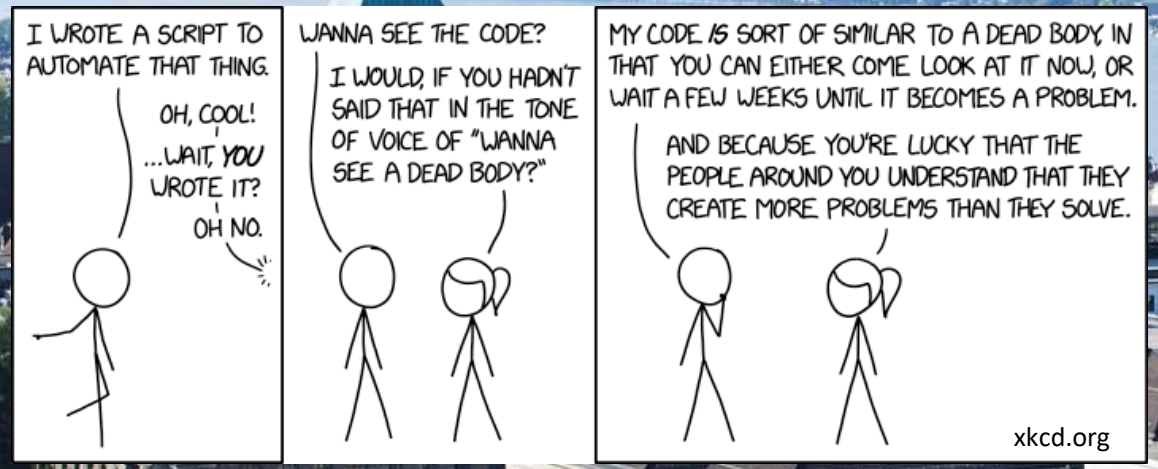
TORSTEN HOEFLER

Parallel Programming

Lock tricks, skip lists, and without Locks I



Neuromorphic Advances Promise Neural Network Performance Jump

By Staff

April 30, 2019

Researchers from Sandia National Laboratories, Stanford University, and UMass Amherst report developing a parallel programming approach for a novel ionic floating-gate memory array that promises to overcome what's been a persistent challenge to improving neuromorphic computing performance on artificial neural networks (ANN).

The new work – which involves breakthroughs in programming and the broader fields of organic electronics and solid-state electrochemistry – was reported in *Science* last week (Parallel programming of an ionic floating-gate memory array for scalable neuromorphic computing).

While there's been plenty of work on the idea that neuromorphic computers will overcome efficiency bottlenecks inherent to conventional computing through parallel programming and read-out of artificial neural network weights in a crossbar memory array, accomplishing that goal has been difficult. The researchers note that the need for "selective and linear weight updates and <10 nanoampere read currents for learning" have restrained efficiency gains compared to conventional digital computing.

Their solution is a new device and programming approach.

Last week

- **Producer/Consumer in detail**
  - Queues, implementation
  - Deadlock cases (repetition)
- **Monitors (repetition)**
  - Condition variables, wait, signal, etc.
  - Java's thread state machine
- **Sleeping barber**
  - Optimize (avoid) notifications using counters
- **RW Locks**
  - Fairness is an issue (application-dependent)
- **Lock tricks on the list-based set example**
  - Fine-grained locking … continued now

Learning goals today

- **Finish lock tricks**
  - Optimistic synchronization
  - Lazy synchronization
- **Conflict-minimizing structures (probabilistic)**
  - Example: skip lists
- **Lock scheduling**
  - Sleeping vs. spinlock (deeper repetition)
- **Lock-free**
  - Stack, list

Fine grained Locking

Often more intricate than visible at a first sight

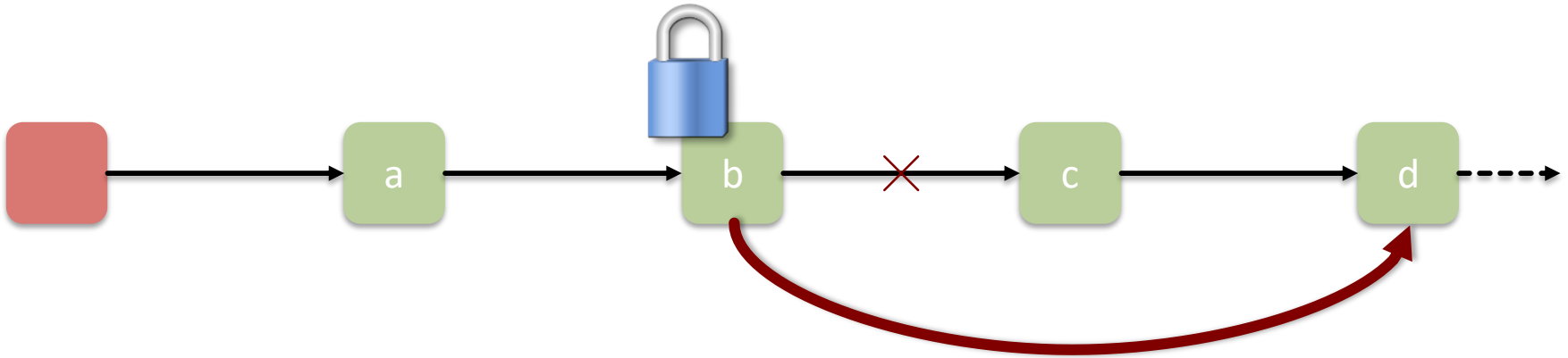
- requires careful consideration of special cases

Idea: split object into pieces with separate locks

- no mutual exclusion for algorithms on disjoint pieces

Let's try this

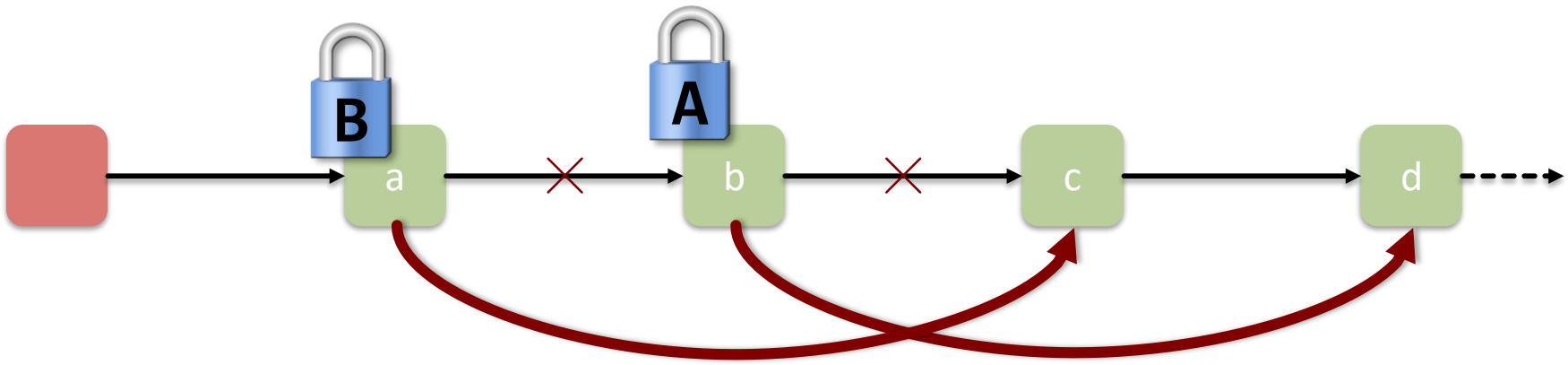
`remove(c)`



`Is this ok?`

Let's try this

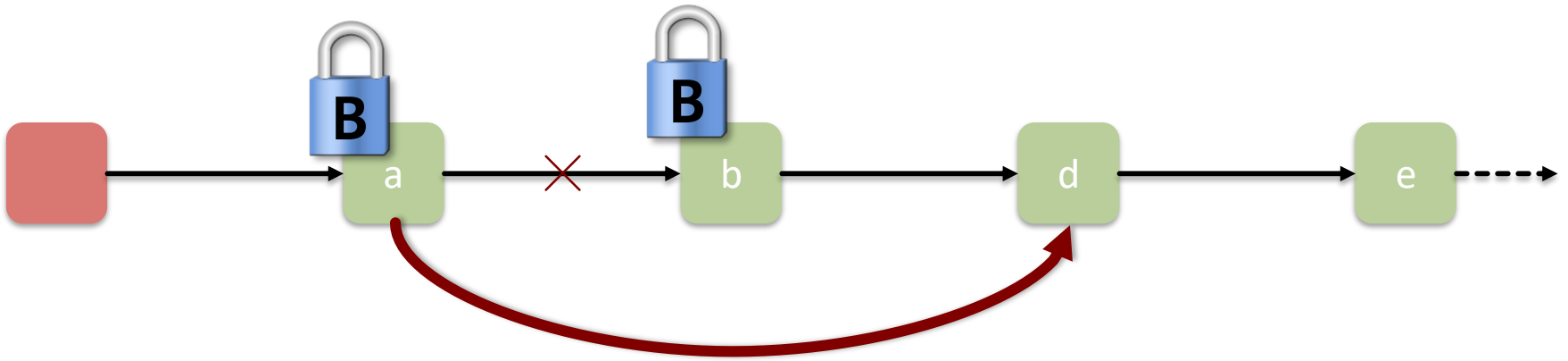
```
Thread A: remove(c)
Thread B: remove(b)
```



c not deleted! ☹

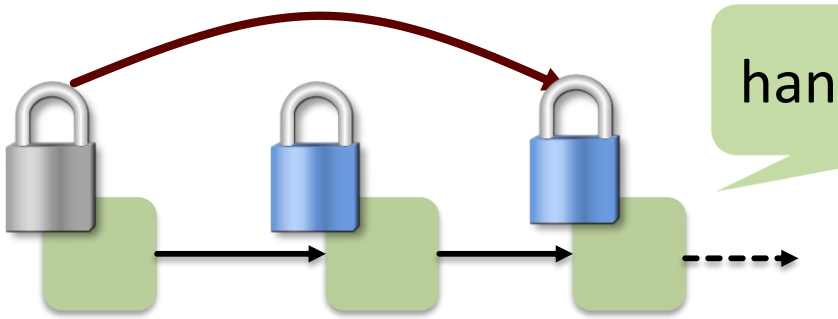
What's the problem?

- When deleting, the next field of next is read, i.e., next also has to be protected.
- A thread needs to lock both, predecessor and the node to be deleted (hand-over-hand locking).



Remove method

```
public boolean remove(T item) {
  Node pred = null, curr = null;
  int key = item.hashCode();
  head.lock();
  try {
   pred = head;
   curr = pred.next;
   curr.lock();
   try {
      // find and remove
   } finally { curr.unlock(); }
  } finally { pred.unlock(); }
}
```



hand over hand

```
while (curr.key < key) {
    pred.unlock();
    pred = curr; // pred still locked
    curr = curr.next;
    curr.lock(); // lock hand over hand
}
if (curr.key == key) {
    pred.next = curr.next; // delete
    return true;
}
return false;
```

remark: sentinel at front and end of list prevents an exception here

Disadvantages?

