ETH zürich

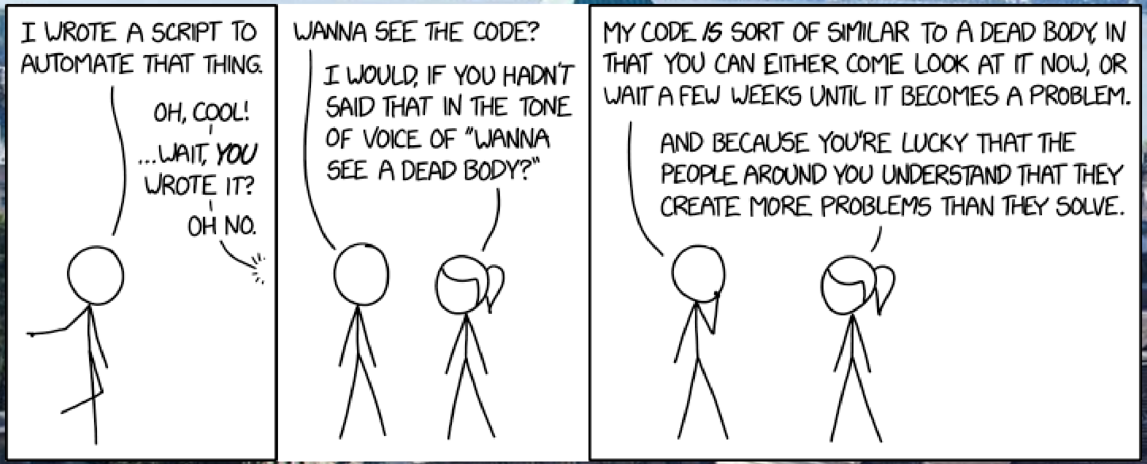
spcl.inf.ethz.ch
@spcl_eth

DINFK

TORSTEN HOEFLER

Parallel Programming

Producer/consumer and lock tricks



May 4, 2021

Parliament votes through €7.5B digital R&D programme

Funding boost for supercomputers and artificial intelligence will help the EU assert its digital sovereignty, MEPs say

By Florin Zubașcu

MEP Valter Flego, rapporteur for Digital Europe. Photo: European Parliament

The European Parliament has formally adopted the EU's new seven-year €7.5 billion investment plan in artificial intelligence, supercomputing and data platforms.

While MEPs are disappointed the programme's proposed budget was cut during the political negotiations on the seven year multi annual financial framework (MFF) at the end of last year, they see it as the EU's main tool for becoming less reliant on digital technologies from outside the bloc.

"Although the programme was not topped up in the final MFF agreement, it remains the most important tool to reach our goal of asserting Europe's digital sovereignty," said MEP Valter Flego, rapporteur for Digital Europe.

"[Digital Europe] will make it possible to reaffirm the EU geopolitical relevance in the second wave of digitalisation," said Portuguese MEP Carlos Zorinho.

In the face of the budget cut, MEPs secured key amendments to the legislation, giving the Parliament a greater say in how the money is spent and allowing third countries to be "partially" associated with the programme.

Digital Europe will invest €2.7 billion in supercomputing projects to build Europe's data processing capabilities, with the deployment of a supercomputer and data infrastructure with exascale capabilities by 2023, and post-exascale facilities by 2027. If successful, the investment will endow the EU with its own independent and competitive supply of high performance computing services.

"We mustn't forget that currently, the use of supercomputers is very expensive and is mainly found in third countries," said Conservative MEP Pilar del Castillo. "This in itself makes the case for proper EU funding for supercomputers based in Europe."

Another €2.5 billion will be invested in R&D projects to promote applications of artificial intelligence in the public and private sector. Public authorities and companies will have access to AI testing and experimentation facilities in member states, assuming the funding is complemented with investments in AI research and innovation under Horizon Europe.

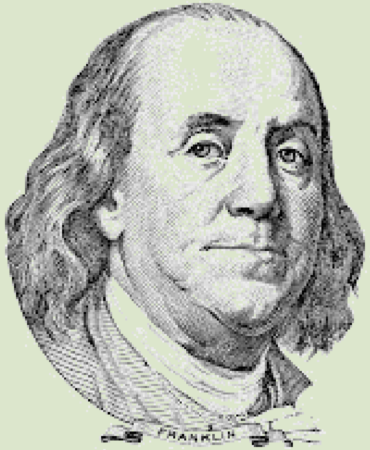
Last lecture

- **Barriers**
  - Multi-process synchronization, important in parallel programming
  - More examples for complexity of parallel programming (trial and error impossible)

- **Producer/Consumer in detail**
  - Queues, implementation
  - Deadlock cases (repetition)

Recap last lecture by a short quiz

- **Please participate in the live poll on movo.ch using the token specified! (it's fully anonymous)**



"Tell me and I forget, teach me and I may remember, involve me and I learn."

Learning goals today

- **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

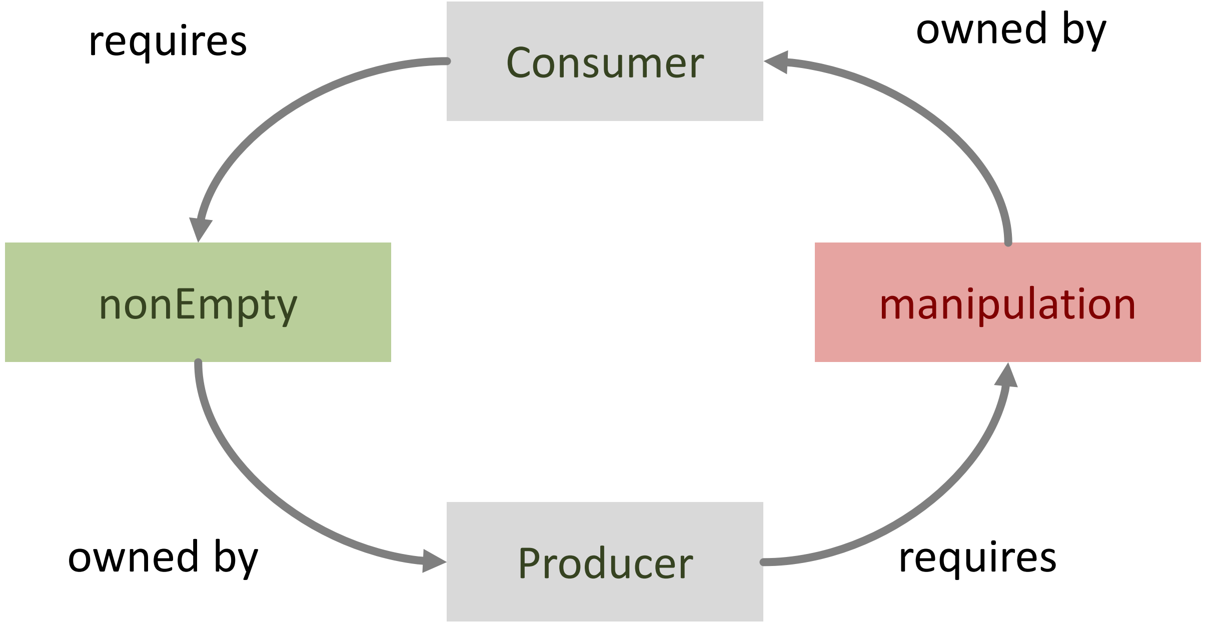
Producer / Consumer queues with semaphores, correct?

