` functionality

# Implementation sketch

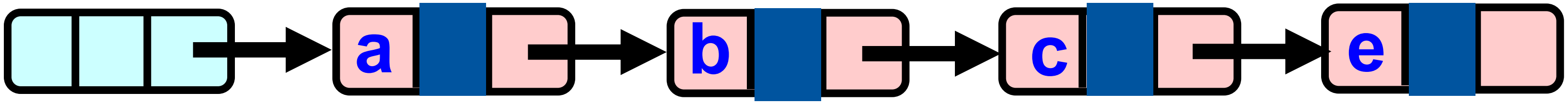
- Use `std::atomic<>::compare_exchange_weak()`
  - Use of CAS: compare-and-swap atomic instruction
    - compares the value of a memory location with a given value and, only if they are the same, modifies the content of that memory location to a new given value

- Atomically
  - Swing reference and
  - Update marker



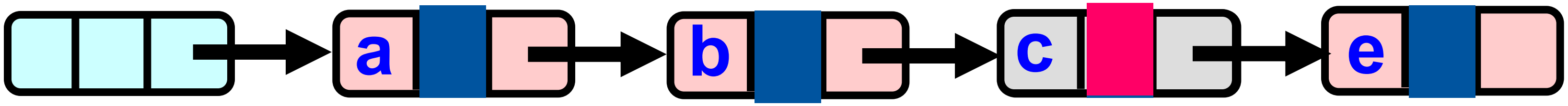
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor’s pointer

# Problem (if CAS used only for pointer)

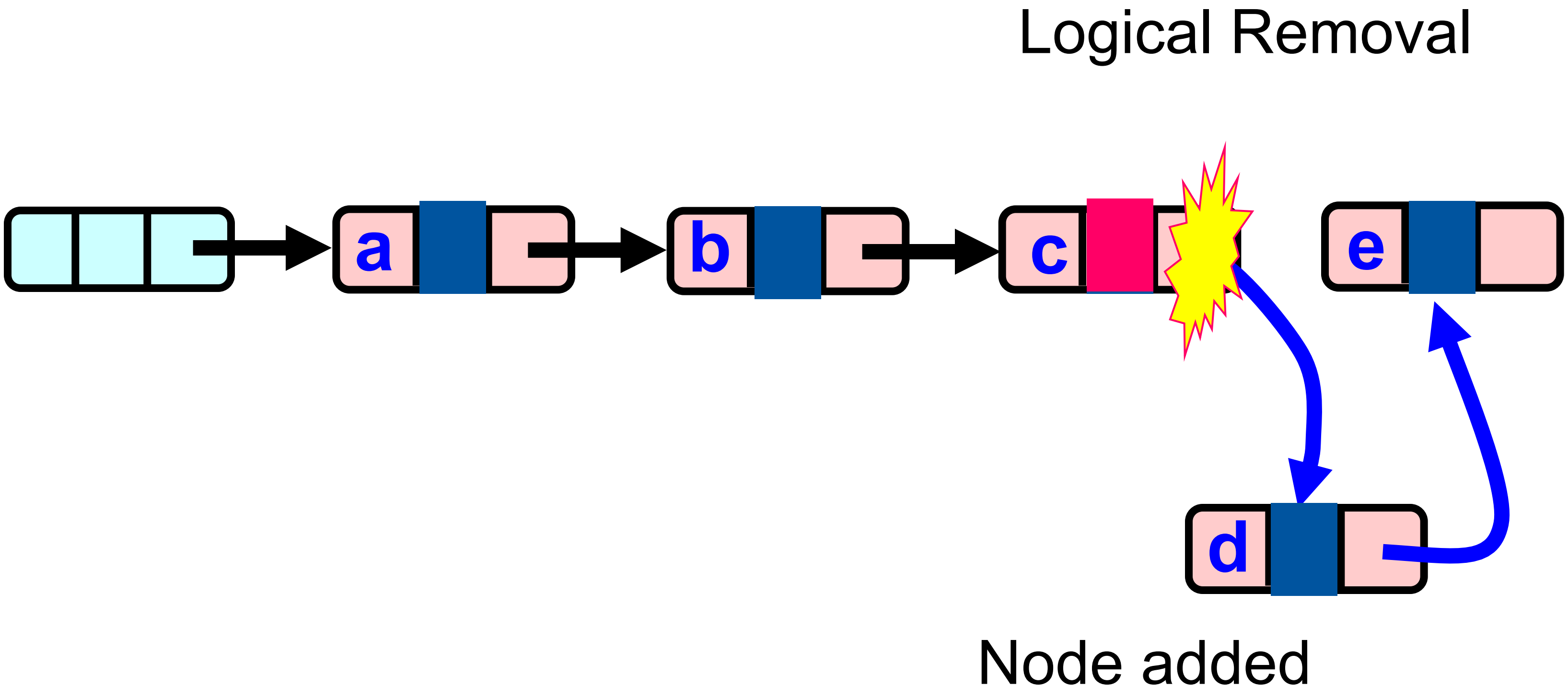


# Problem (if CAS used only for pointer)

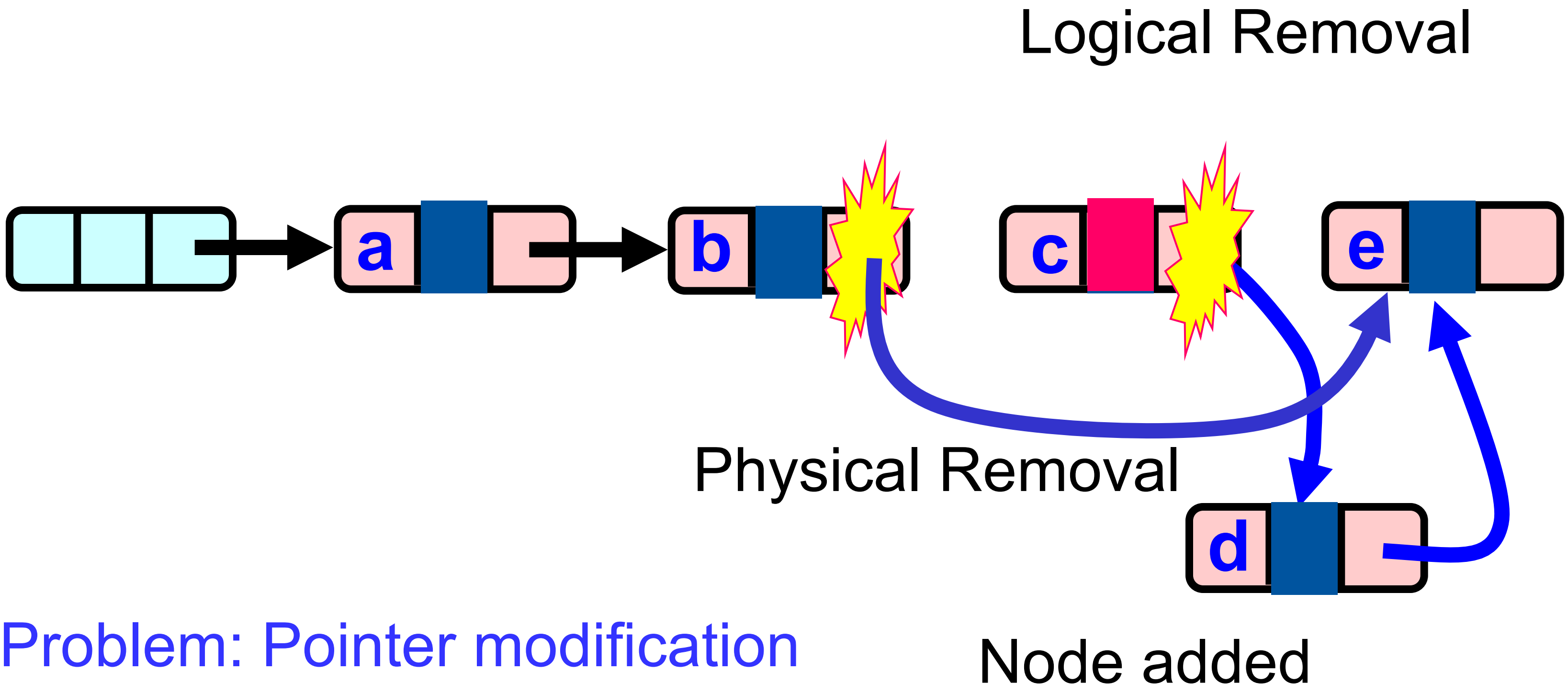
Logical Removal



# Problem (if CAS used only for pointer)

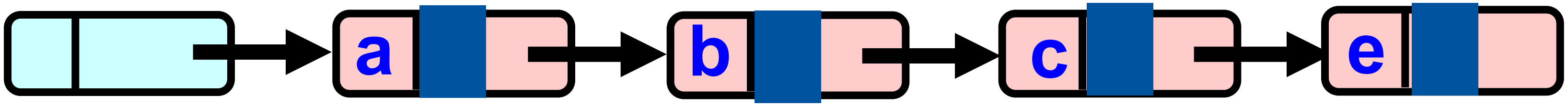


# Problem (if CAS used only for pointer)



Problem: Pointer modification and marker update are not considered together.