- Potentially long sequence of acquire / release before the intended action can take place
- One (slow) thread locking "early nodes" can block another thread wanting to acquire "late nodes"

OPTIMISTIC SYNCHRONIZATION

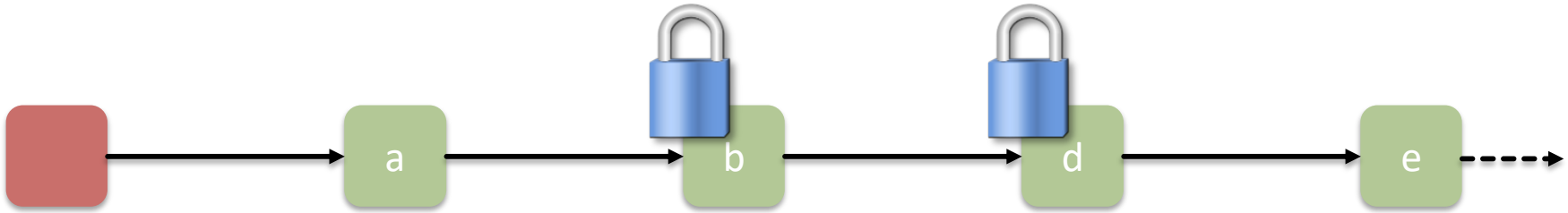
Idea

Find nodes without locking,

- **then lock nodes and**
- **check that everything is ok (validation)**

What do we need to “validate”?

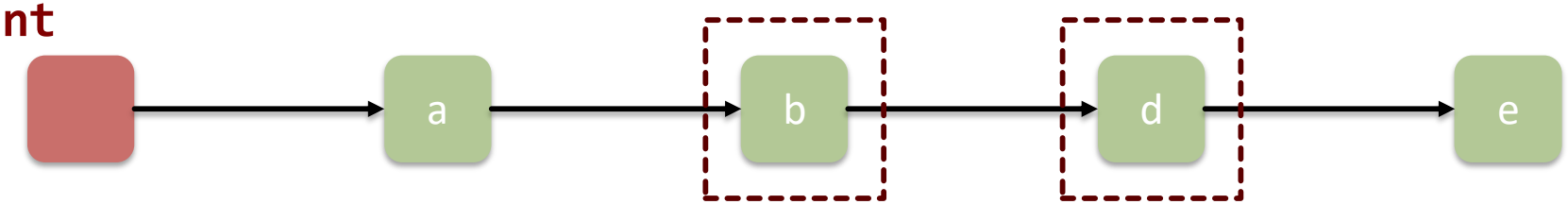
e.g., add(c)



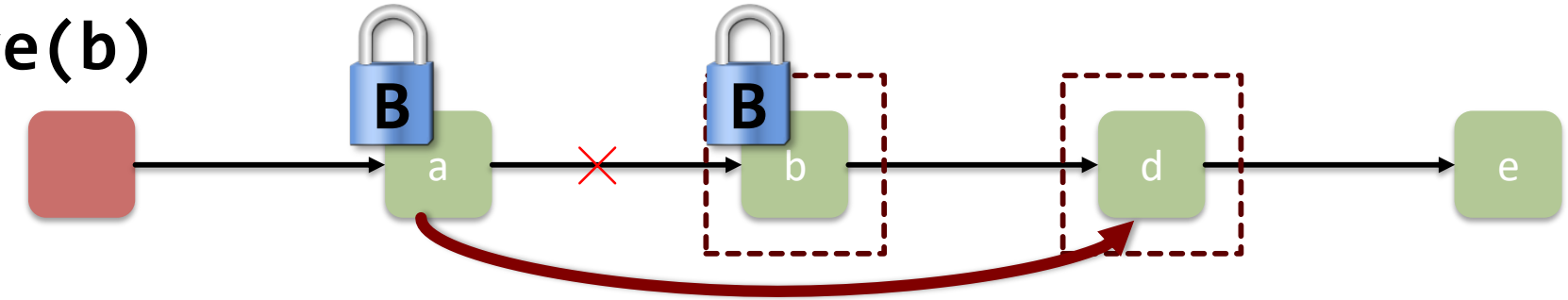
Validation: what could go wrong?

Thread A: add(c)

A: find insertion point



Thread B: remove(b)

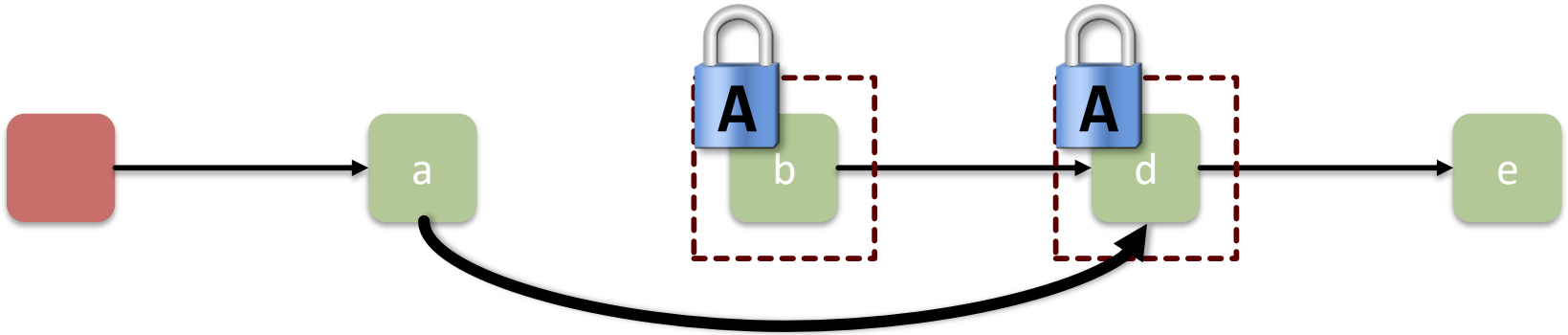


A: lock

A: validate: rescan

A: b not reachable

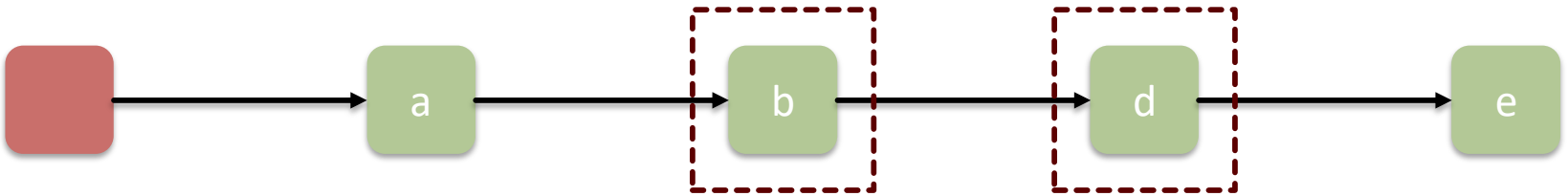
→return false



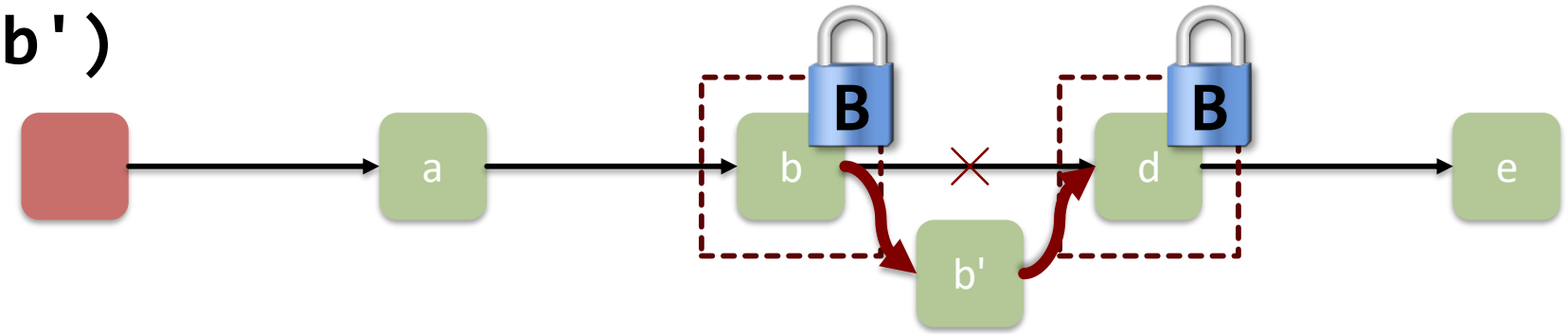
Validation: what else could go wrong?

Thread A: add(c)

A: find insertion point



Thread B: insert(b')

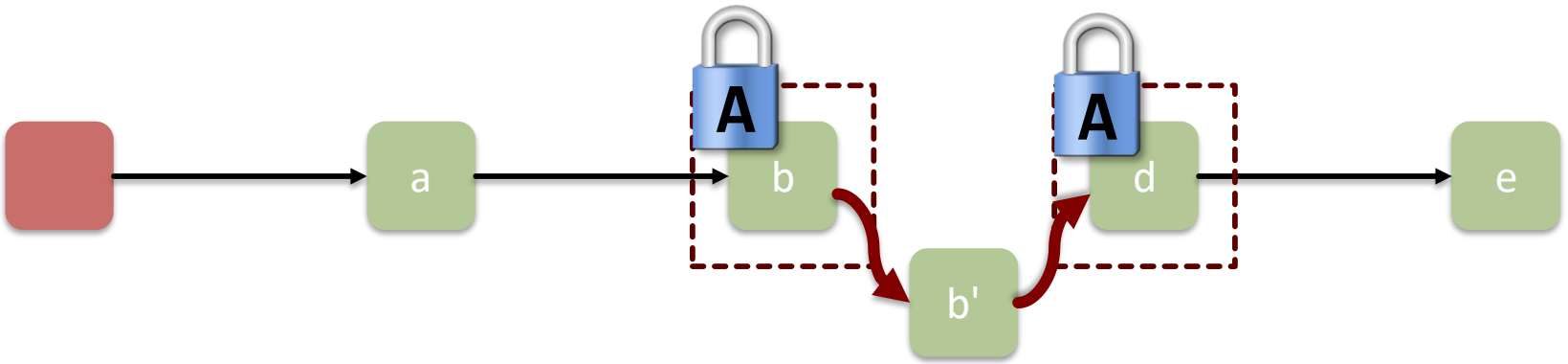


A: lock

A: validate: rescan

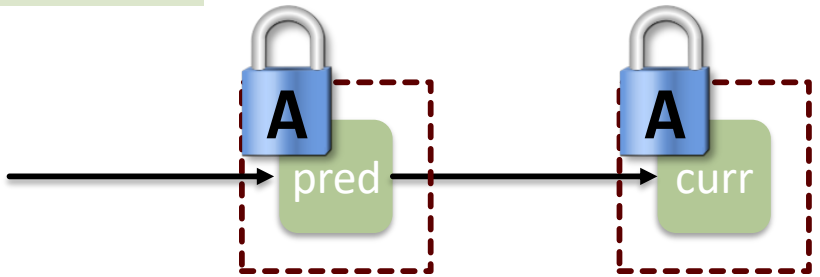
A: d != succ(b)

→return false



Validate - summary

```
private Boolean validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) { // reachable?
       if (node == pred)
             return pred.next == curr; // connected?
       node = node.next;
   }
   return false;
}
```



Correctness (remove c)

If

- nodes b and c both locked
- node b still reachable from head
- node c still successor to b

then

- neither is in the process of being deleted

➔ ok to delete and return true

Correctness (remove c)

If

- nodes b and d both locked
- node b still reachable from head
- node d still successor to b

then

- neither is in the process of being deleted,
  therefore a new element c must appear between b and d
- no thread can add between b and d:
  c cannot have appeared after our locking

➔ ok to return false

Optimistic List

Good:

- No contention on traversals.
- Traversals are wait-free.
- Less lock acquisitions.

Wait-Free:
Every call finishes in a finite number of steps (NEVER waits for other threads).

Bad:

- Need to traverse list twice
- The contains() method needs to acquire locks
- Not starvation-free

Is the optimistic list starvation-free? Why/why not?

LAZY SYNCHRONISATION

Laziness

The quality that makes you go to great effort to reduce overall energy expenditure [...] **the first great virtue of a programmer.**

Larry Wall, Programming Perl
(emphasis mine)

Lazy List

Like optimistic list but

- Scan only once
- Contains() never locks

How?