Do you see the problem?

```
void enqueue(long x) {

    try {
        manipulation.acquire();
        nonFull.acquire();
        buf[in] = x;
        in = (in+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        manipulation.acquire();
        nonEmpty.acquire();
        x = buf[out];
        out = (out+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

Deadlock (nearly the same as before, actually)!



Producer / Consumer queues with semaphores

In practice, some issue with interrupts remains

```
void enqueue(long x) {

    try {
        nonFull.acquire();
        manipulation.acquire();
        buf[in] = x;
        in = next(in);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buf[out];
        out = next(out);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

Why are semaphores (and locks) problematic?

Semaphores are unstructured. Correct use requires high level of discipline.

Easy to introduce deadlocks with semaphores.

We need: a lock that we can temporarily escape from when waiting on a condition.

Monitors

Monitors

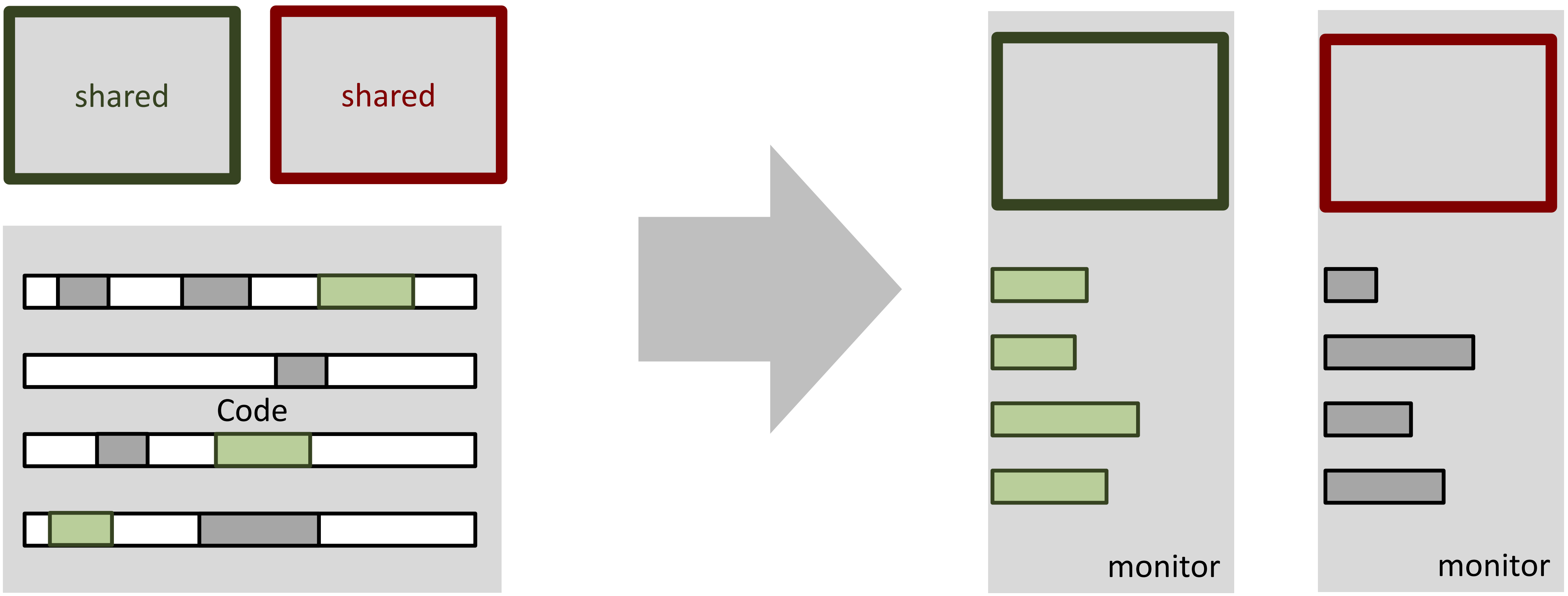
Monitor:
abstract data structure equipped with a set of operations that run in mutual exclusion.

Invented by Tony Hoare and Per Brinch Hansen (cf. Monitors: An Operating System Structuring Concept, Tony Hoare, 1974)

Tony Hoare
(1934-today)

Per Brinch Hansen
(1938-2007)

Monitors vs. Semaphores/Unbound Locks



Producer / Consumer queues

```
public void synchronized enqueue(long item) {
   "while (isFull()) wait"
   doEnqueue(item);
}
```

```
public long synchronized dequeue() {
   "while (isEmpty()) wait"
   return doDequeue();
}
```

The mutual exclusion part is nicely available already.
But: while the buffer is full we need to give up the lock, how?

Monitors

Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:

If a condition does not hold
- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops (spinning)

Monitors in Java

Uses the intrinsic lock (`synchronized`) of an object

+ `wait / notify / notifyAll`:

`wait()` – the current thread waits until it is signaled (via notify)

`notify()` – wakes up *one* waiting thread (an arbitrary one)

`notifyAll()` – wakes up *all* waiting threads

Producer / Consumer with monitor in Java

```
class Queue {
      int in, out, size;
      long buf[];

      Queue(int s) {
            size = s;
            buf = new long[size];
            in = out = 0;
      }
      ...
}
```