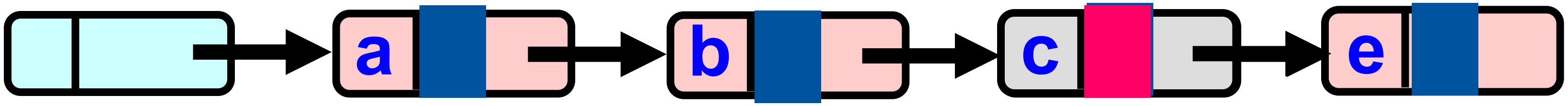
# Implementation sketch



Marker and Pointer
are CASed together

# Implementation sketch

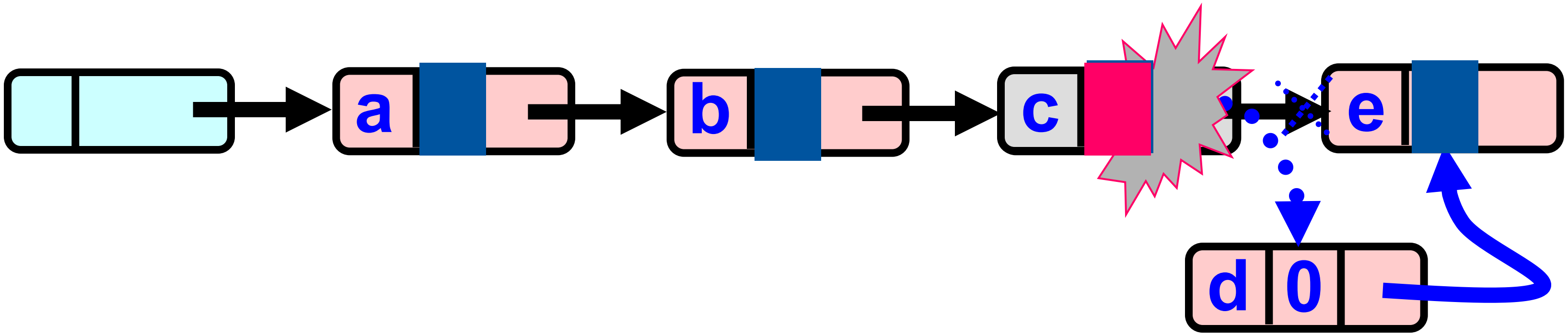
Logical Removal = Set Mark Bit



Marker and Pointer are CASed together

# Implementation sketch

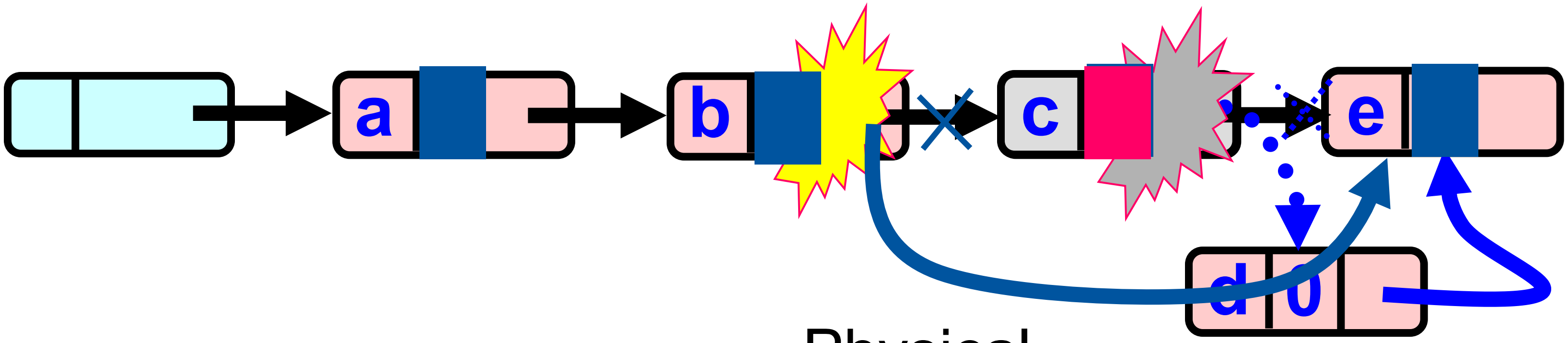
Logical Removal =
Set Mark Bit



Marker and Pointer are CASed together

Fail CAS: Node not added after logical Removal

# Implementation sketch



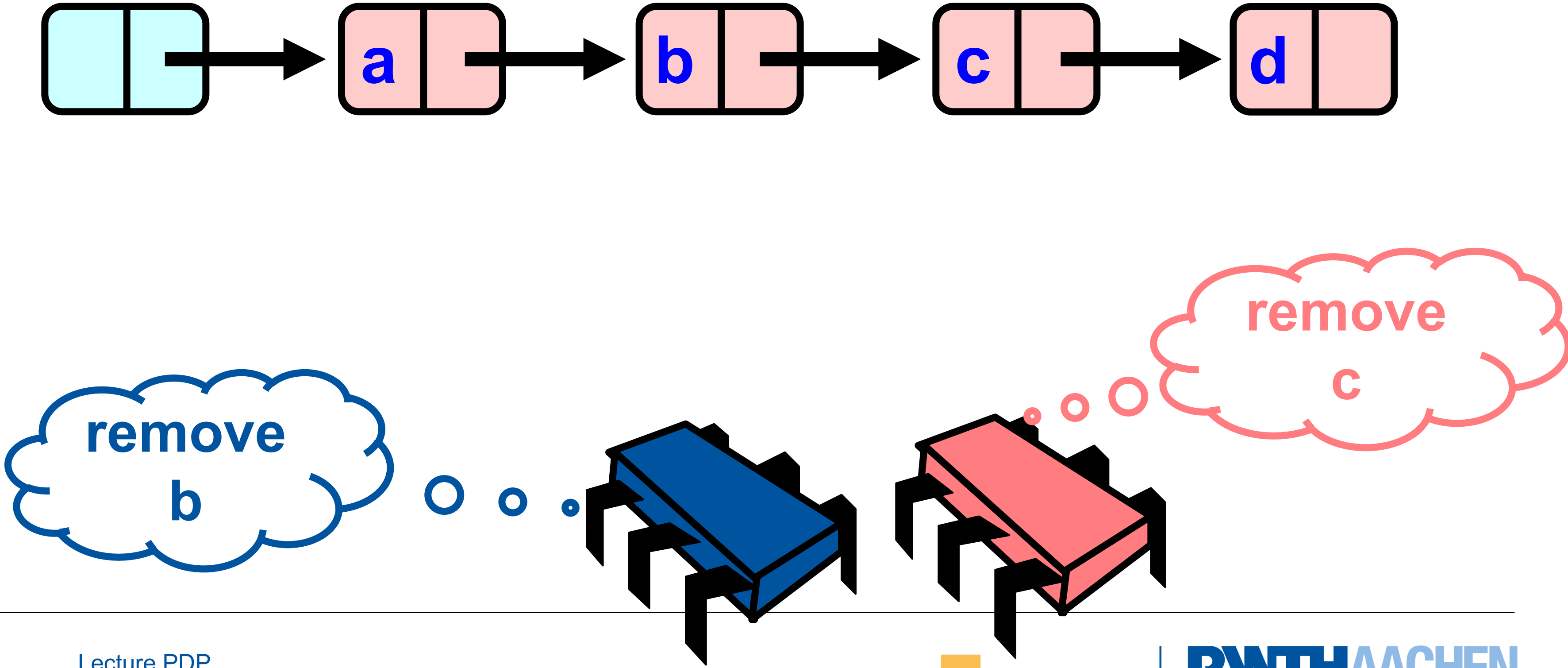
Logical Removal = Set Mark Bit

Physical Removal CAS

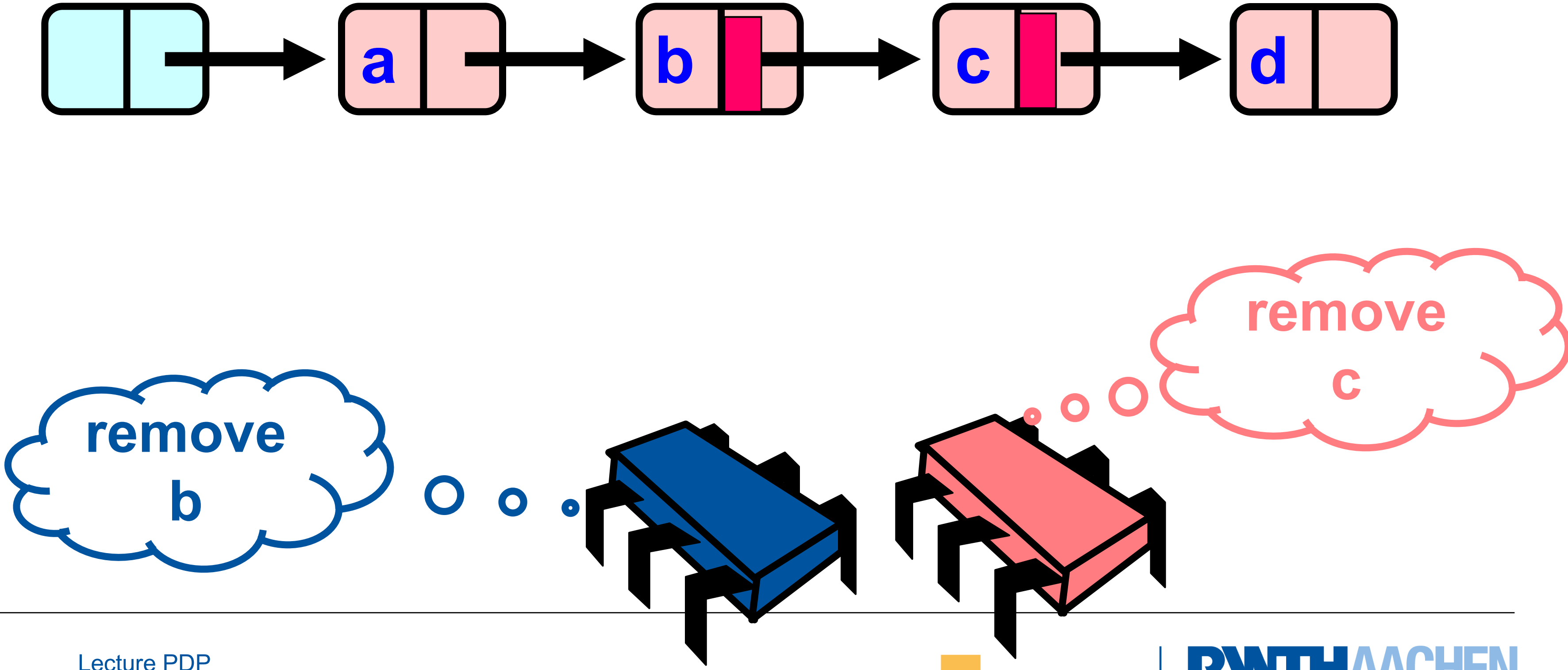
Fail CAS: Node not added after logical Removal