- Removing nodes causes trouble
- Use deleted-markers → invariant: every unmarked node is reachable!
- Remove nodes «lazily» after marking

Lazy List: Remove

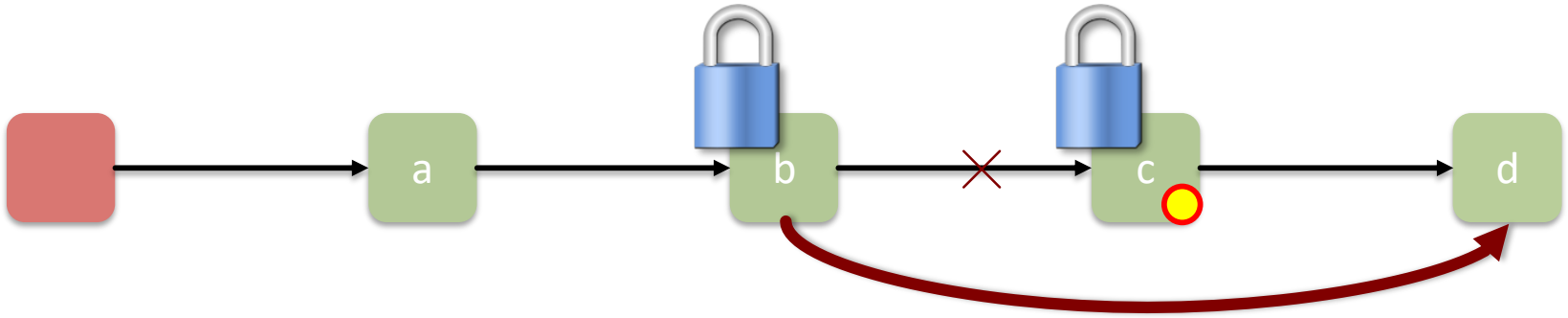
Scan list (as before)

Lock predecessor and current (as before)

Logical delete: mark current node as removed

Physical delete: redirect predecessor's next

e.g., remove(c)



Key invariant

If a node is not marked then

- It is reachable from head
- And reachable from its predecessor

A: remove(c)

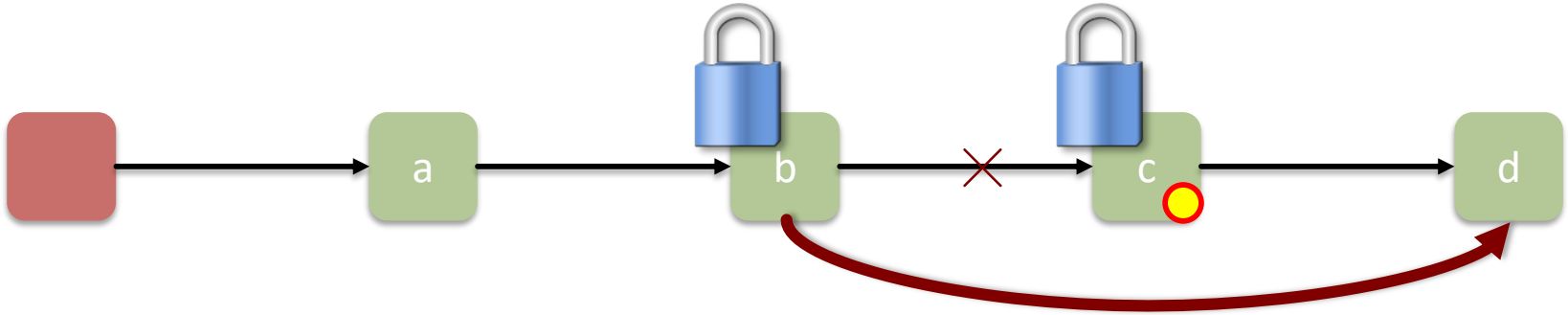
lock

check if b or c are marked

not marked? ok to delete:

mark c

delete c



Remove method

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) { // optimistic, retry
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        // remove or not
      } finally { curr.unlock(); }
    } finally { pred.unlock(); }
  }
}
```