Producer / Consumer with monitor in Java

```
synchronized void enqueue(long x) {

    while (isFull())
        try {
            wait();
        } catch (InterruptedException e) { }
    doEnqueue(x);
    notifyAll();
}
```

```
synchronized long dequeue() {
    long x;
    while (isEmpty())
        try {
            wait();
        } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

Wouldn't an if be sufficient?

(Why) can't we use notify()?

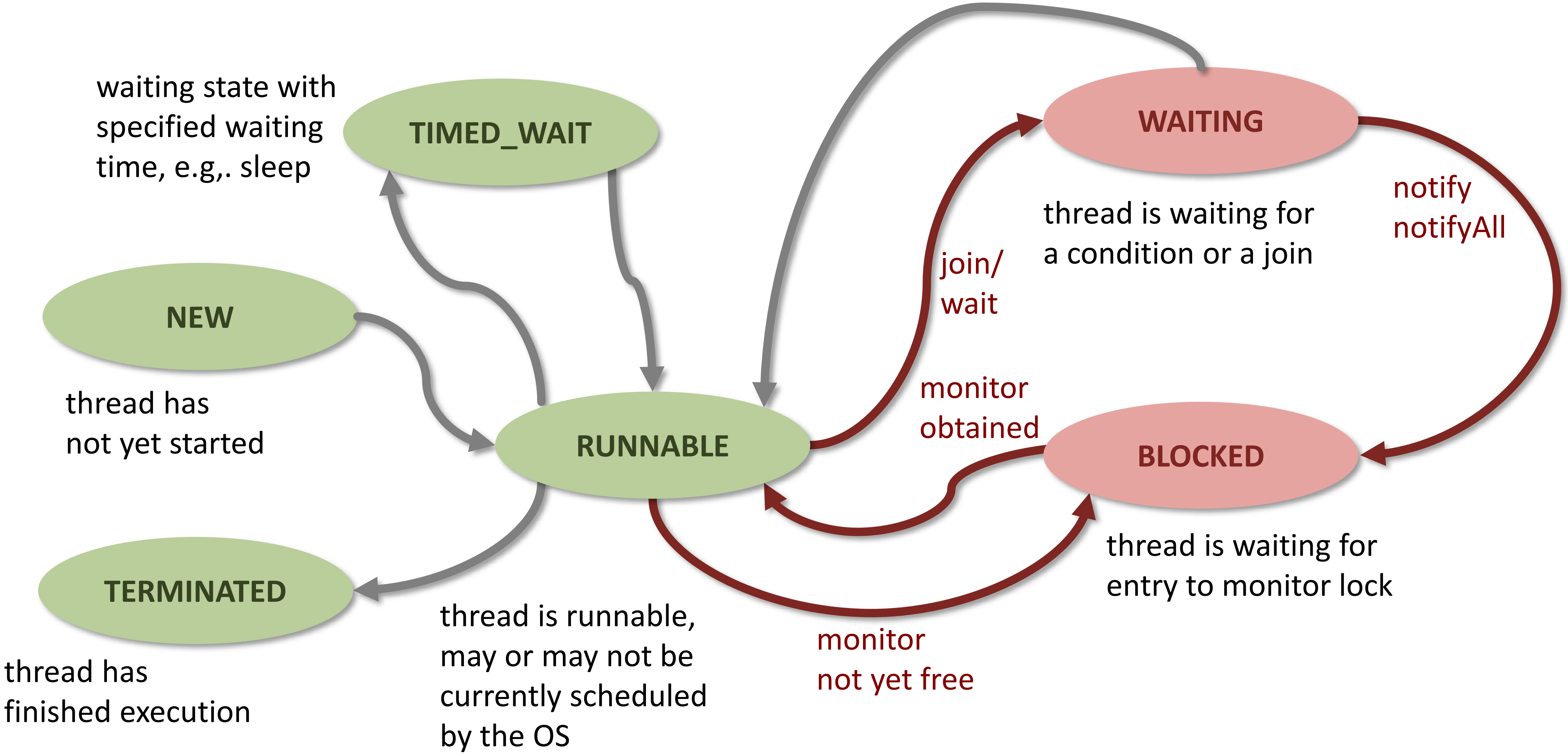
IMPORTANT TO KNOW JAVA MONITOR IMPLEMENTATION DETAILS

Thread States in Java

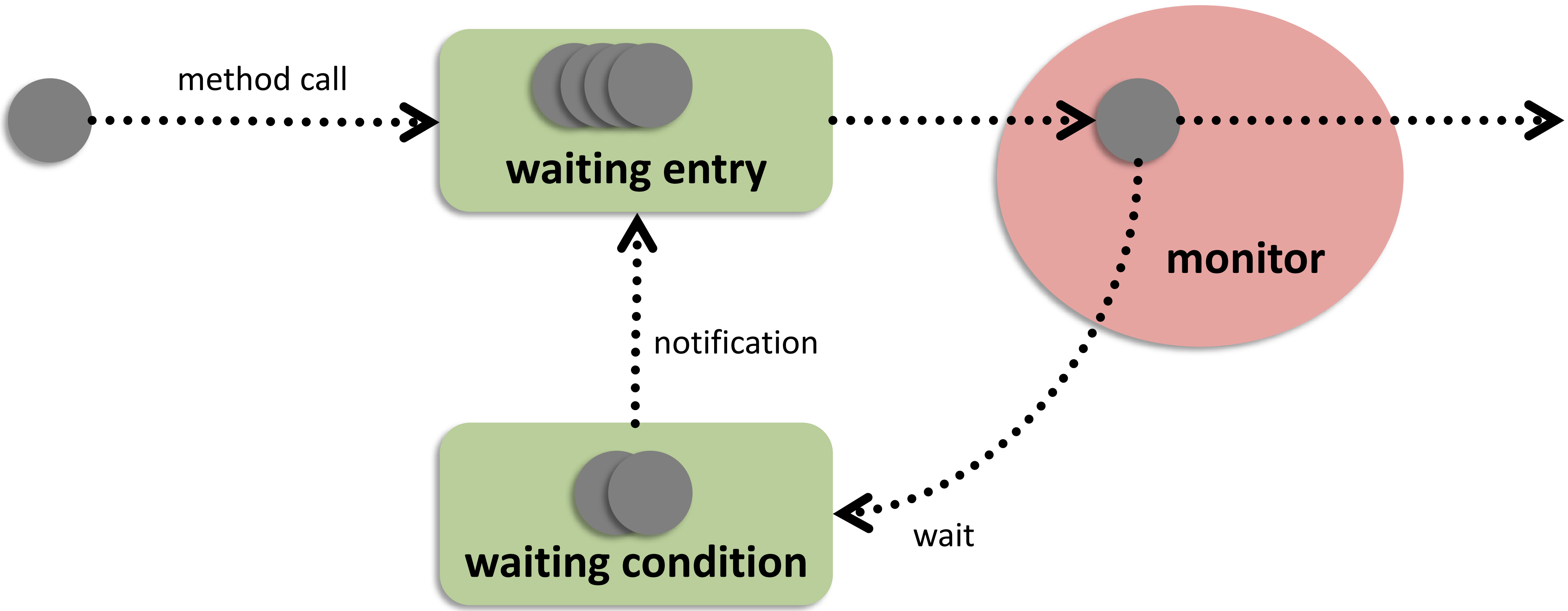
Spaced repetition

From Wikipedia, the free encyclopedia

Spaced repetition is a learning technique that incorporates increasing intervals of time between subsequent review of previously learned material in order to exploit the psychological spacing effect. Alternative names include *spaced rehearsal*, *expanding rehearsal*, *graduated intervals*, *repetition spacing*, *repetition scheduling*, *spaced retrieval* and *expanded retrieval*.[1]



Monitor Queues



Various (exact) semantics possible

Important to know for the programmer (you): what happens upon notification? Priorities?

signal and wait

- signaling process exits the monitor (goes to waiting entry queue)
- signaling process passes monitor lock to signaled process

signal and continue

- signaling process continues running
- signaling process moves signaled process to waiting entry queue

other semantics: signal and exit, signal and urgent wait …

Why is this important? Let's try this implementing a semaphore:

```
class Semaphore {
   int number = 1; // number of threads allowed in critical section

   synchronized void acquire() {
      if (number <= 0)
         try { wait(); } catch (InterruptedException e) { };
      number--;
   }

   synchronized void release() {
      number++;
      if (number > 0)
         notify();
   }
}
```

Looks good, doesn't it?
But there is a problem.
Do you know which?

Java Monitors = signal + continue

```
R synchronized void acquire() {
    if (number <= 0)
       try { wait(); } Q
       catch (InterruptedException e) { };
    number--;
  }

  synchronized void release() {
P   number++;
    if (number > 0)
       notify();
  }
```

Scenario:

1. **Process P has previously acquired the semaphore and decreased number to 0.**
2. **Process Q sees number = 0 and goes to waiting list.**
3. **P is executing release. In this moment process R wants to enter the monitor via method acquire.**
4. **P signals Q and thus moves it into wait entry list (signal and continue!). P exits the function/lock.**
5. **R gets entry to monitor before Q and sees the number = 1**
6. **Q continues execution with number = 0!**

Inconsistency!

The cure – a while loop.