Marker and Pointer are CASed together

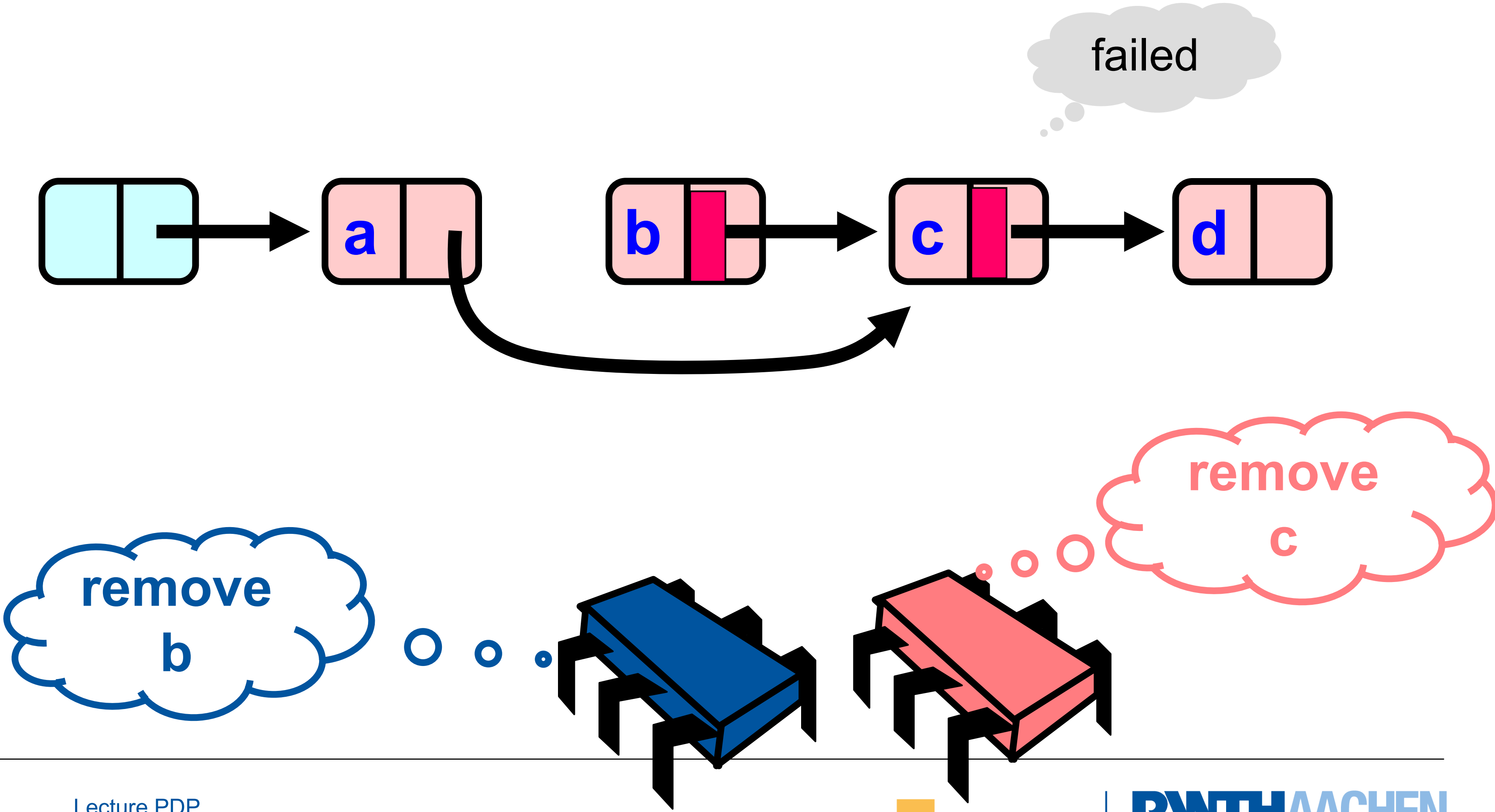
# Illustration: removal



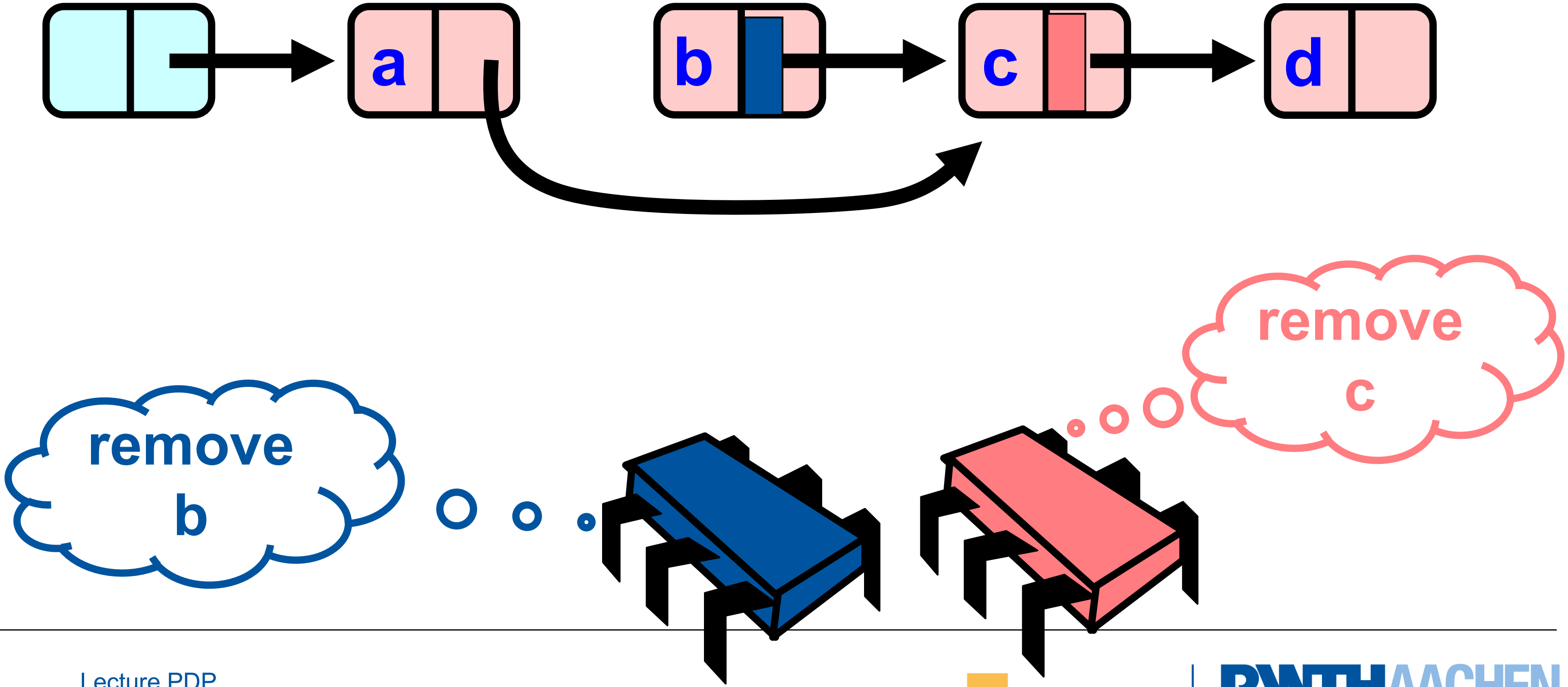
# Illustration: removal



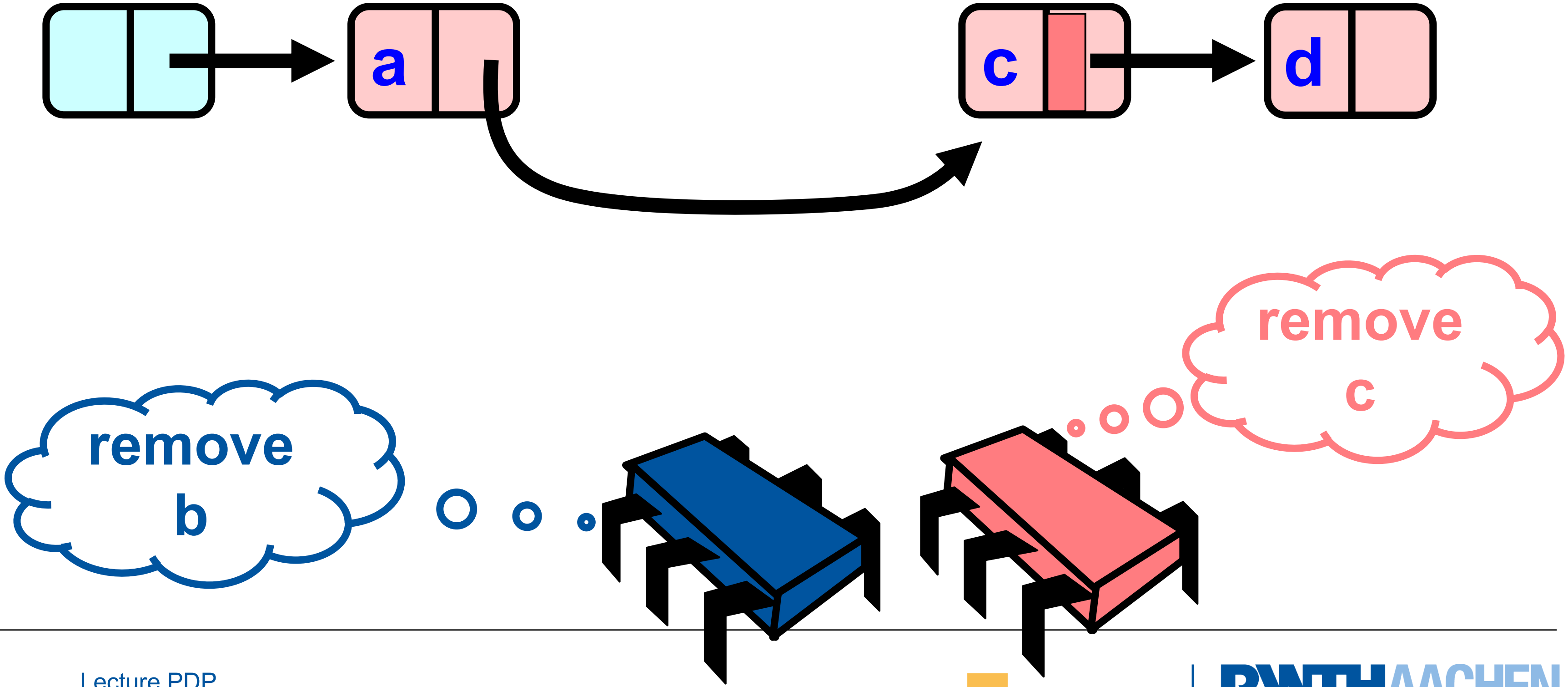
# Illustration: removal



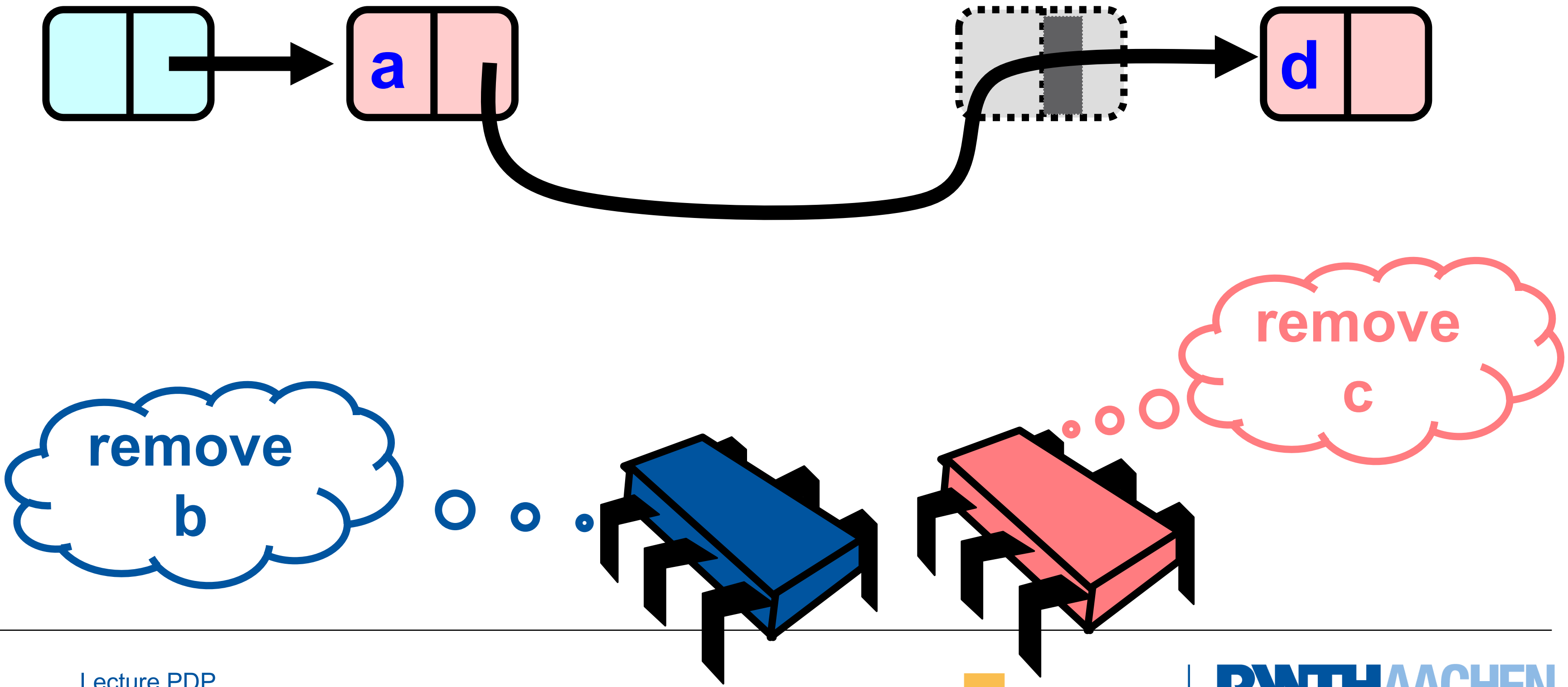