```
if (!pred.marked && !curr.marked &&
      pred.next == curr) {
   if (curr.key != key)
      return false;
   else {
      curr.marked = true;    // logically remove
      pred.next = curr.next; // physically remove
      return true;
   }
}
```

Wait-Free Contains

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

This set data structure is again for demonstration only. Do not use this to implement a list! Now on to something more practical.

Bill Pugh received a Ph.D. in Computer Science (with a minor in Acting) from Cornell University. He was a professor at the University of Maryland for 23.5 years, and in January 2012 became professor emeritus to start new adventure somewhere at the crossroads of software development and entrepreneurship.

Bill Pugh is a Packard Fellow, and invented Skip Lists, a randomized data structure that is widely taught in undergraduate data structure courses. He has also made research contributions in in techniques for analyzing and transforming scientific codes for execution on supercomputers, and in a number of issues related to the Java programming language, including the development of JSR 133 - Java Memory Model and Thread Specification Revision. Prof. Pugh's current research focus is on developing tools to improve software productivity, reliability and education. Current research projects include FindBugs, a static analysis tool for Java, and Marmoset, an innovative framework for improving the learning and feedback cycle for student programming projects.

Prof. Pugh has spoken at numerous developer conferences, including JavaOne, Goto/Jaoo in Aarhus, the Devoxx conference in Antwerp, and CodeMash. At JavaOne, he received six JavaOne RockStar awards, given to the speakers that receive the highest evaluations from attendees.

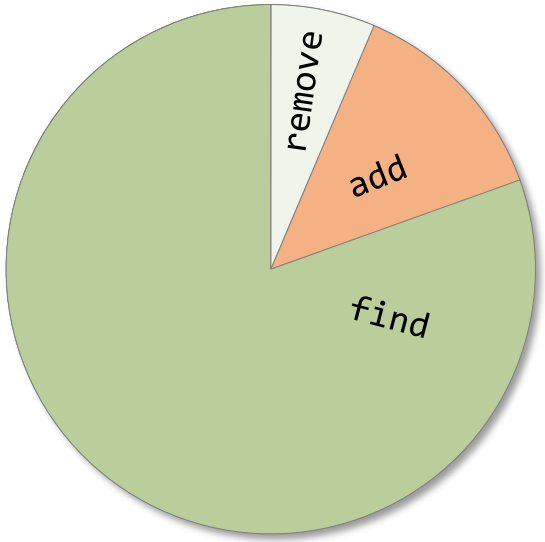
Professor Pugh spent the 2008-2009 school year on sabbatical at Google, where, among other activities, he learned how to eat fire.

More practical:
Lazy Skip Lists

Bill Pugh

Skip list – a practical representation for sets!

- **Collection of elements (without duplicates)**
- **Interface:**
  - `add` `// add an element`
  - `remove` `// remove an element`
  - `find` `// search an element`
- **Assumptions:**
  - Many calls to find()
  - Fewer calls to add() and much fewer calls to remove()



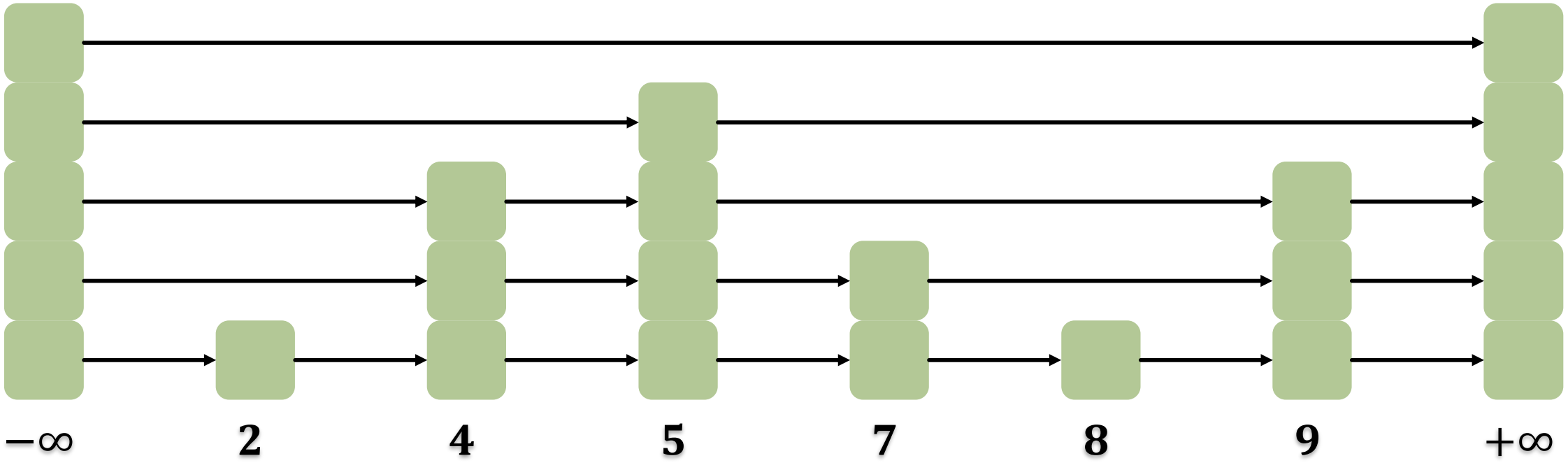
How about balanced trees?

- **AVL trees, red-black trees, treaps, ...**
  - rebalancing after `add` and `remove` expensive
  - rebalancing is a *global* operation (potentially changing the whole tree)
  - particularly hard to implement in a lock-free way.

- **→ Skip lists solve challenges probabilistically (Las Vegas style)**

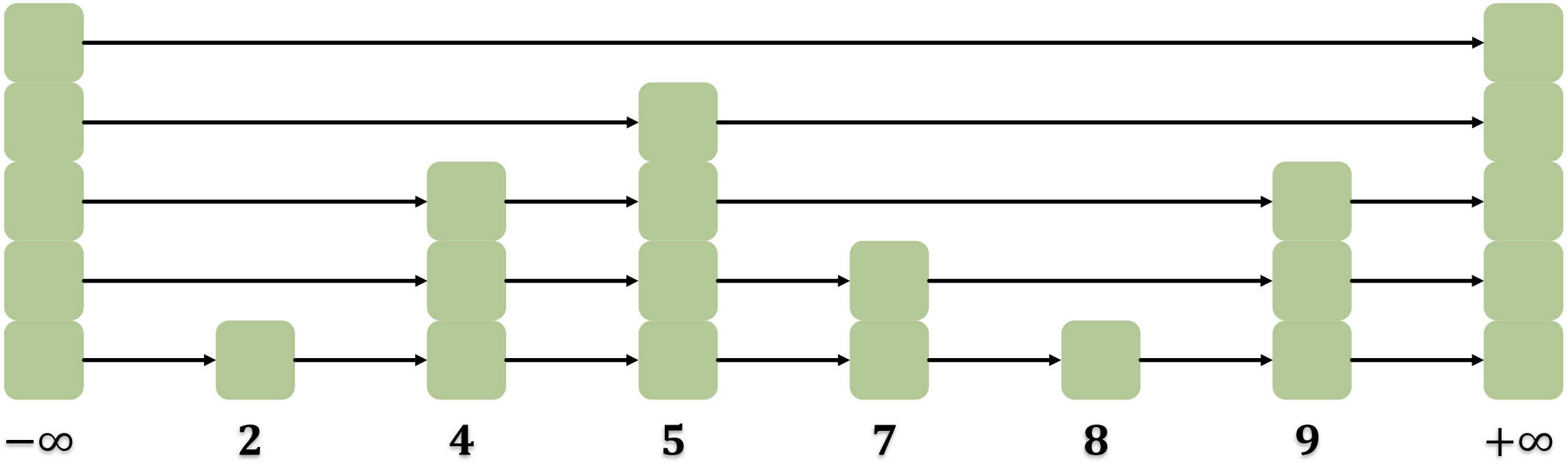
Skip lists

- **Sorted multi-level list**
- **Node height probabilistic, e.g., $\mathbb{P}(height = n) = 0.5^n$, no rebalancing**



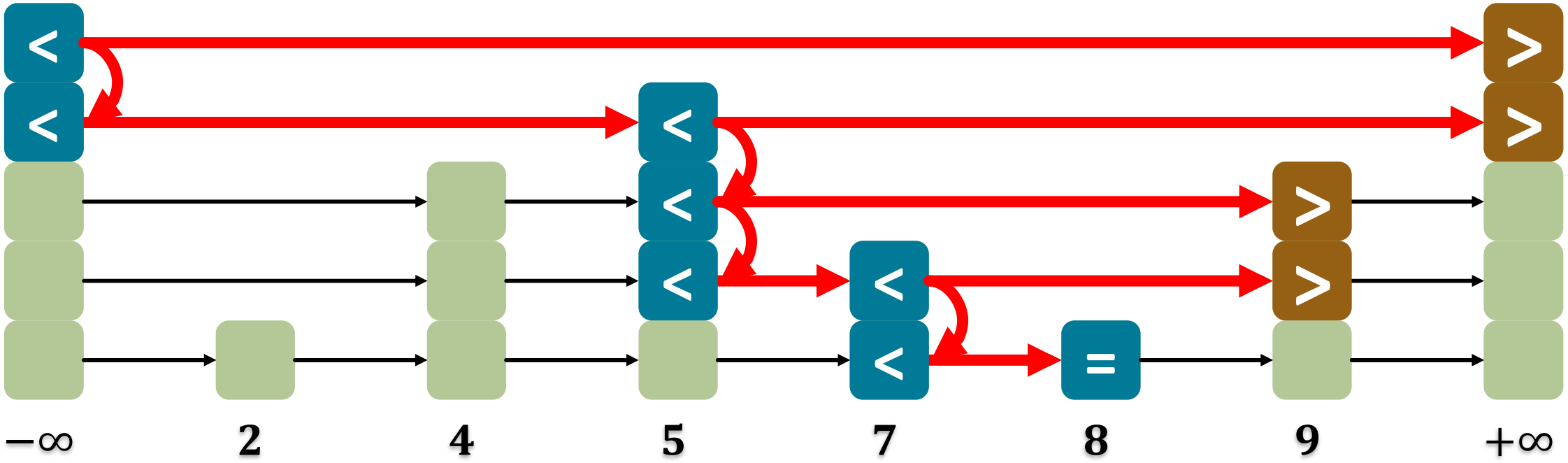
Skip list property

- **Sublist relationship between levels: higher level lists are always contained in lower-level lists. Lowest level is entire list.**



Searching

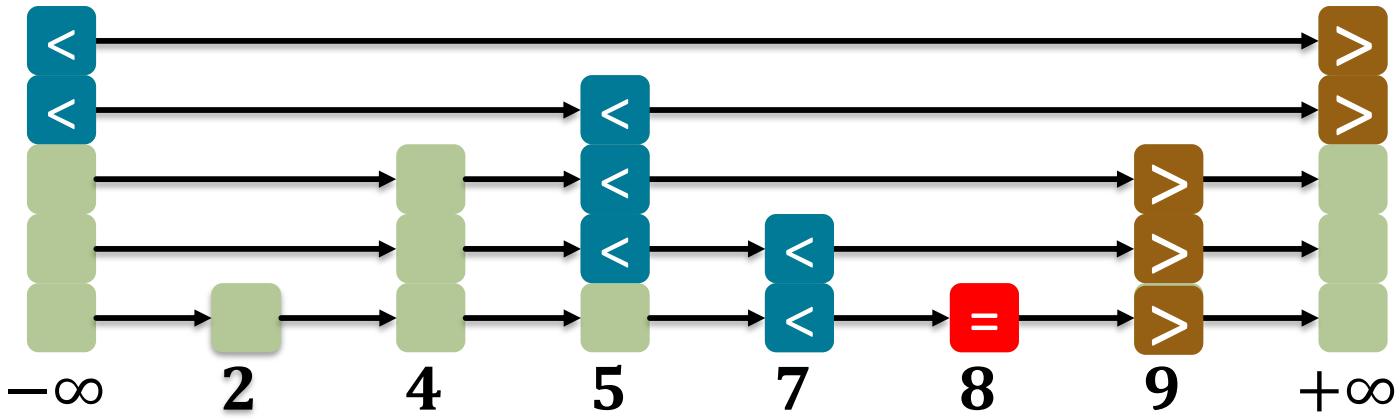
- Logarithmic Search (with high probability)
- Example: Search for 8



Sequential find

- `// find node with value x`
- `// return -1 if not found, node level, succ, and pre otherwise`
- `// pre = array of predecessor node for all levels`
- `// succ = array of successor node for all levels`
- `int find(T x, Node<T>[] pre, Node<T>[] succ)`

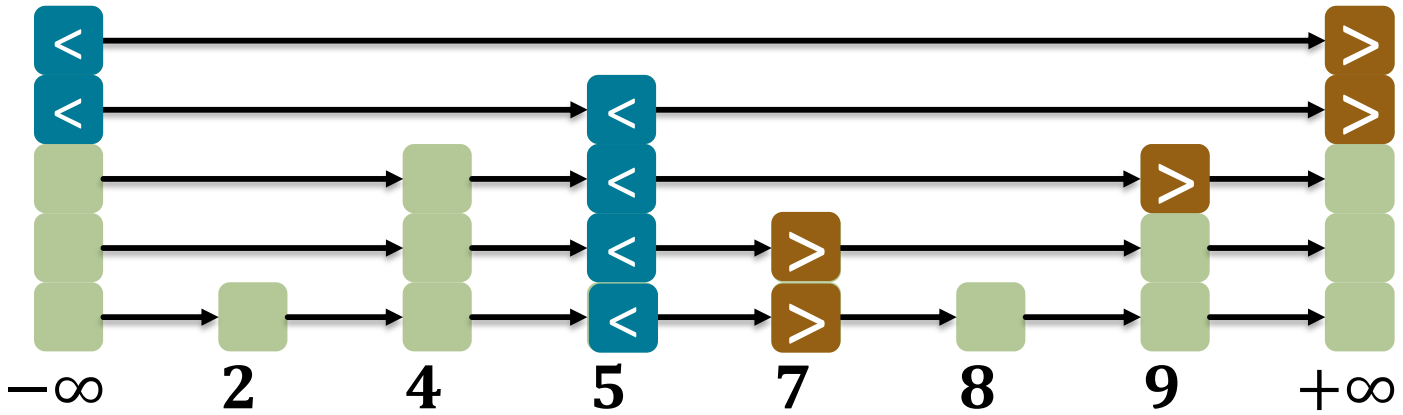
- e.g., x = 8
- returns 0



Sequential find

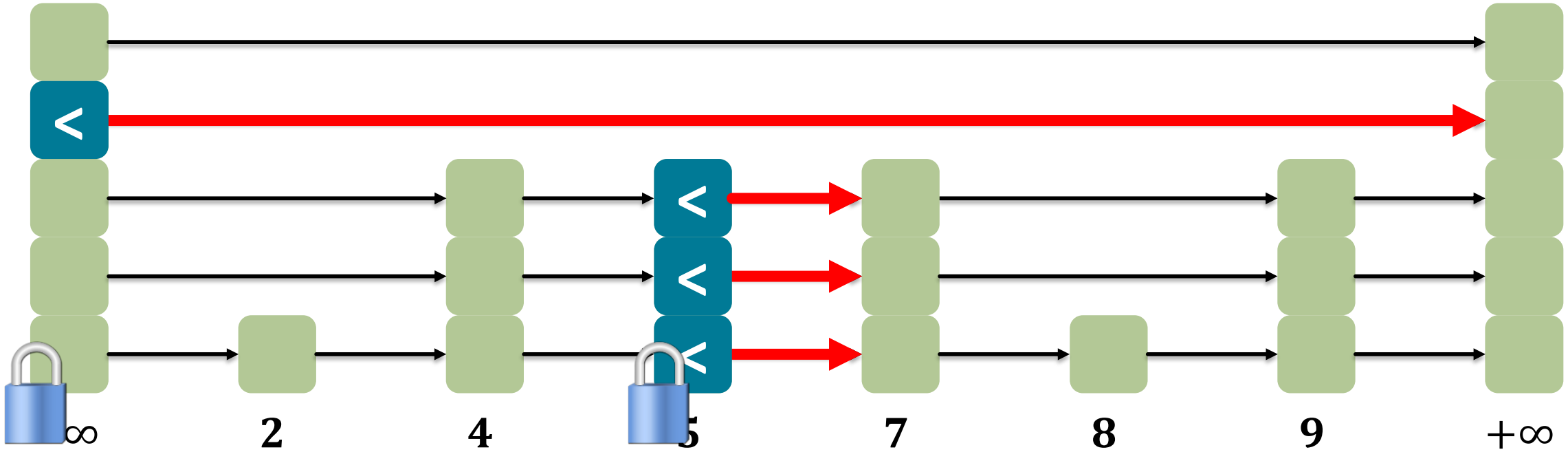
- `// find node with value x`
- `// return -1 if not found, node level, succ, and pre otherwise`
- `// pre = array of predecessor node for all levels`
- `// succ = array of successor node for all levels`
- `int find(T x, Node<T>[] pre, Node<T>[] succ)`

- e.g., x = 6
- returns -1



add (6) – with four levels!

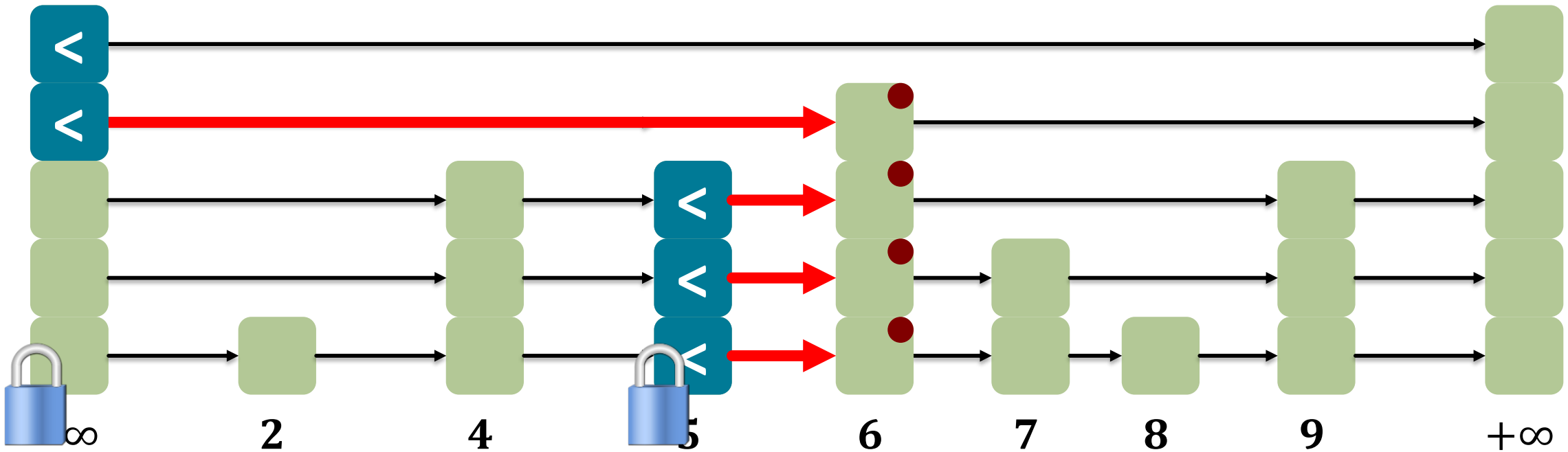
- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)



add (6)

- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)

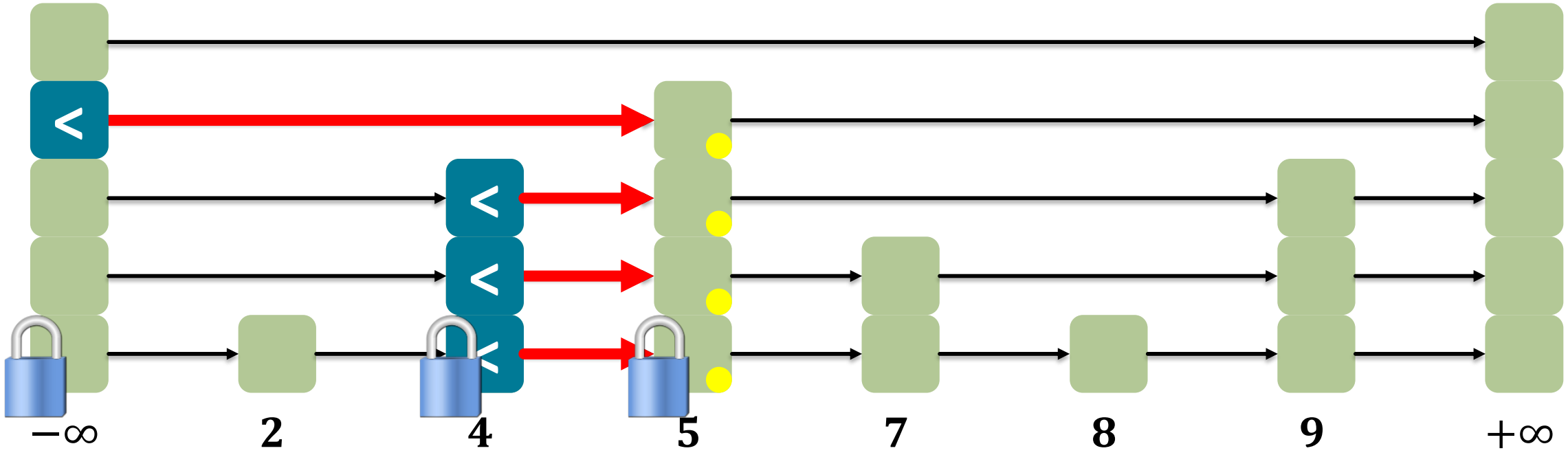
- Splice
- mark fully linked
- Unlock



remove(5)

- find predecessors
- lock victim