```
synchronized void acquire() {
 while (number <= 0)
  try { wait(); }
   catch (InterruptedException e) { };
 number--;
}
```

```
synchronized void release() {
   number++;
   if (number > 0)
      notify();
}
```

If, additionally, different threads evaluate different conditions, the notification has to be a `notifyAll`. In this example it is not required.

Something different: Java Interface Lock

Intrinsic locks ("synchronized") with objects provide a good abstraction and should be first choice

Limitations

- one implicit lock per object
- are forced to be used in blocks
- limited flexibility

Java offers the Lock interface for more flexibility (e.g., lock can be polled).

```
final Lock lock = new ReentrantLock();
```

Condition interface

Java Locks provide *conditions that can be instantiated*

```
Condition notFull  = lock.newCondition();
```

Java conditions offer

.await() – the current thread waits until condition is signaled

.signal() – wakes up one thread *waiting on this condition*

.signalAll() – wakes up all threads *waiting on this condition*

Condition interface

→ **Conditions are always associated with a lock**
lock.newCondition()

.await()

- called with the lock held
- **atomically** releases the lock and waits until thread is signaled
- When it returns, it is **guaranteed** to hold the lock
- thread **always** needs to check condition

.signal{,All}() – wakes up one (all) waiting thread(s)

- called with the lock held

Producer / Consumer with explicit Lock

```
class Queue {
     int in=0, out=0, size;
     long buf[];
     final Lock lock = new ReentrantLock();
     final Condition notFull  = lock.newCondition();
     final Condition notEmpty = lock.newCondition();

     Queue(int s) {
          size = s;
          buf = new long[size];
     }

...

}
```

Producer / Consumer with explicit Lock

```
void enqueue(long x) {

   lock.lock();
   while (isFull())
      try {
         notFull.await();
      } catch (InterruptedException e){}
   doEnqueue(x);
   notEmpty.signal();
   lock.unlock();
}
```

```
long dequeue() {
   long x;
   lock.lock();
   while (isEmpty())
      try {
         notEmpty.await();
      } catch (InterruptedException e){}
   x = doDequeue();
   notFull.signal();
   lock.unlock();
   return x;
}
```

The Sleeping Barber Variant (E. Dijkstra)

Disadvantage of the solution: notFull and notEmpty signal will be sent in any case, even when no threads are waiting.

Seemingly simple solution (in barber analogy)

1. ***Barber cuts hair, when done, check waiting room, if nobody left, sleep***
2. ***Client arrives, either enqueues or wakes sleeping barber***

What can go wrong (really only in a threaded world)?

Sleeping barber **requires additional counters for checking if processes are waiting:**

$m \leq 0 \Leftrightarrow$ **buffer full &** $-m$ **producers (clients) are waiting**

$n \leq 0 \Leftrightarrow$ **buffer empty &** $-n$ **consumers (barbers) are waiting**

Producer Consumer, Sleeping Barber Variant

```
class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull  = lock.newCondition();
    int m; final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s; m=size-1;
        buf = new long[size];
    }
...
}
```

Two variables ☹ sic!
(cf. last lecture)

Producer Consumer, Sleeping Barber Variant

```
void enqueue(long x) {

   lock.lock();
   m--; if (m<0)
      while (isFull())
         try { notFull.await(); }
         catch(InterruptedException e){}
   doEnqueue(x);
   n++;
   if (n<=0) notEmpty.signal();
   lock.unlock();

}
```

```
long dequeue() {
   long x;
   lock.lock();
   n--; if (n<0)
      while (isEmpty())
         try { notEmpty.await(); }
         catch(InterruptedException e){}
   x = doDequeue();
   m++;
   if (m<=0) notFull.signal();
   lock.unlock();
   return x;
}
```

Guidelines for using condition waits

- **Always have a condition predicate**
- **Always test the condition predicate:**
  - before calling wait
  - after returning from wait
- **Always call wait in a loop**
- **Ensure state is protected by lock associated with condition**

Check out java.util.concurrent

Java (luckily for us) provides many common synchronization objects:

- Semaphores
- Barriers (CyclicBarrier)
- Producer / Consumer queues
- and many more... (Latches, Futures, ...)

Reader / Writer Locks

Literature: Herlihy – Chapter 8.3

Reading vs. writing

Recall:

- **Multiple concurrent reads of same memory: *Not* a problem**
- **Multiple concurrent writes of same memory: Problem**
- **Multiple concurrent read & write of same memory: Problem**

So far:

- **If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time**

But this is unnecessarily conservative:

- **Could still allow multiple simultaneous readers!**

Example

Consider a hashtable with one coarse-grained lock

- So only one thread can perform operations at a time

But suppose:

- There are many simultaneous **`lookup`** operations
- **`insert`** operations are very rare

Note: Important that `lookup` does not actually mutate shared memory, like a move-to-front list operation would

WIKIPEDIA
The Free Encyclopedia

Number of edits (2007-04/26/2021): 1,020,000,000
Average views per day: ~200,000,000

→ 0.12% write rate

Reader/writer locks

A new abstract data type for synchronization : The reader/writer lock

This lock's states fall into three categories:

- **"not held"**
- **"held for writing" by one thread**
- **"held for reading" by *one or more* threads**

$0 \leq$ writers ≤ 1
$0 \leq$ readers
writers*readers == 0

Reader/writer locks

`new:`	make a new lock, initially "not held"
`acquire_write:`	block if currently "held for reading" or "held for writing", else make "held for writing"
`release_write:`	make "not held"
`acquire_read:`	block if currently "held for writing", else make/keep "held for reading" and increment ***readers count***
`release_read:`	decrement readers count, if 0, make "not held"

Pseudocode example

```
class Hashtable<K,V> {
 …
 // coarse-grained, one lock for table
 RWLock lk = new RWLock();
 …
```

```
 V lookup(K key) {
   int bucket = hashval(key);
   lk.acquire_read();
   … read array[bucket] …
   lk.release_read();
 }
```

```
  void insert(K key, V val) {
    int bucket = hashval(key);
    lk.acquire_write();
    … write V to array[bucket] …
    lk.release_write();
  }
  …
}
```

A Simple Monitor-based Implementation

```
class RWLock {
  int writers = 0;
  int readers = 0;

synchronized void acquire_read() {
  while (writers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
  readers++;
}

synchronized void release_read() {
  readers--;
  notifyAll();
}
```

Is this lock fair?

The simple implementation gives priority to readers:
- when a reader reads, other readers can enter
- no writer can enter during readers reading

```
 synchronized void acquire_write() {
   while (writers > 0 || readers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
   writers++;
 }

 synchronized void release_write() {
  writers--;
  notifyAll();
 }
}
```

Exercise: come up with a better performing version using condition variables!

Strong priority to the writers

```
class RWLock {
  int writers = 0;
  int readers = 0;
  int writersWaiting = 0;

  synchronized void acquire_read() {
    while (writers > 0 || writersWaiting > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    readers++;
  }

  synchronized void release_read() {
    readers--;
    notifyAll();
  }
```

```
  synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    writersWaiting--;
    writers++;
  }

  synchronized void release_write() {
    writers--;
    notifyAll();
  }
}
```

Is this lock now fair?
(this was just to see of you're awake)

A fair(er) model

What is fair in this context?

For example

- **When a writer finishes, a number k of currently waiting readers may pass.**
- **When the k readers have passed, the next writer may enter (if any), otherwise further readers may enter until the next writer enters (who has to wait until current readers finish).**

A fair(er) model

writers: # writers in CS
readers: # readers in CS
writersWaiting: # writers trying to enter CS
readersWaiting: # readers trying to enter CS
writersWait: # readers the writers have to wait