# Illustration: removal



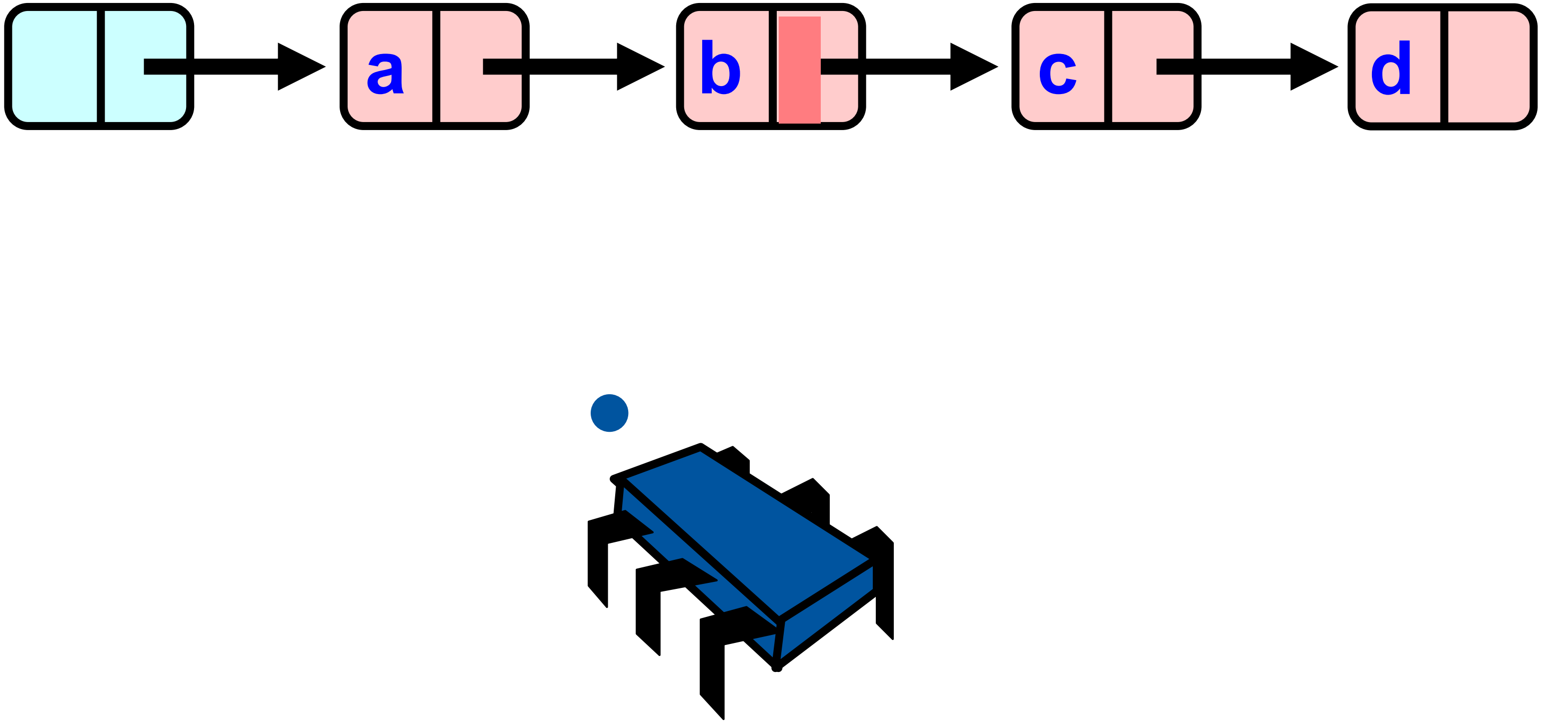
# Illustration: removal



# Illustration: removal

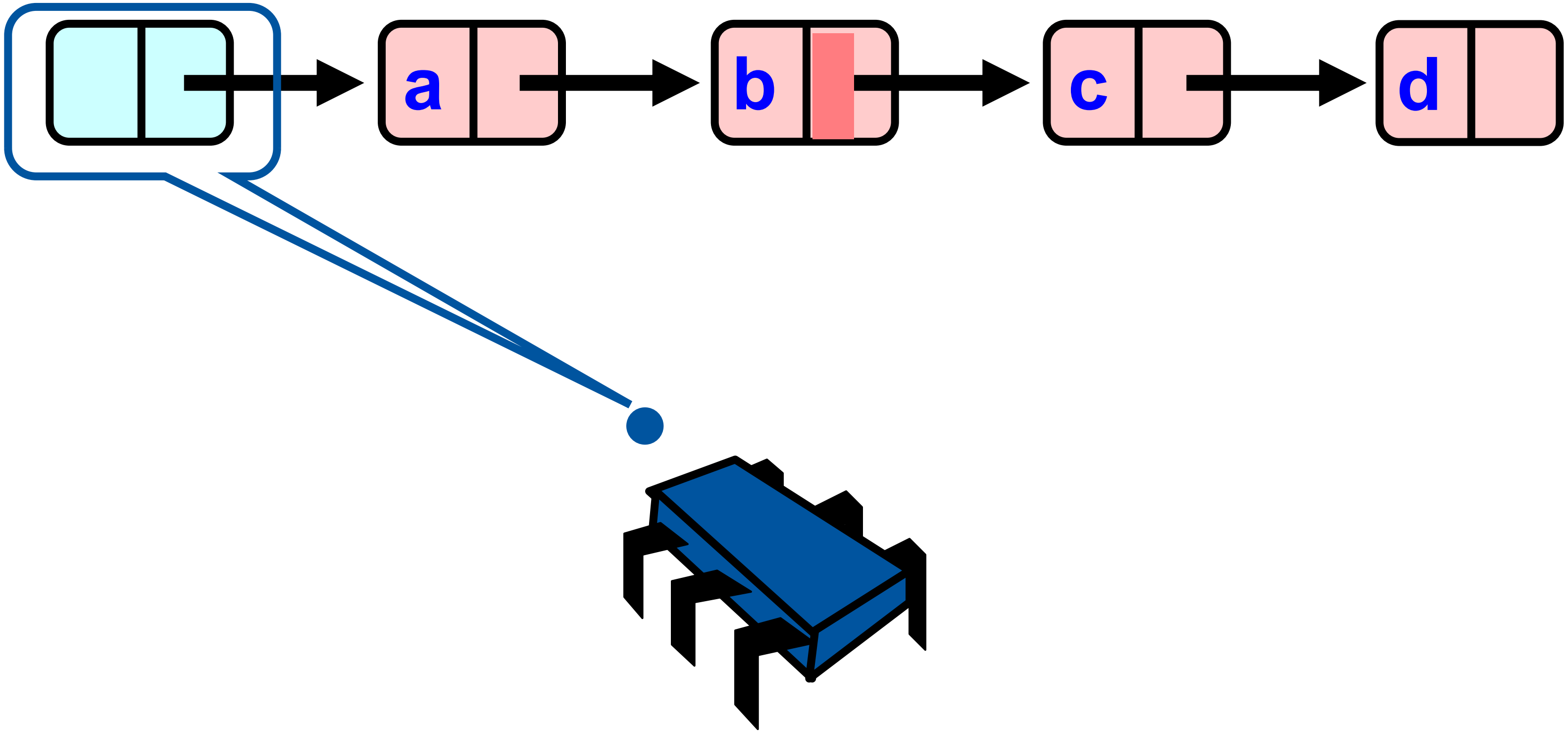


# Illustration: traversal



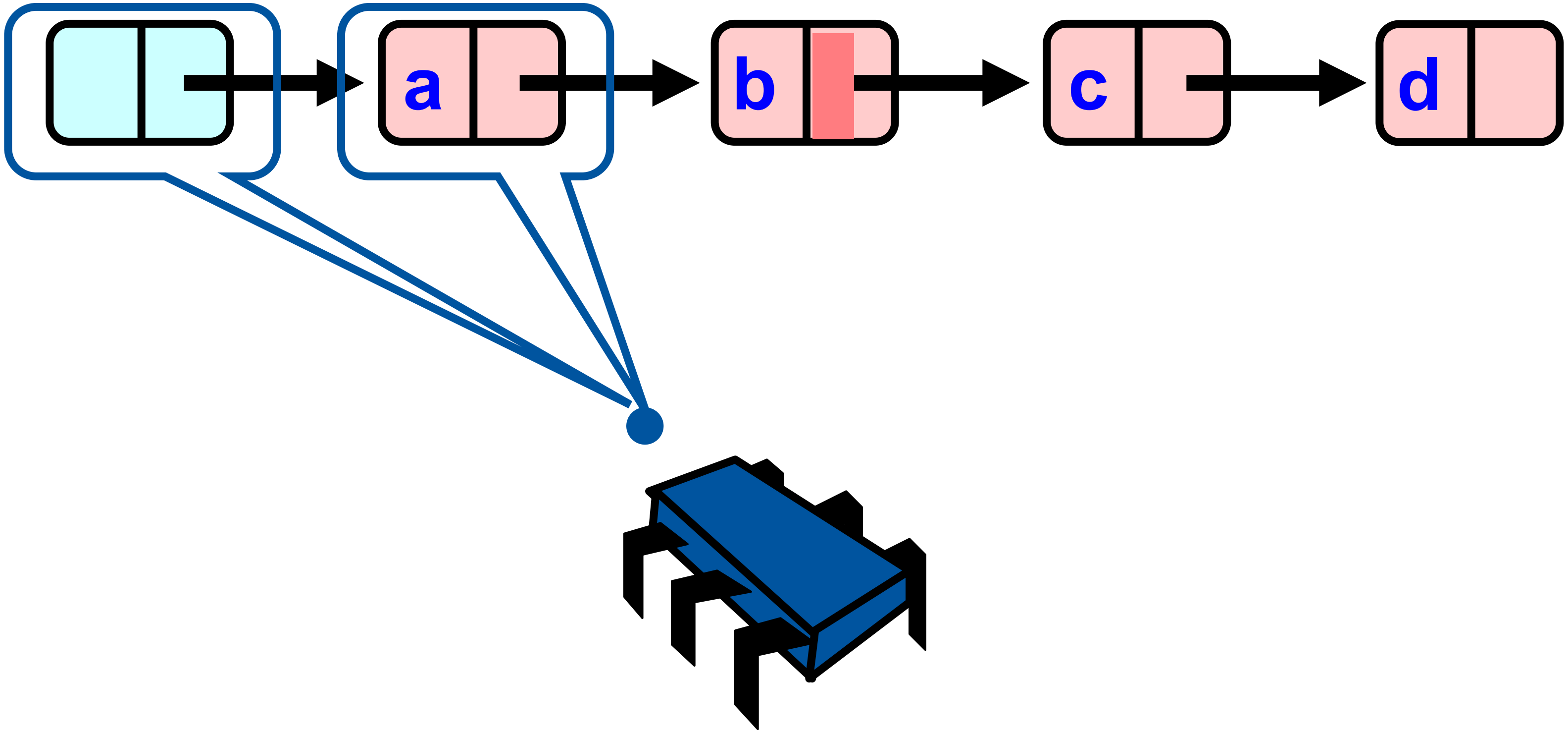
- Reaction to a logically deleted node? CAS the predecessor's next field and proceed (repeat if necessary)