- logically remove victim (mark)

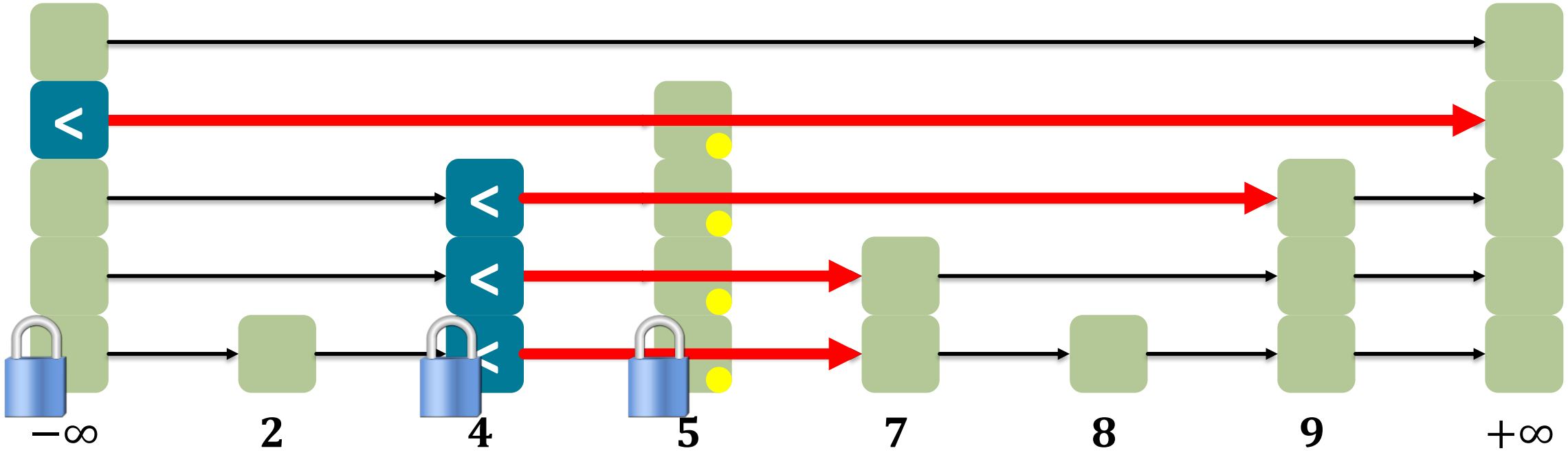
- Lock predecessors and validate



remove(5)

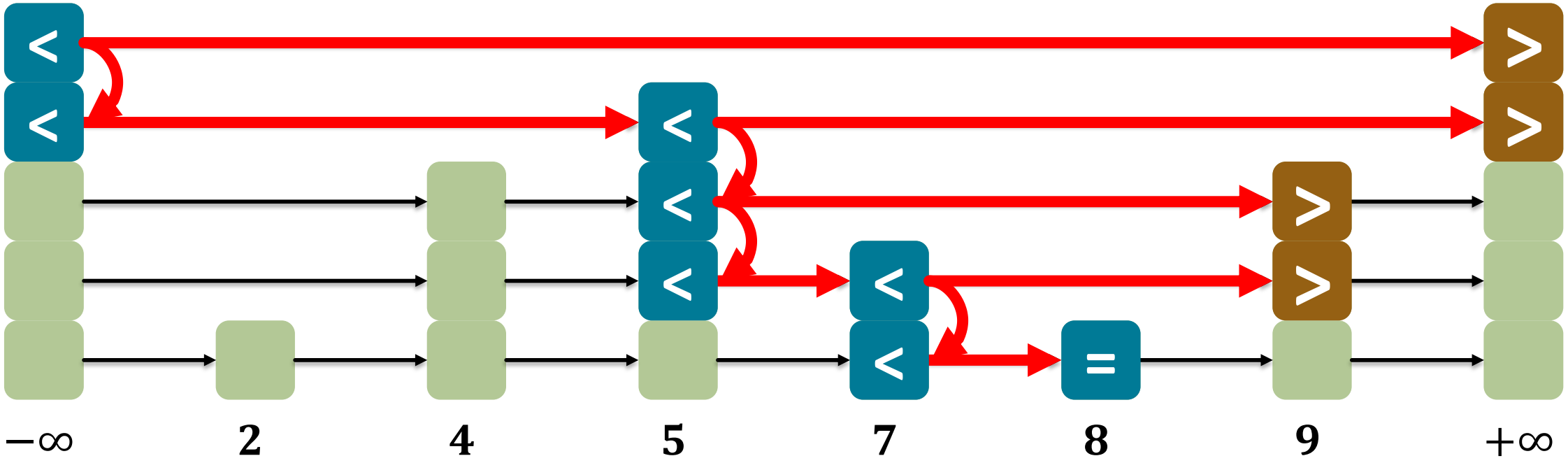
- find predecessors
- lock victim
- logically remove victim (mark)

- Lock predecessors and validate
- physically remove
- unlock



contains(8)

- sequential find() & not logically removed & fully linked
- even if other nodes are removed, it stays reachable
- contains is wait-free (while add and remove are not)



Skip list

- **Practical parallel datastructure**
- **Code in book (latest revision!) – 139 lines**
  - Too much to discuss in detail here
- **Review and implement as exercise**

Now back to locks to motivate lock-free

- Spinlocks vs Scheduled Locks
- Lock-free programming
- Lock-free data structures: stack and list set

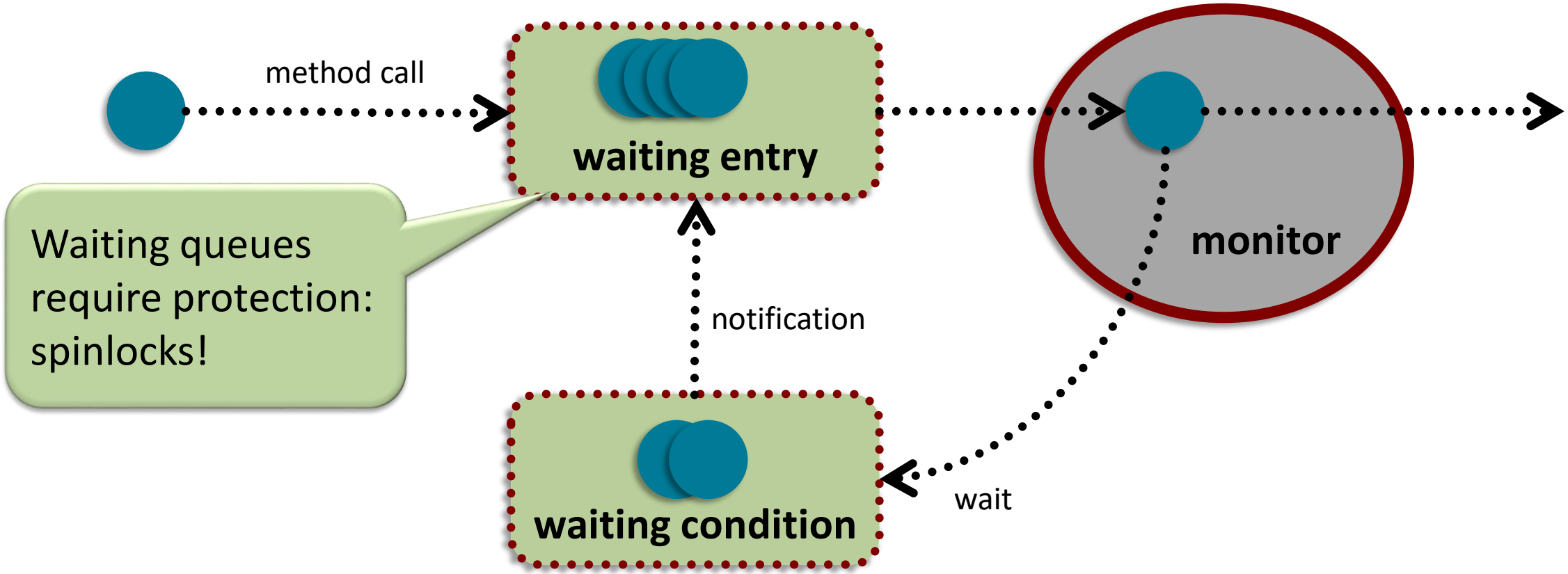
Literature:
-Herlihy Chapter 11.1 – 11.3
-Herlihy Chapter 9.8

Reminder: problems with spinlocks

- **Scheduling fairness / missing FIFO behavior.**
  - Solved with Queue locks – not presented in class but very nice!
- **Computing resources wasted, overall performance degraded, particularly for long-lived contention.**
- **No notification mechanism.**

Locks with waiting/scheduling

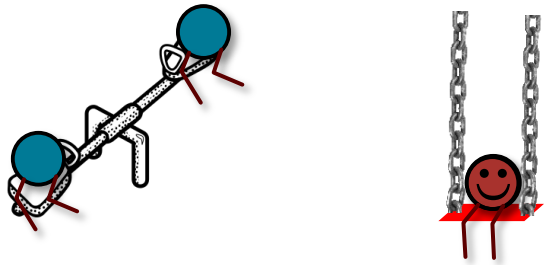
- **Locks that suspend the execution of threads while they wait. Semaphores, mutexes and monitors are typically implemented using a scheduled lock.**



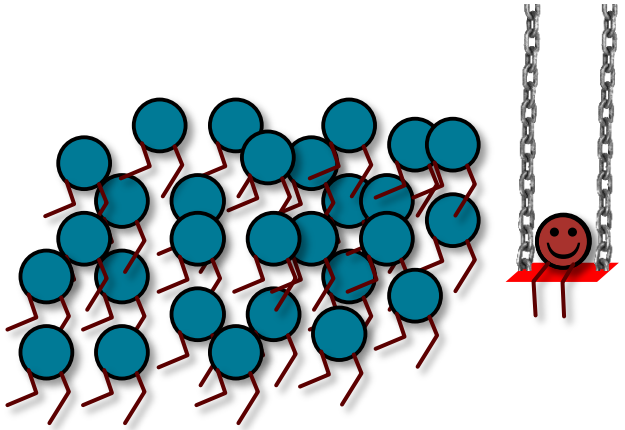
Locks with waiting/scheduling

- **Require support from the runtime system (OS, scheduler).**
- **Data structures for scheduled locks need to be protected against concurrent access, again using spinlocks, if not implemented lock-free (→ this lecture).**
- **Such locks have a higher wakeup latency (need to involve some scheduler).**
- **Hybrid solutions: try access with spinlock for a certain duration before rescheduling.**
  - Cf. “competitive spinning” (much later)

Locks performance



- **Uncontended case**
  - when threads do not compete for the lock
  - lock implementations try to have minimal overhead
  - typically "just" the cost of an atomic operation



- **Contended case**
  - when threads do compete for the lock
  - can lead to significant performance degradation
  - also, starvation
  - there exist lock implementations that try to address these issues

Disadvantages of locking

Locks are pessimistic by design

- Assume the worst and enforce mutual exclusion

Performance issues

- Overhead for each lock taken even in uncontended case
- Contended case leads to significant performance degradation
- Amdahl's law!

Blocking semantics (wait until acquire lock)

- If a thread is delayed (e.g., scheduler) when in a critical section → all threads suffer
- What if a thread dies in the critical section
- Prone to deadlocks (and also livelocks)
- Without precautions, locks cannot be used in interrupt handlers

Lock-Free Programming

Recap: Definitions for blocking synchronization

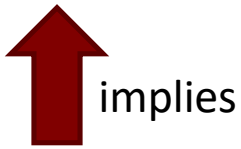
- Deadlock: group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed

- Livelock: competing processes are able to detect a potential deadlock but make no observable progress while trying to resolve it

- Starvation: repeated but unsuccessful attempt of a recently unblocked process to continue its execution

Definitions for Lock-free Synchronisation

- **Lock-freedom: at least one thread always makes progress even if other threads run concurrently.**
  Implies system-wide progress but not freedom from starvation.



- **Wait-freedom: all threads eventually make progress.**
  Implies freedom from starvation.

Progress conditions with and without locks

	Non-blocking (no locks)	Blocking (locks)
Everyone makes progress	Wait-free	Starvation-free
Someone make progress	Lock-free	Deadlock-free

Non-blocking Algorithms

Locks/blocking: a thread can indefinitely delay another thread

Non-blocking: failure or suspension of one thread cannot cause failure or suspension of another thread !

CAS (again)

compare **old** with data
at memory location

if and only if data at memory
equals **old** overwrite data with
new

return previous memory value
(in Java: return if success)

int CAS (memref a, int old, int new)

atomic