```
class RWLock{
  int writers = 0; int readers = 0;
  int writersWaiting = 0; int readersWaiting = 0;
  int writersWait = 0;

  synchronized void acquire_read() {
   readersWaiting++;
   while (writers > 0 ||
         (writersWaiting > 0 && writersWait <= 0))
     try { wait(); }
     catch (InterruptedException e) {}
   readersWaiting--;
   writersWait--;
   readers++;
  }

  synchronized void release_read() {
   readers--;
   notifyAll();
  }

  synchronized void acquire_write() {
   writersWaiting++;
   while (writers > 0 || readers > 0 || writersWait > 0)
      try { wait(); }
     catch (InterruptedException e) {}
   writersWaiting--;
   writers++;
  }

  synchronized void release_write() {
   writers--;
   writersWait = readersWaiting;
   notifyAll();
  }
}
```

Writers are waiting and the readers don't have priority any more.

When a writer finishes, the number of currently waiting readers may pass.

Writers have to wait until the waiting readers have finished.

Exercise: come up with a better performing version using condition variables! Introduce an upper bound of k readers!

Reader/writer lock details

A reader/writer lock implementation (“not our problem”) usually gives *priority* to writers:

- Once a writer blocks, no readers *arriving later* will get the lock before the writer
- Otherwise an `insert` could *starve*

Re-entrant?

- Mostly an orthogonal issue
- But some libraries support *upgrading* from reader to writer

In Java

Java's `synchronized` statement does not support readers/writer

Instead, library

`java.util.concurrent.locks.ReentrantReadWriteLock`

Different interface: methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods

Does *not* have writer priority or reader-to-writer upgrading

- Always read the documentation

LOCK GRANULARITY

Literature: Herlihy – Chapter 9

The Five-Fold Path

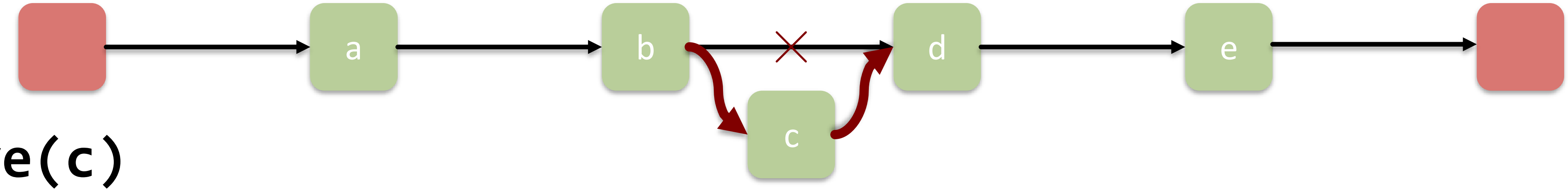
- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization (locking)
- Lazy synchronization (locking)

- Next lecture: Lock-free synchronization

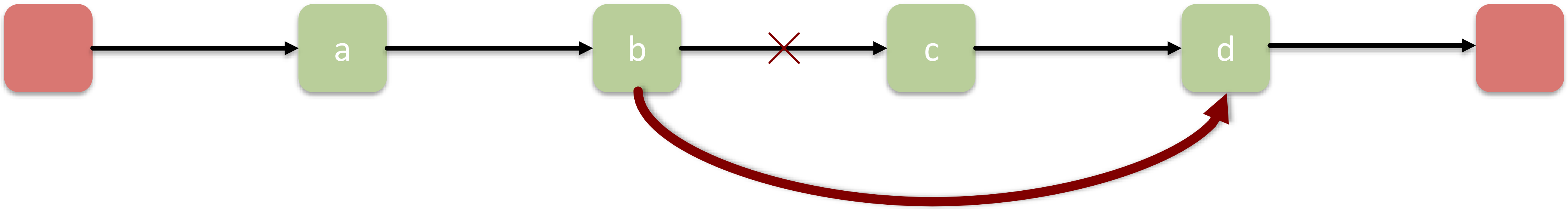
Running Example: Sequential List Based Set

Add, Remove, and Find unique elements in a sorted linked list.

- `add(c)`

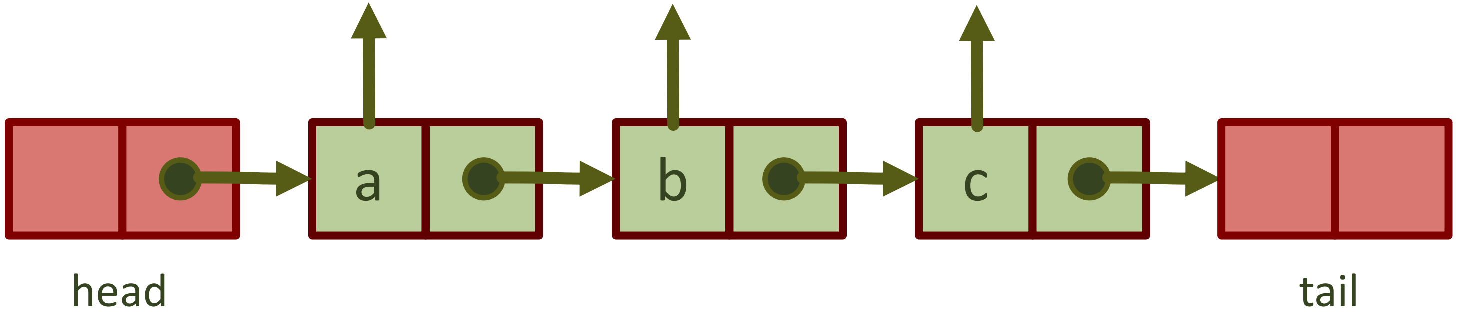


- `remove(c)`



Set and Node

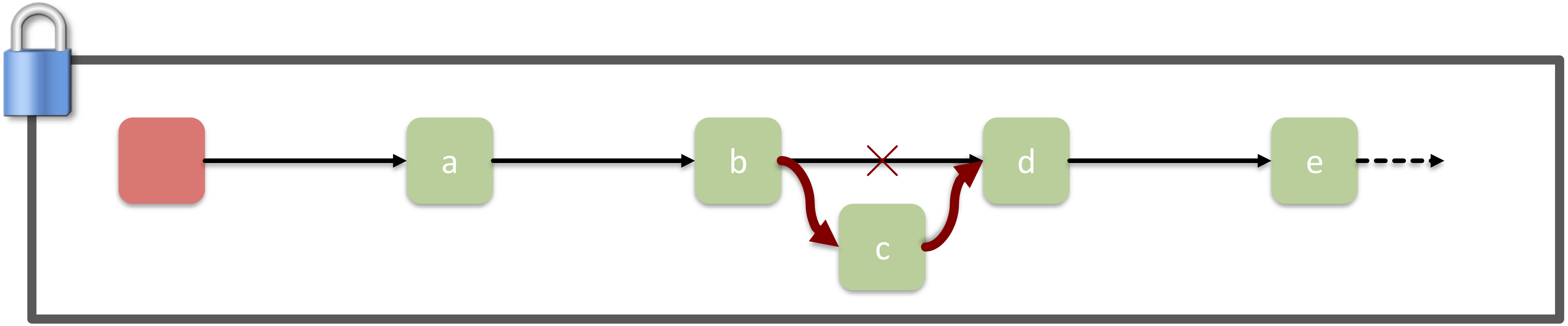
```
public class Set<T> {