# Illustration: traversal



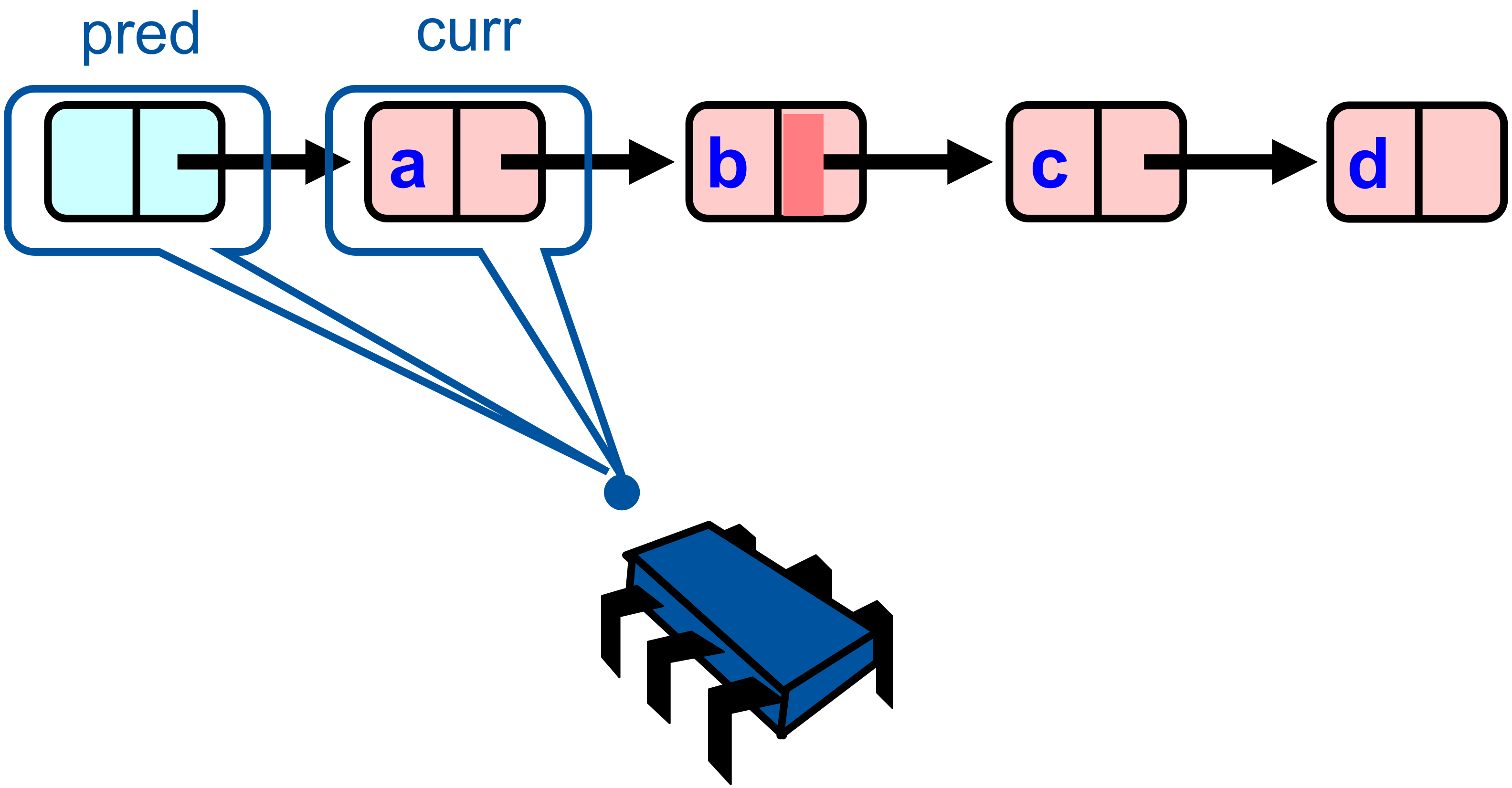
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



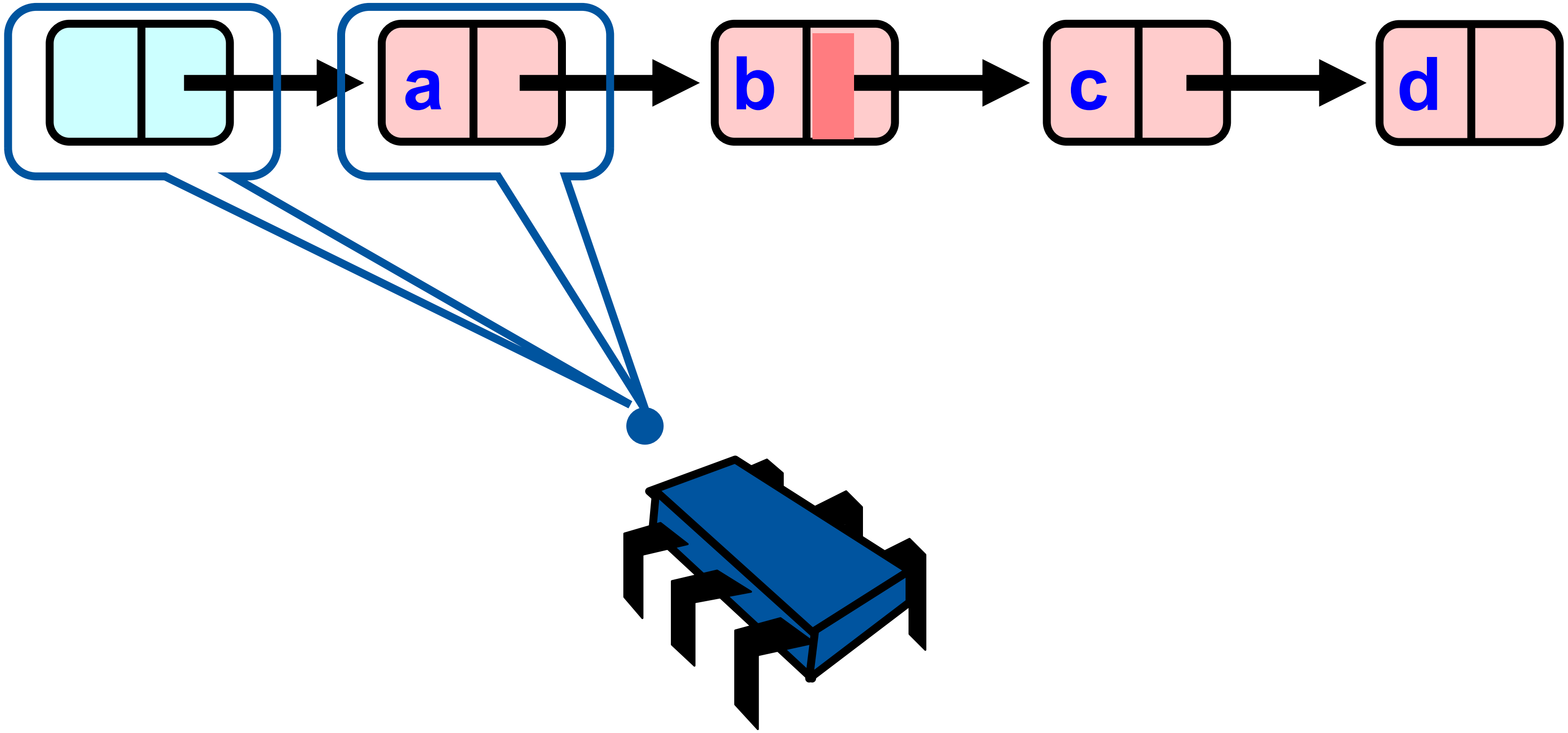
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



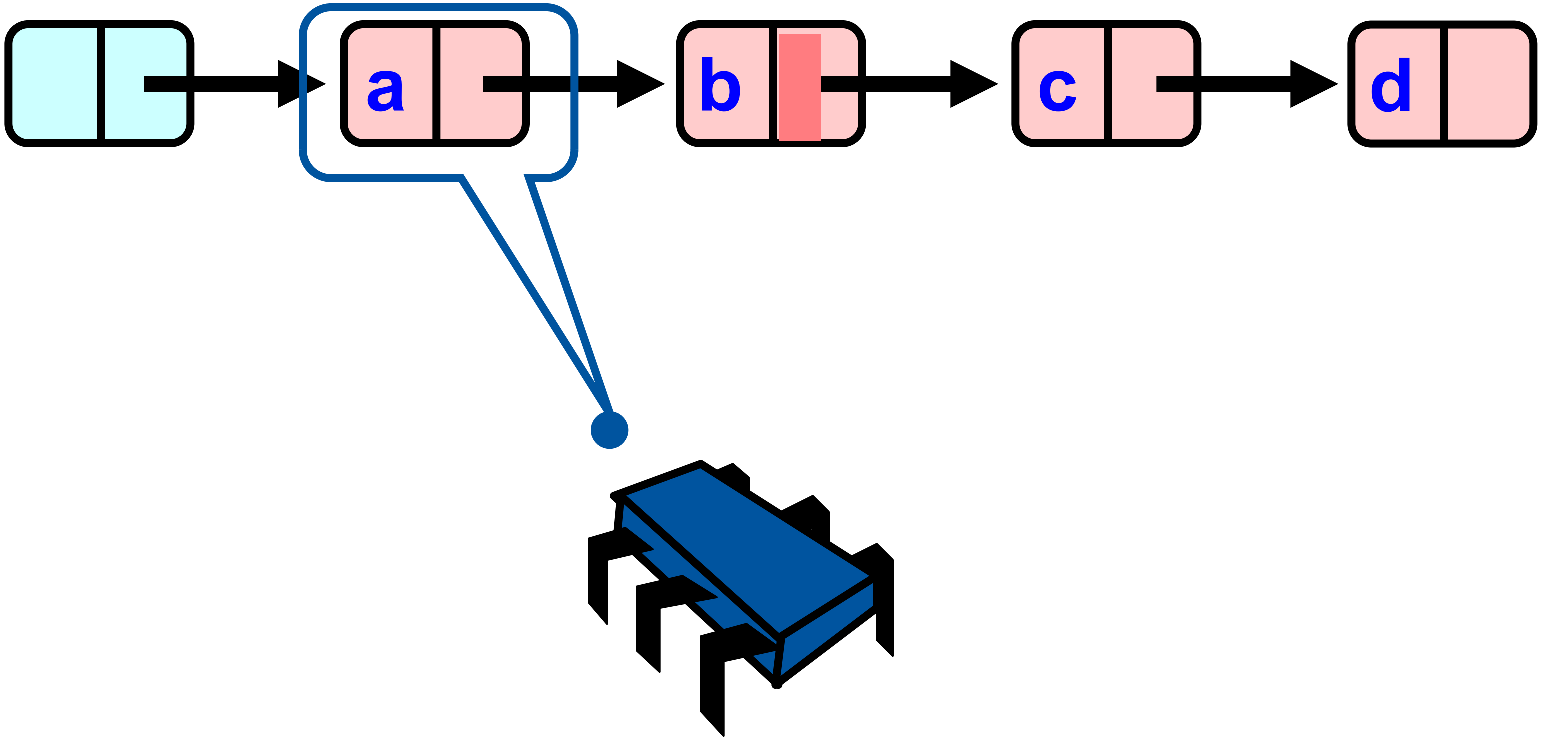
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