```
oldval = mem[a];
if (old == oldval)
    mem[a] = new;
return oldval;
```

CAS is more powerful than TAS as we will see later

CAS can be implemented wait-free (!) by hardware.

Non-blocking counter

Deadlock/Starvation?

```
public class CasCounter {
    private AtomicInteger value;

    public int getVal() {
       return value.get();
    }

    // increment and return new value
    public int inc() {
       int v;
       do {
              v = value.get();
       } while (!value.compareAndSet(v, v+1));
       return v+1;
    }
}
```

What happens if some processes see the same value?

Assume one thread dies.
Does this affect other threads?

Mechanism

(a) read current value v

(b) modify value v'

(c) try to set with CAS

(d) return if success
restart at (a) otherwise

Positive result of CAS of (c) *suggests* that no other thread has written between (a) and (c)

Handle CAS with care

Positive result of CAS *suggests* that no other thread has written

It is not always true, as we will find out (→ ABA problem).

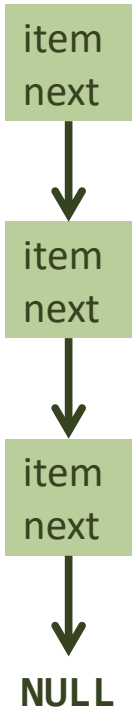
However, it is still THE mechanism to check for exclusive access in lock-free programming.

Sidenote: maybe transactional memory will become competitive at some point

Lock-Free Stack

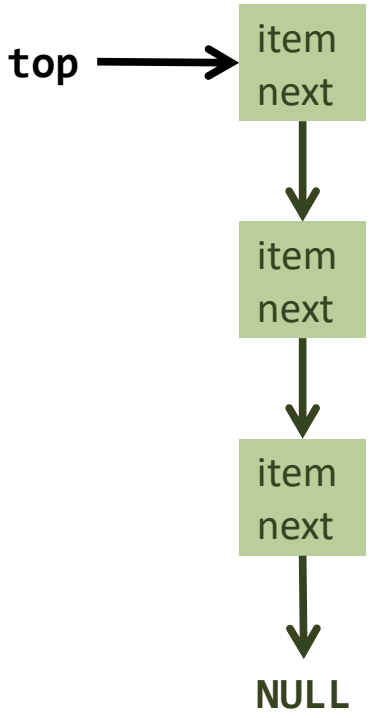
Stack Node

```
public static class Node {
   public final Long item;
   public Node next;

   public Node(Long item) {
       this.item = item;
   }

   public Node(Long item, Node n) {
       this.item = item;
       next = n;
   }

}
```



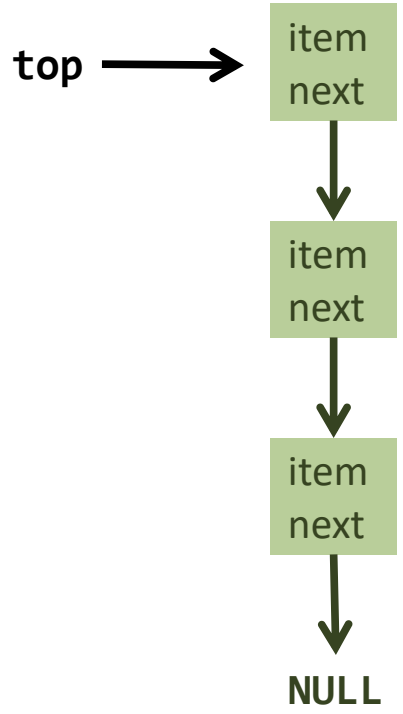
Blocking Stack

```
public class BlockingStack {
   Node top = null;

   synchronized public void push(Long item) {
      top = new Node(item, top);
   }

   synchronized public Long pop() {
      if (top == null)
         return null;
      Long item = top.item;
      top = top.next;
      return item;
   }
}
```



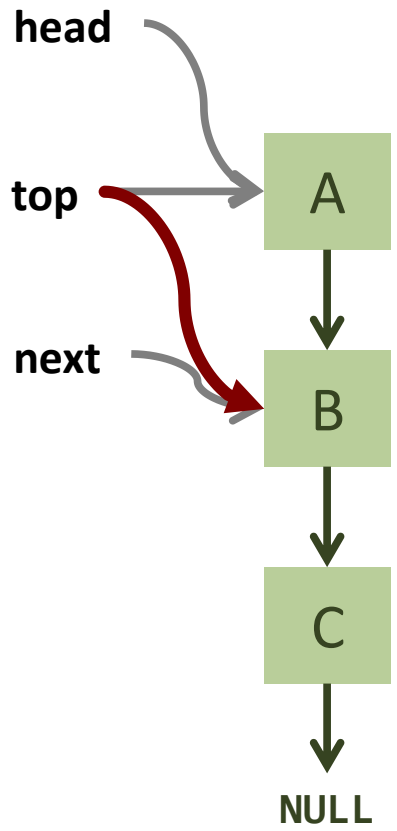
Non-blocking Stack

```
public class ConcurrentStack {
   AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```



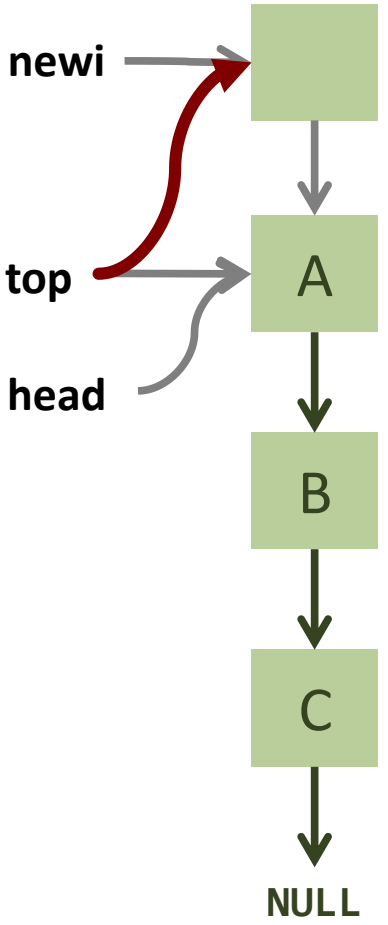
Pop

```
public Long pop() {
   Node head, next;

   do {
      head = top.get();
      if (head == null) return null;
      next = head.next;
   } while (!top.compareAndSet(head, next));

   return head.item;
}
```



Push

```
public void push(Long item) {
        Node newi = new Node(item);
        Node head;

        do {
                head = top.get();
                newi.next = head;
        } while (!top.compareAndSet(head, newi));
}
```



What's the benefit?