   private class Node {
      T item;
      int key;
      Node next;
   }
   private Node head;
   private Node tail;

   public boolean add(T x) {...};
   public boolean remove(T x) {...};
   public boolean contains(T x) {...};
}
```



Note that the list is not "in place" but provides references to its items

Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```



Simple, but a bottleneck for all threads.

Fine grained Locking

Often more intricate than visible at a first sight

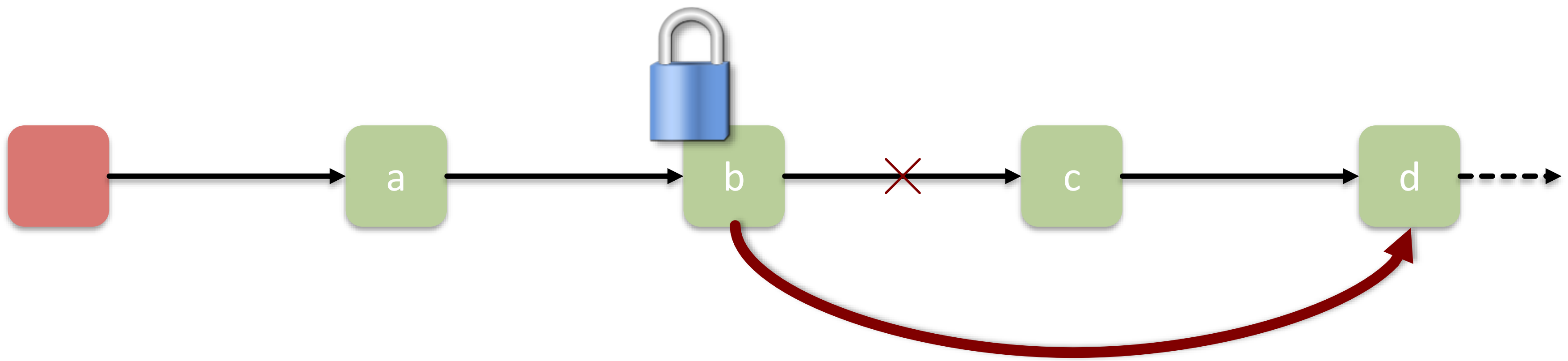
- requires careful consideration of special cases

Idea: split object into pieces with separate locks