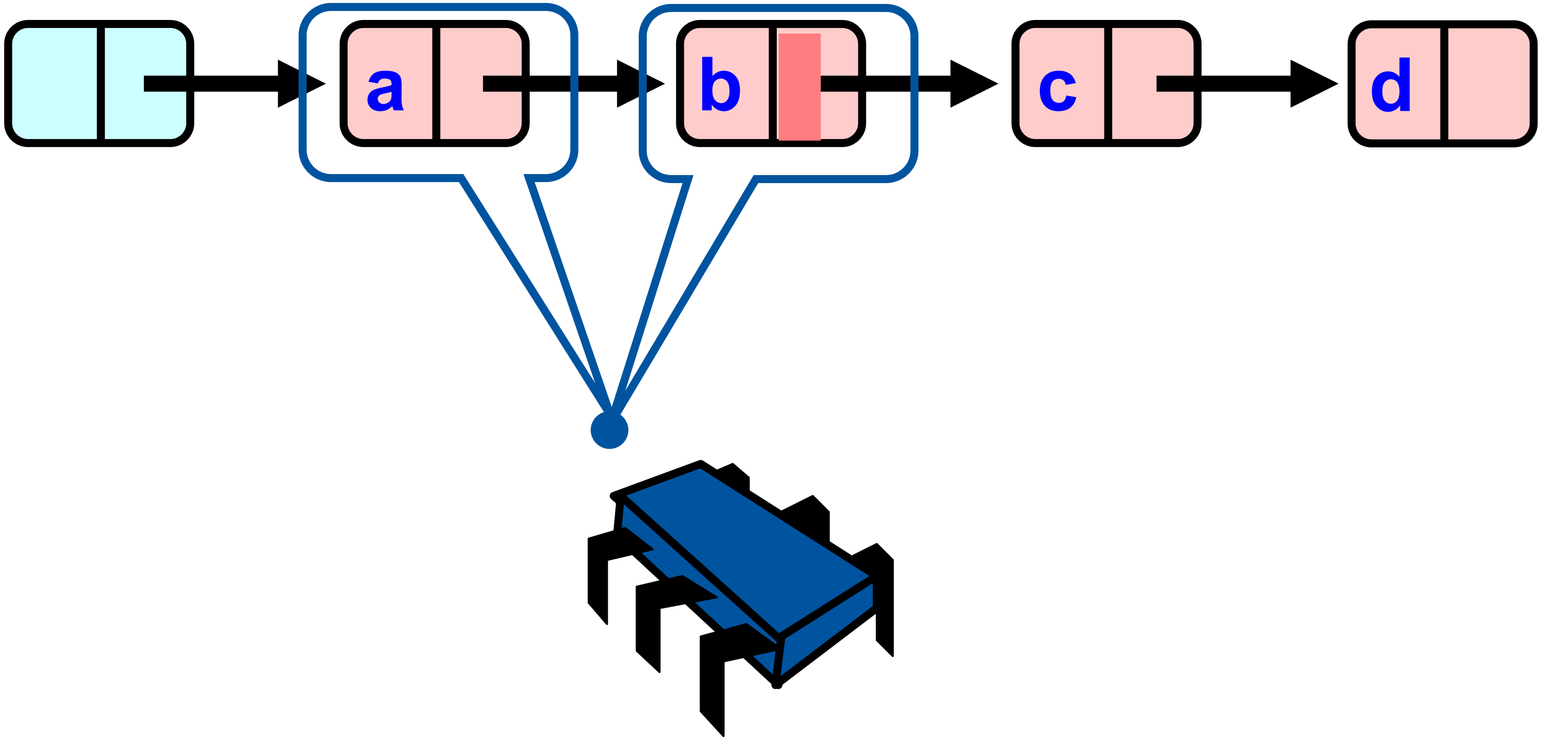
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



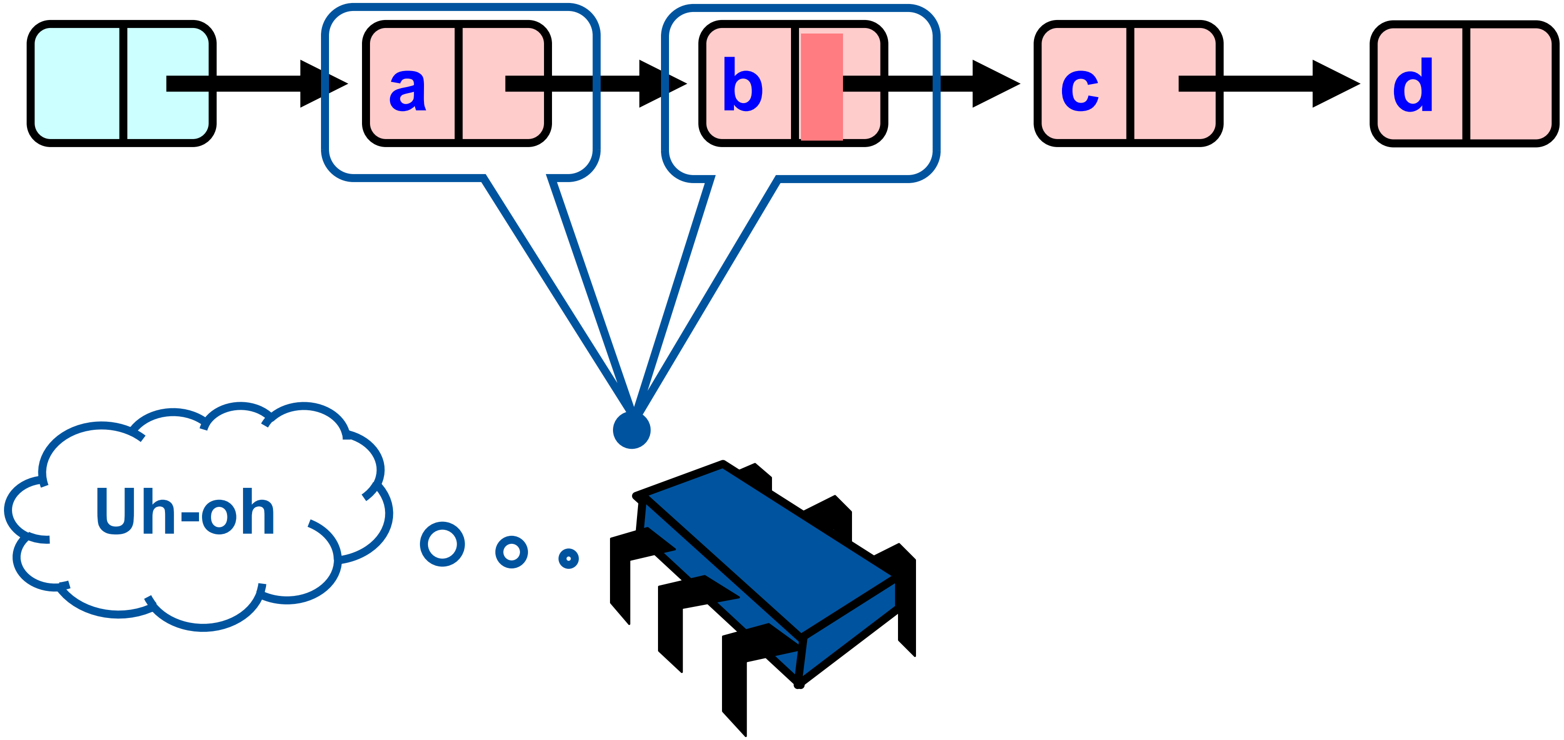
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



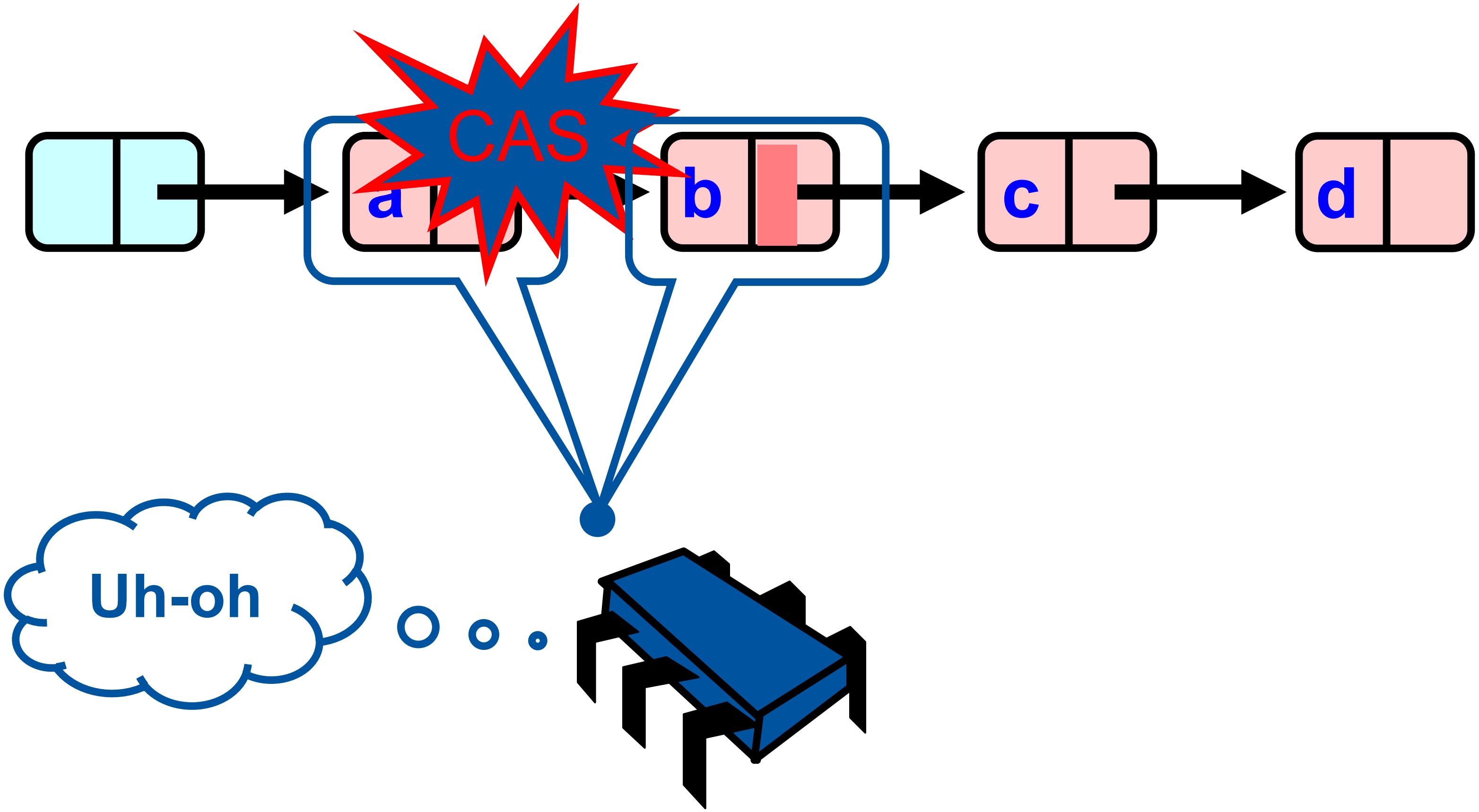
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