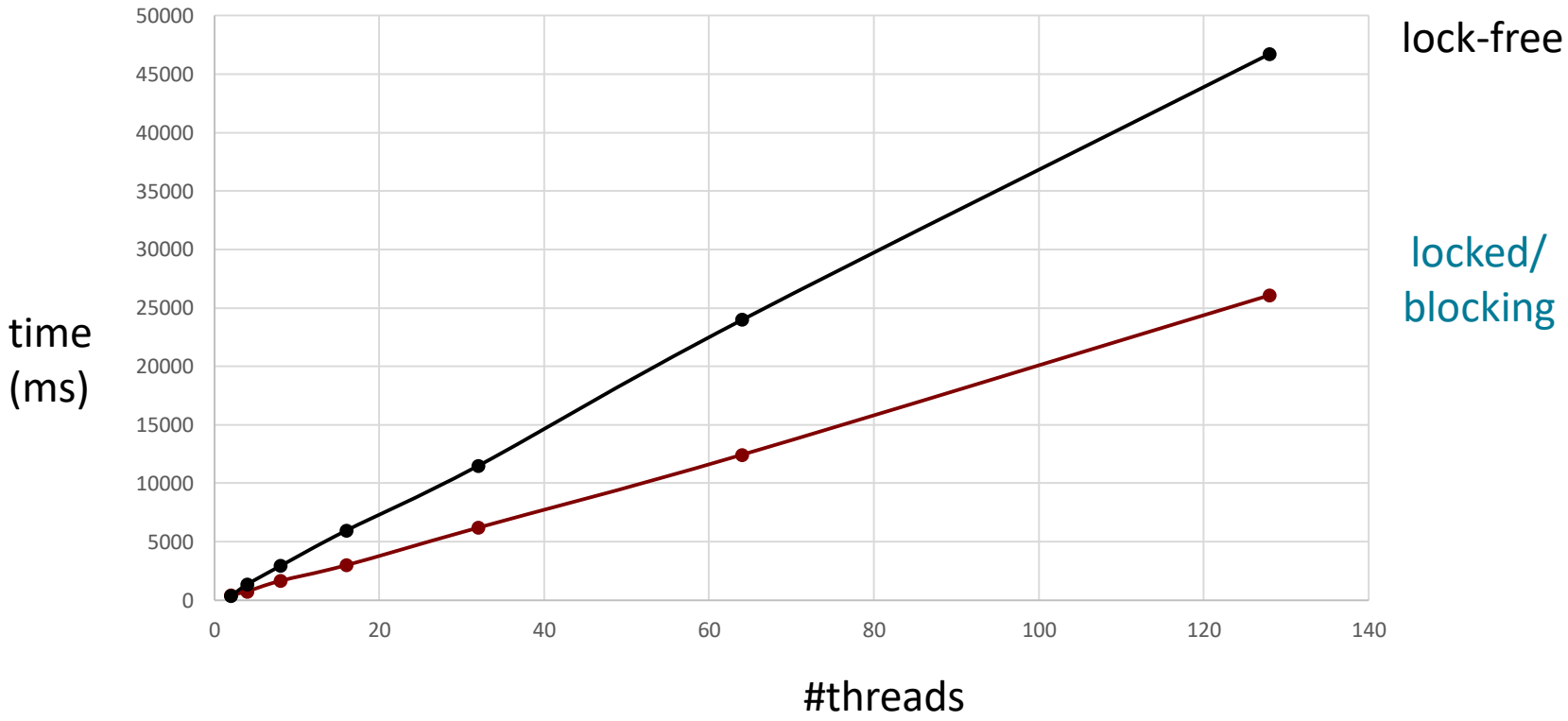
Lock-free programs are deadlock-free by design.

How about performance?

n threads

100,000 push/pop operations

10 times

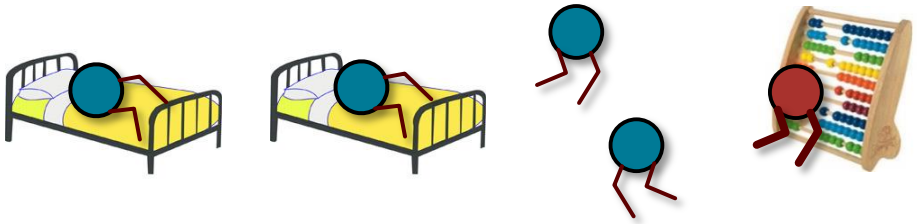


Performance

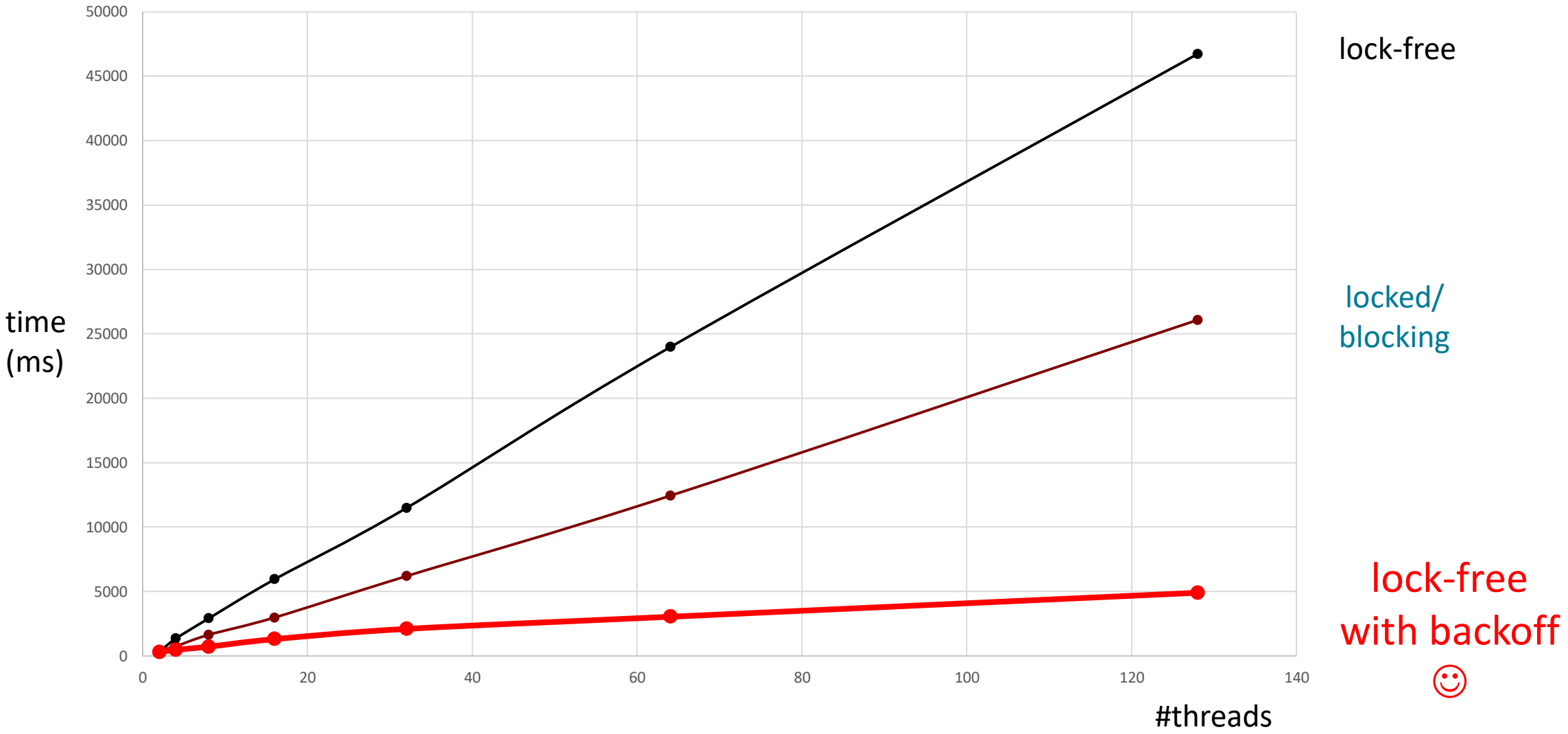
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

Atomic operations are expensive and contention can still be a problem.

→ Backoff, again.



With backoff



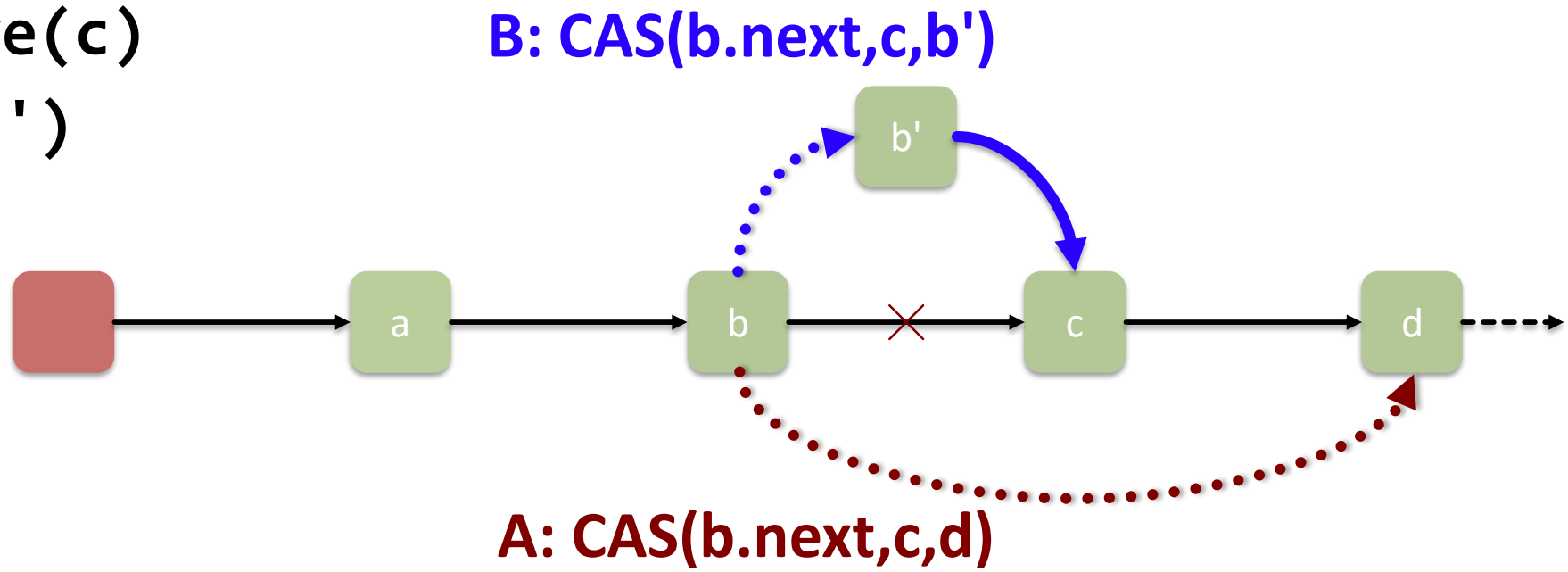
LOCK FREE LIST SET

(NOT SKIP LIST!)

Some of the material from "Herlihy: Art of Multiprocessor Programming"

Does this work?

```
A: remove(c)
B: add(b')
```



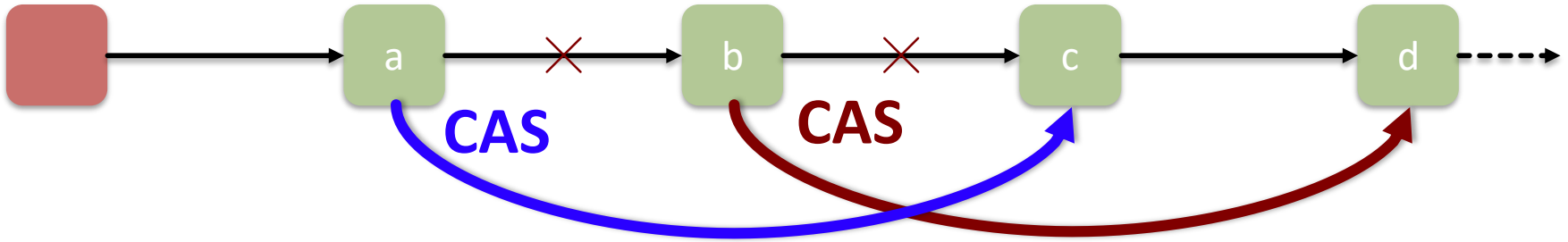
ok?

CAS decides who wins → this seems to work

So does this CAS approach work generally??