- no mutual exclusion for algorithms on disjoint pieces

Let's try this

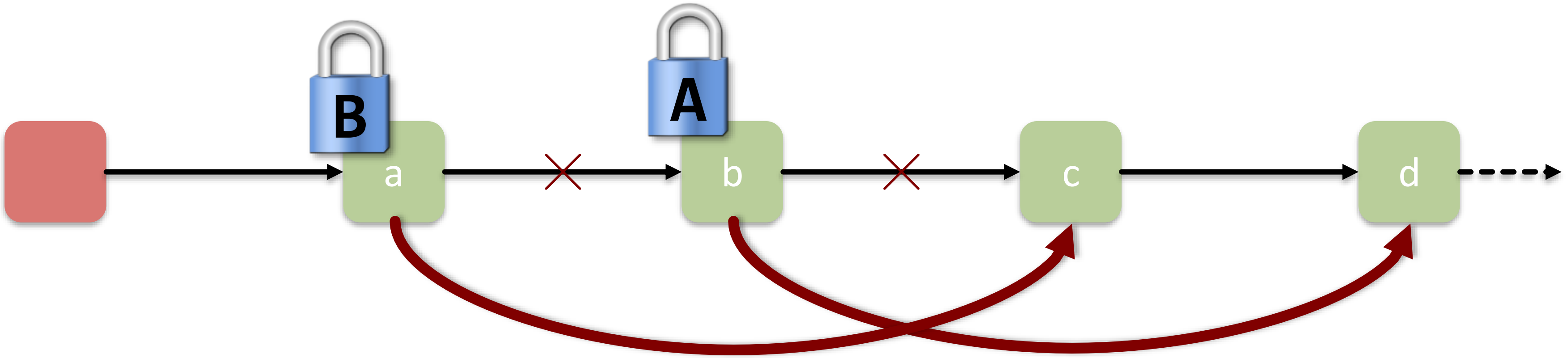
`remove(c)`



`Is this ok?`

Let's try this

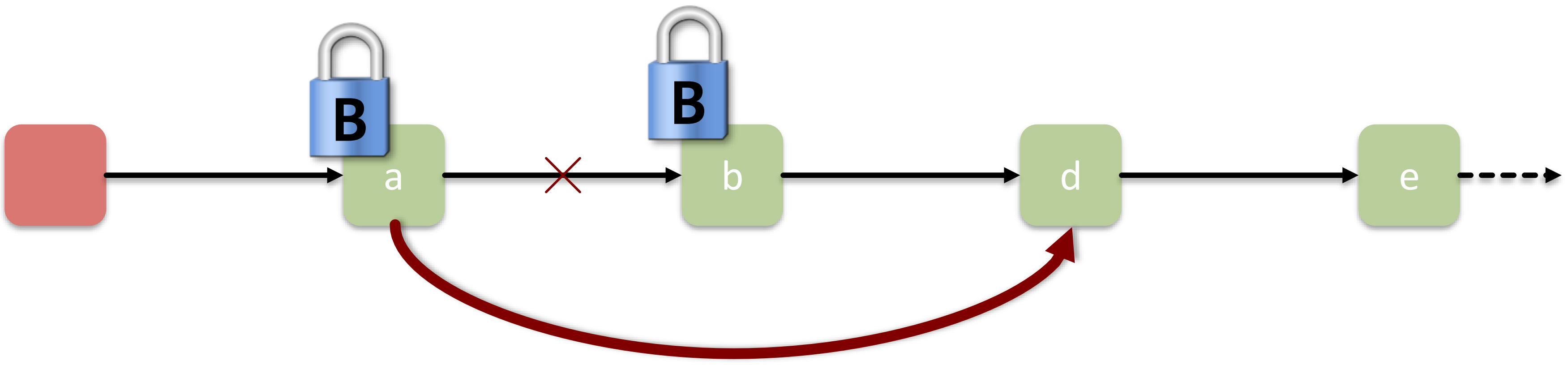
```
Thread A: remove(c)
Thread B: remove(b)
```



c not deleted! ☹

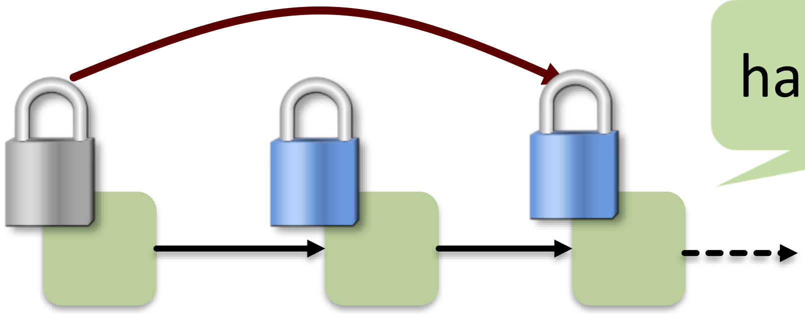
What's the problem?

- **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"

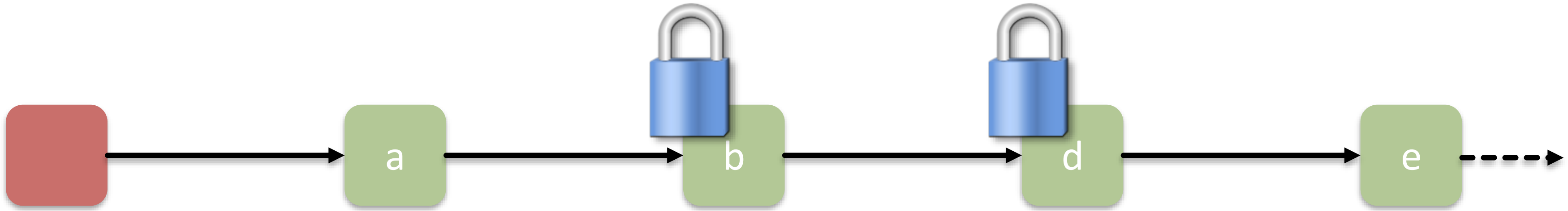
OPTIMISTIC SYNCHRONIZATION

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

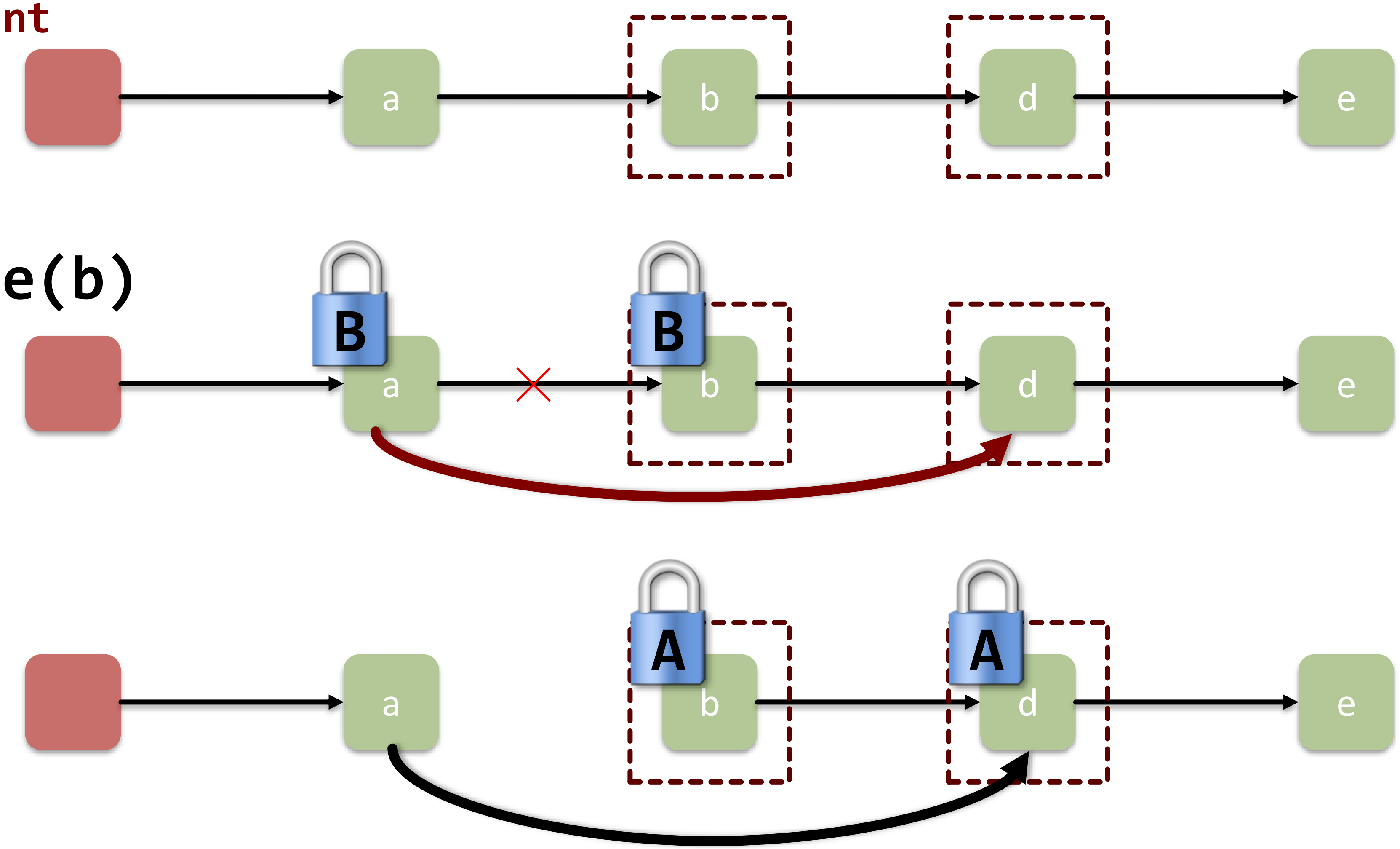
Thread B: remove(b)