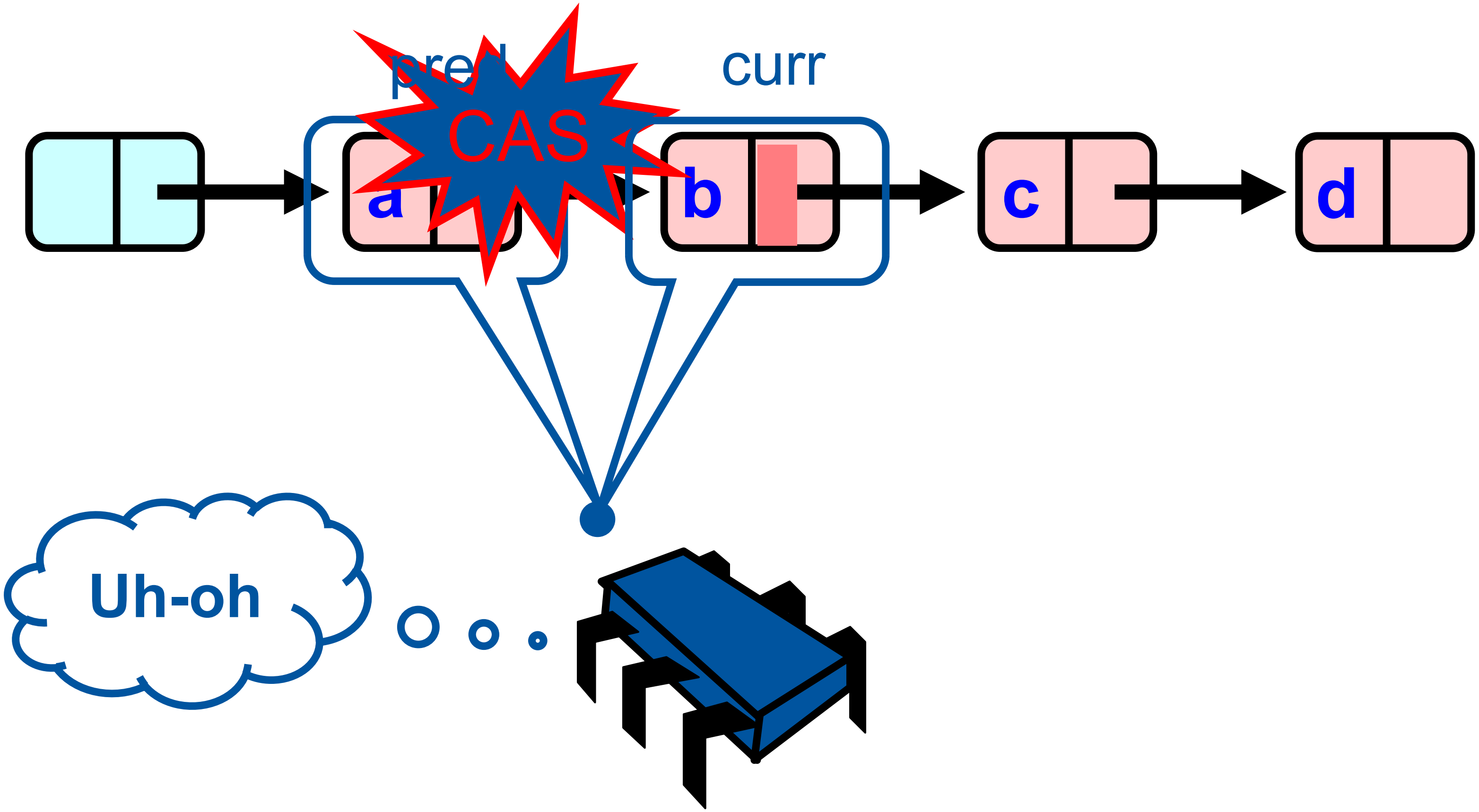
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



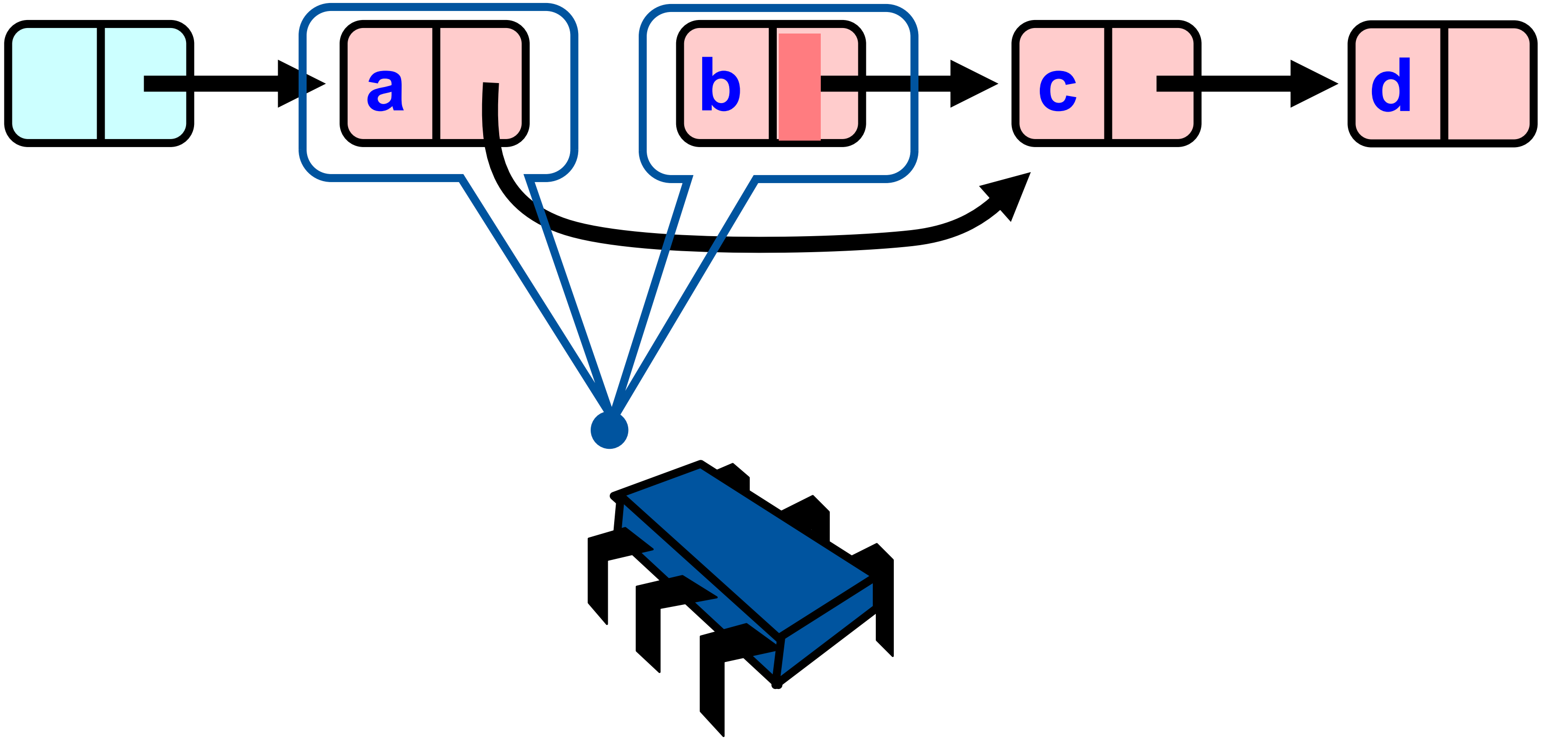
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



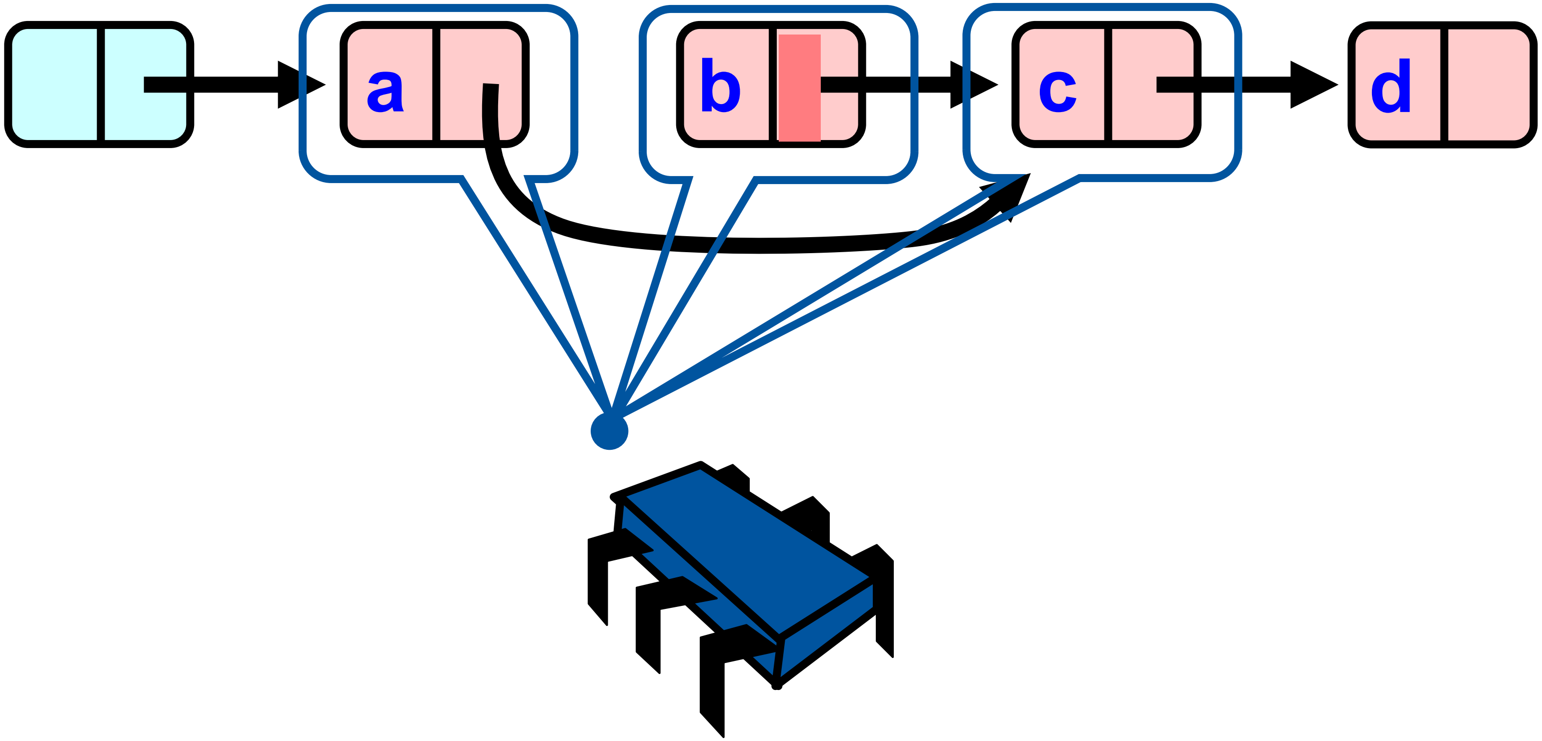
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