Another scenario

```
A: remove(c)
B: remove(b)
```



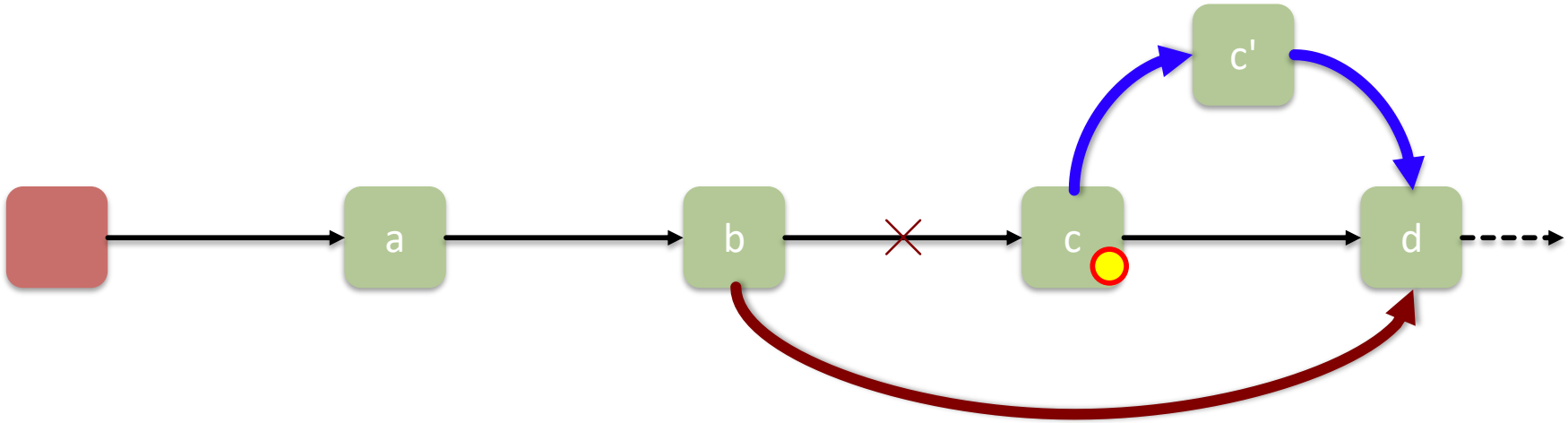
c not deleted! ☹

Mark Bit Approach?

```
A: remove(c)
B: add(c')
```

B: c.mark ?

B: CAS(c.next,d,c')



c' not added! ☹

A: CAS(c.mark,false,true)

A: CAS(b.next,c,d)

The Problem

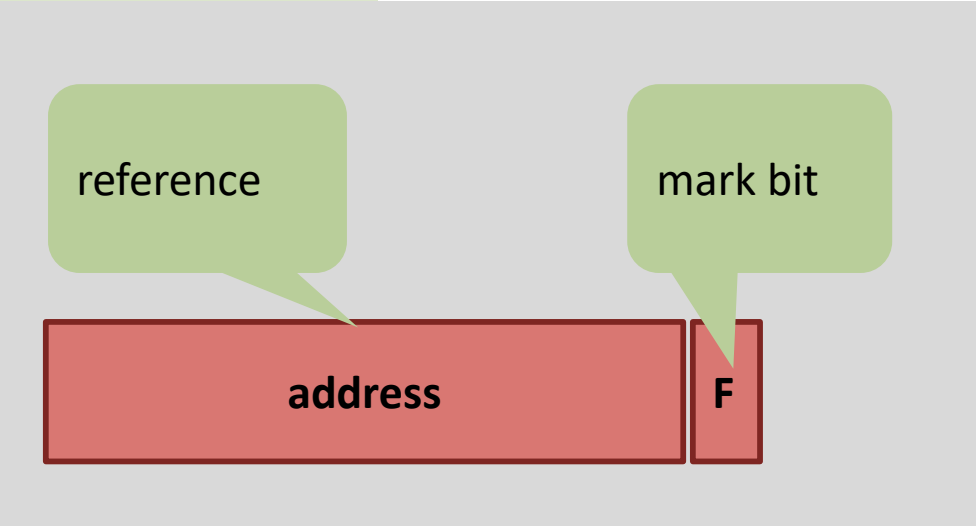
The difficulty that arises in this and many other problems is:

- We cannot (or don't want to) use synchronization via locks
- We still want to atomically establish consistency **of two things**
  Here: mark bit & next-pointer

The Java Solution

```
Java.util.concurrent.atomic
AtomicMarkableReference<V> {
    boolean attemptMark(V expectedReference, boolean newMark)
    boolean compareAndSet(V expectedReference, V newReference,
                          boolean expectedMark, boolean newMark)
    V get(boolean[] markHolder)
    V getReference()
    boolean isMarked()
    set(V newReference, boolean newMark)
}
```

DCAS on V and mark



The Algorithm using AtomicMarkableReference

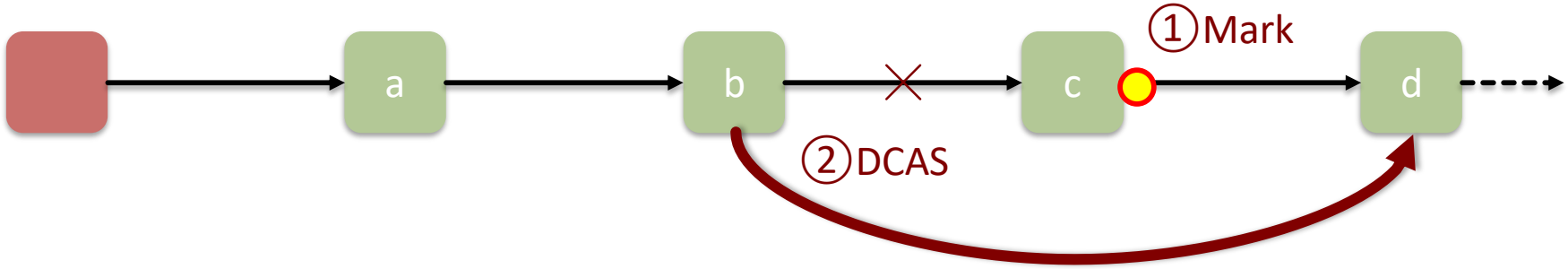
- **Atomically**
  - Swing reference *and*
  - Update flag
- **Remove in two steps**
  - Set mark bit in next field
  - Redirect predecessor's pointer

Algorithm Idea

A: `remove(c)`

Why "try to"? How can it fail? What then?

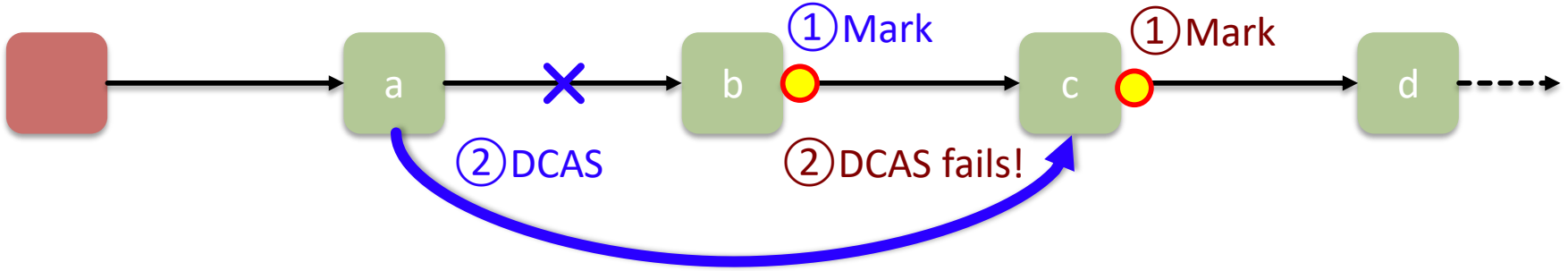
1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,unmarked], [d,unmarked]);



It helps!

```
A: remove(c)
B: remove(b)
```

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,**unmarked**], [d,unmarked]);



c remains marked ☹ (logically deleted)

1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

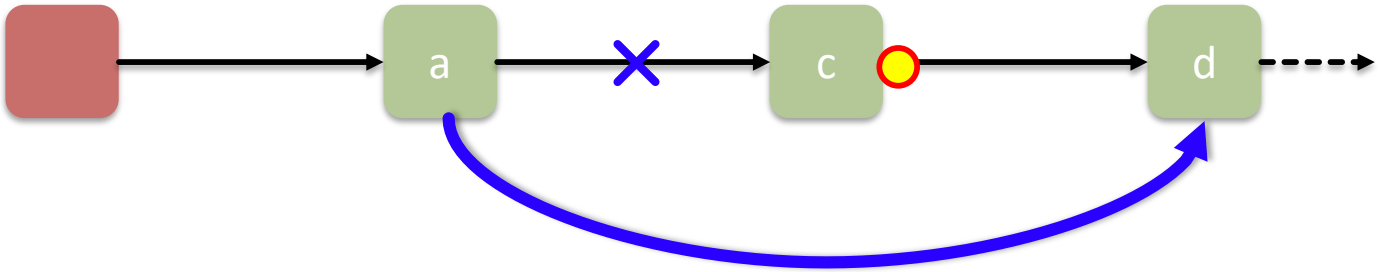
Traversing the List

Q: what do you do when you find a “logically” deleted node in your path?

A: finish the job.

CAS the predecessor’s next field

Proceed (repeat as needed)



Find Node

```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    while (true) {
        pred = head;
        curr = pred.next.getReference();
        boolean done = false;
        while (!done) {
            marked = curr.next.get(marked);
            succ = marked[1:n]; // pseudo-code to get next ptr
            while (marked[0] && !done) { // marked[0] is marked bit
                if pred.next.compareAndSet(curr, succ, false, false) {
                    curr = succ;
                    succ = curr.next.get(marked);
                }
                else done = true;
            }
            if (!done && curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
} } }
```

loop over nodes until position found

if marked nodes are found, delete them, if deletion fails restart from the beginning

```
class Window {
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        this.pred = pred;
        this.curr = curr;
    }
}
```

Remove

```
public boolean remove(T item) {
   Boolean snip;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) {
         return false;
      } else {
         Node succ = curr.next.getReference();
         snip = curr.next.attemptMark(succ, true);
         if (!snip) continue;
         pred.next.compareAndSet(curr, succ, false, false);
         return true;
      }
   }
}
```

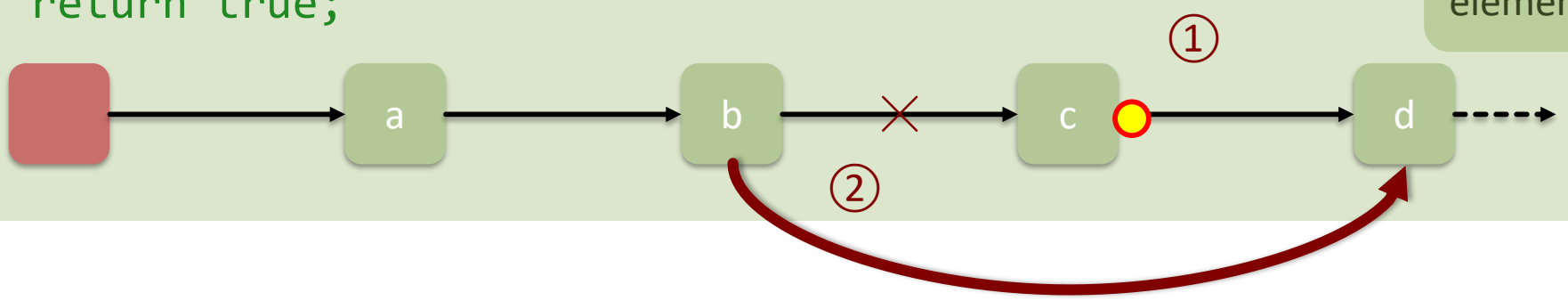
Find element and prev element from key

If no such element -> return false

Otherwise try to logically delete (set mark bit). ①

If no success, restart from the very beginning

Try to physically delete the element, ignore result ②



Add

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

and try to insert. If insertion fails (next set by other thread or mark bit set), retry

Observations

- We used a special variant of DCAS (double compare and swap) in order to be able check two conditions at once.
  This DCAS was possible because one bit was free in the reference.
- We used a lazy operation in order to deal with a consistency problem. Any thread is able to repair the inconsistency.
  If other threads would have had to wait for one thread to cleanup the inconsistency, the approach would not have been lock-free!
- This «helping» is a recurring theme, especially in wait-free algorithms where, in order to make progress, threads must help others (that may be off in the mountains ☺)