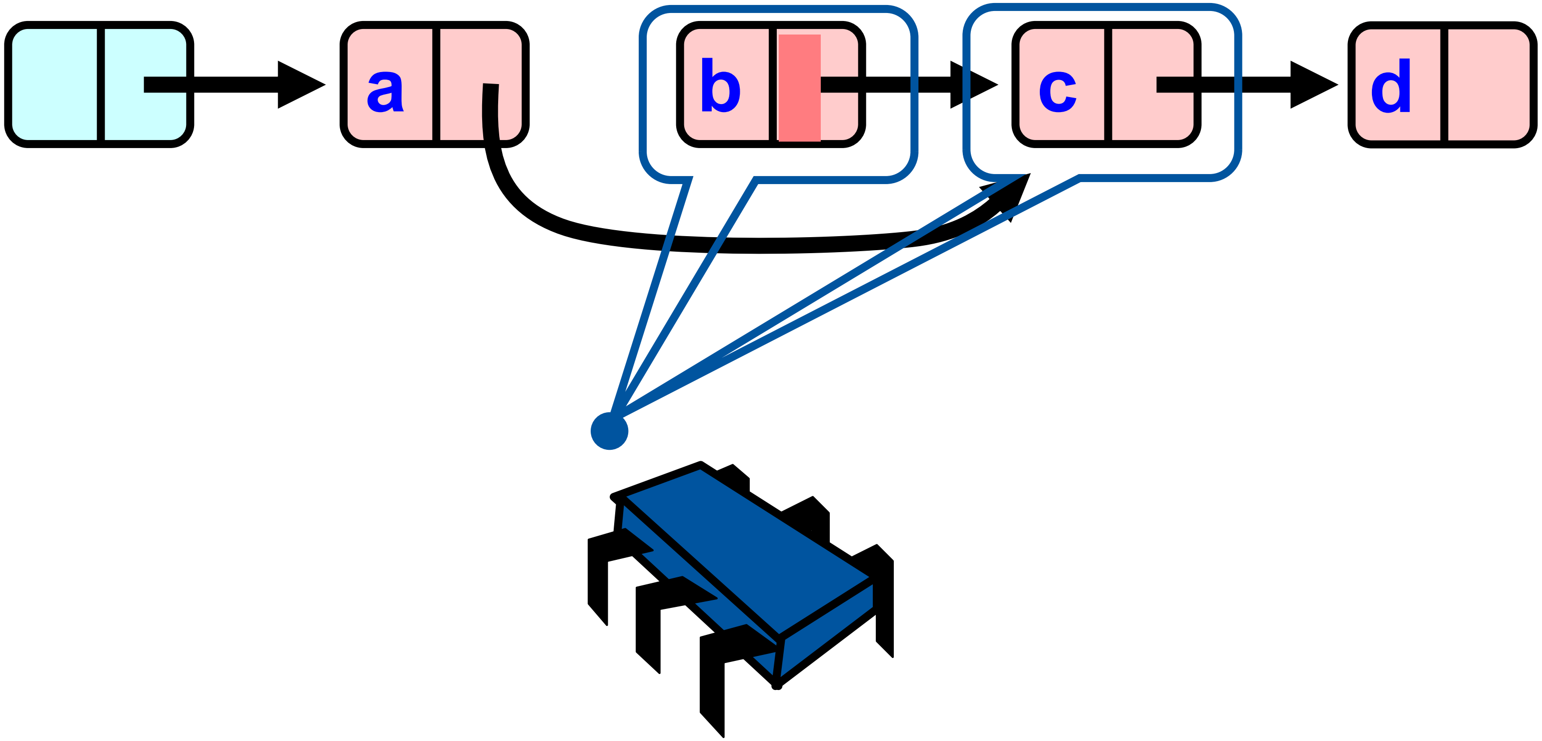
- Reaction to a logically deleted node? CAS the predecessor’s next field and proceed (repeat if necessary)

# Illustration: traversal



- Reaction to a logically deleted node? CAS the predecessor's next field and proceed (repeat if necessary)

# Illustration: traversal



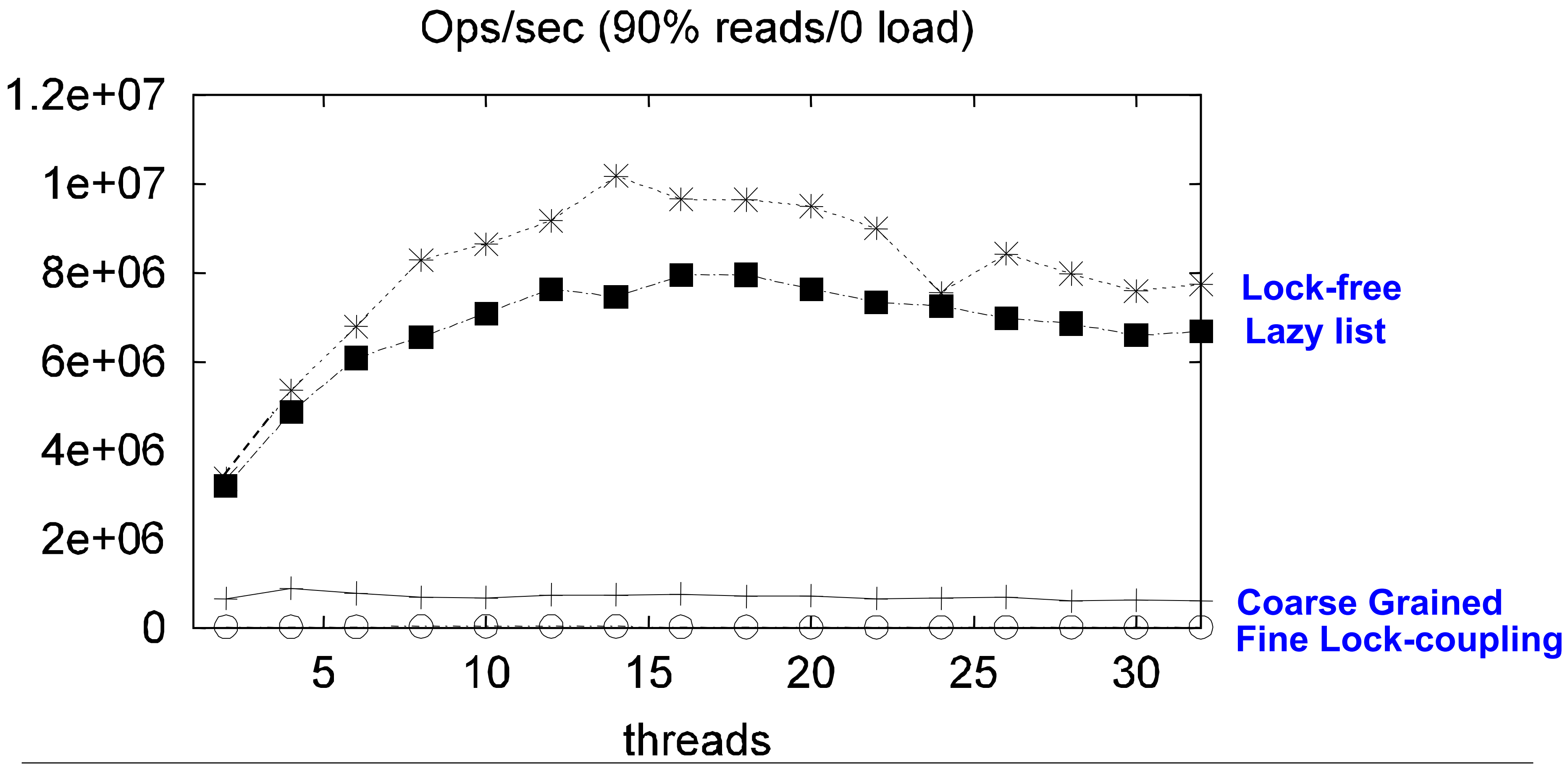
- Reaction to a logically deleted node? CAS the predecessor's next field and proceed (repeat if necessary)

# Performance

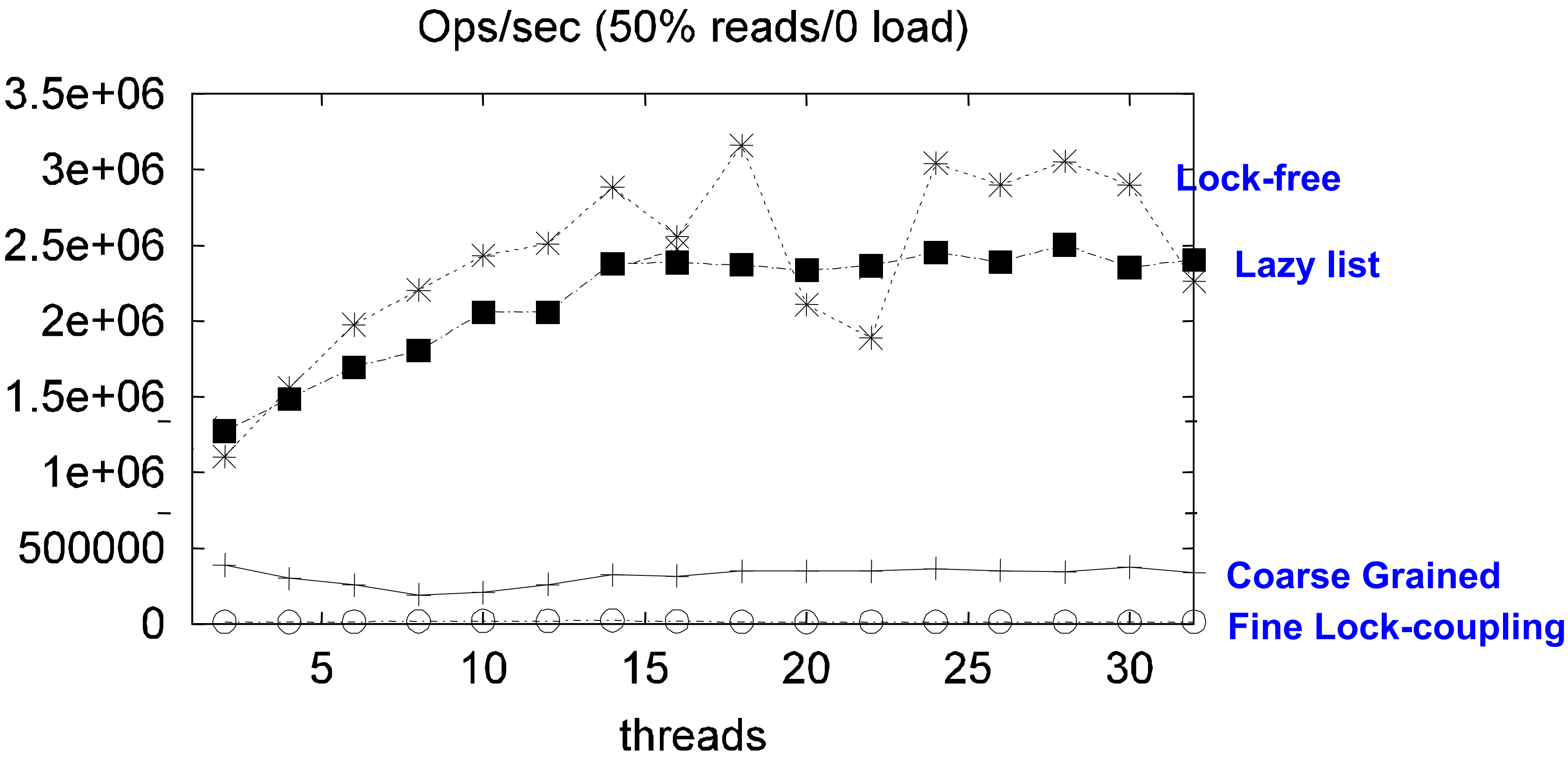
- As presented in the book (Java, The Art of Multiprocessor Prog.):
  - Different list-based set implementations
  - 16-node machine
  - Vary percentage of `contains()` calls

- Thesis offer: Reproduce these measurements on current hardware
  - Inclusion of Transactional Memory
  - Sensitivity to the Memory Hierarchy

# Performance: high contains ratio



# Performance: low contains ratio



# Summary: Lock-free Sync.

- Don’t use locks at all
  - Use `std::atomic<>::compare_exchange_weak()` …

- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex and more implementation work
  - Sometimes high overhead

- Locking vs. Non-blocking:
  - Extremist views on both sides
  - Remember: Blocking/non-blocking is a property of a method