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

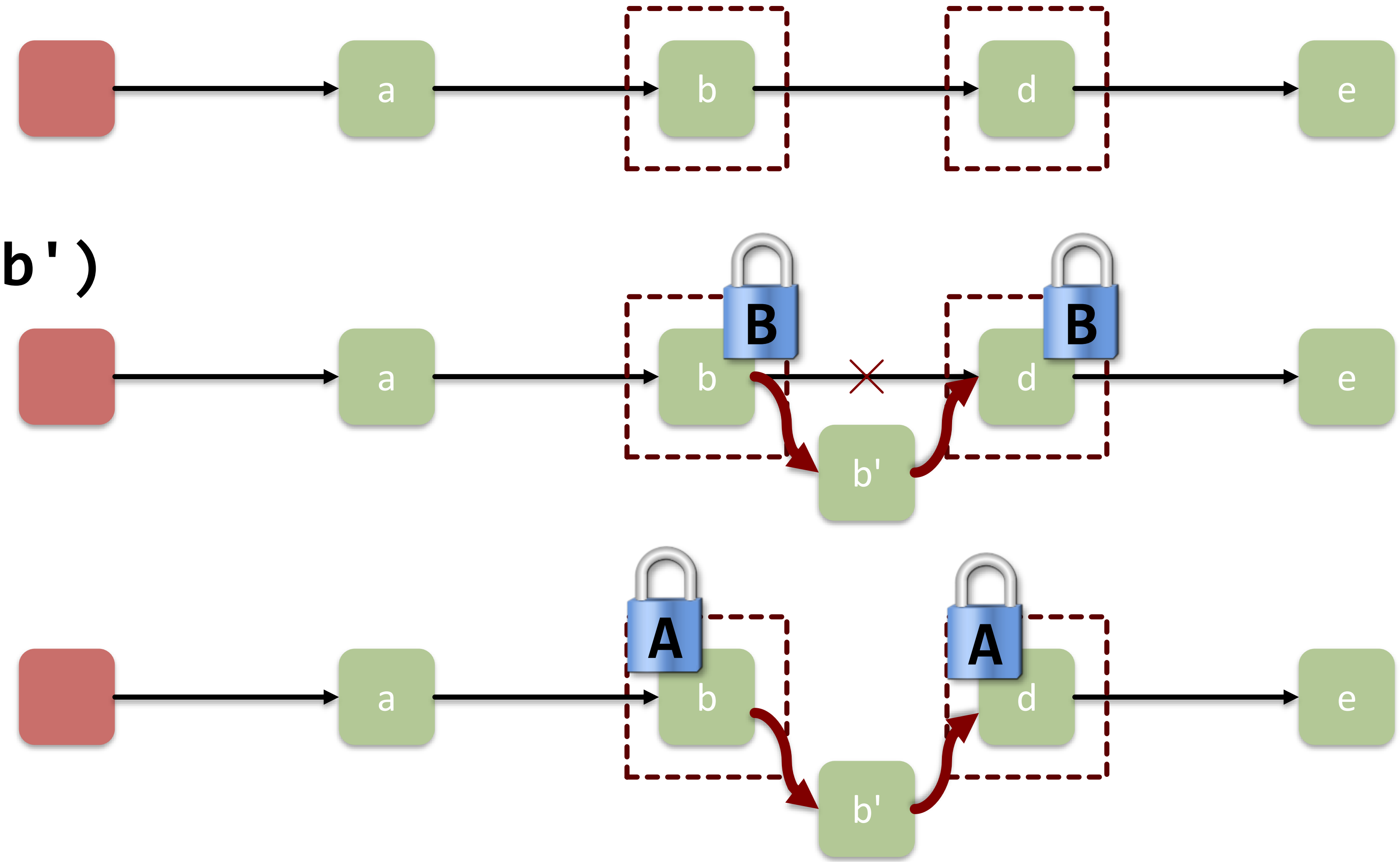
Thread B: insert(b')

A: lock

A: validate: rescan

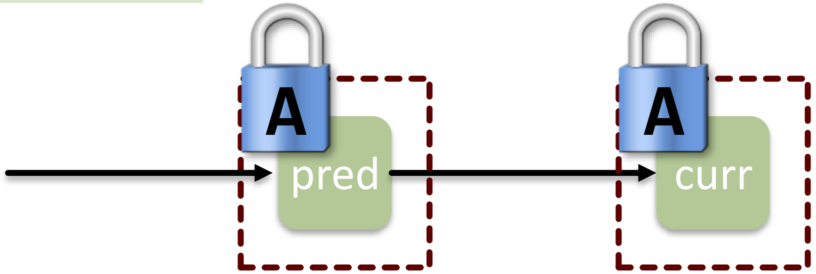
A: d != succ(b)

→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

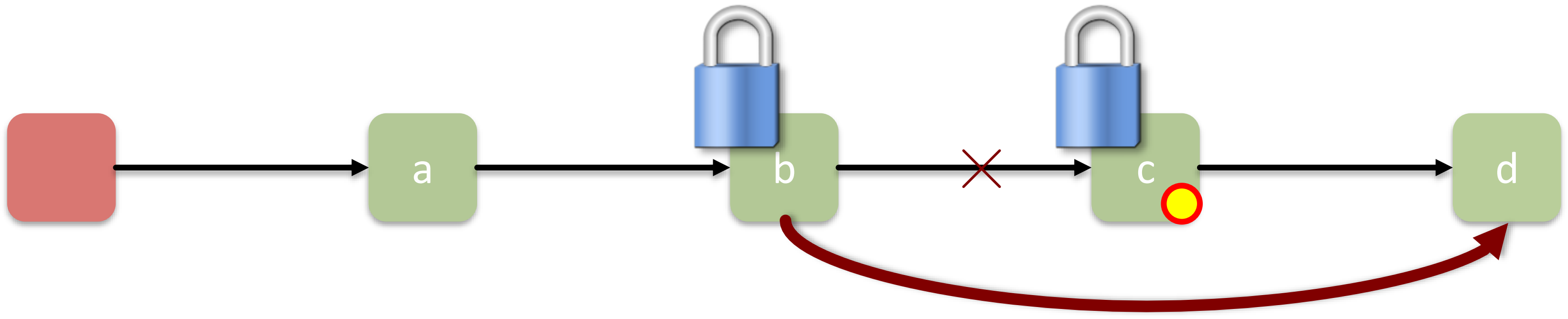
Scan list (as before)

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)

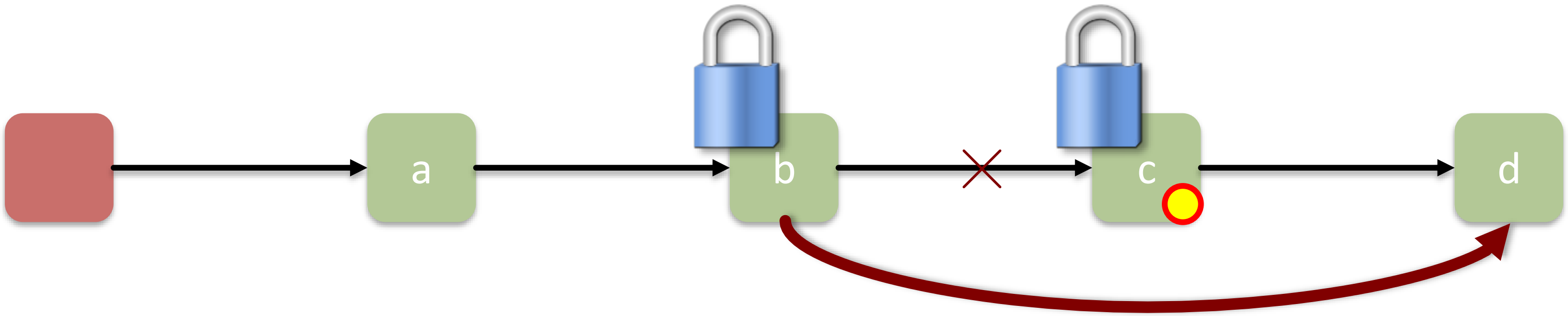
lock

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.

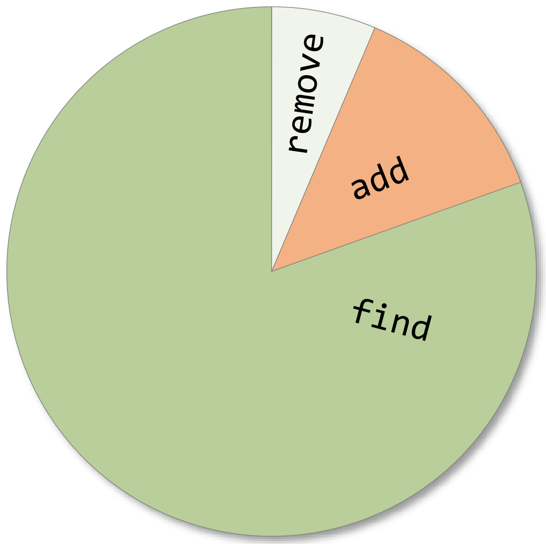
More practical:
Lazy Skip Lists

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()



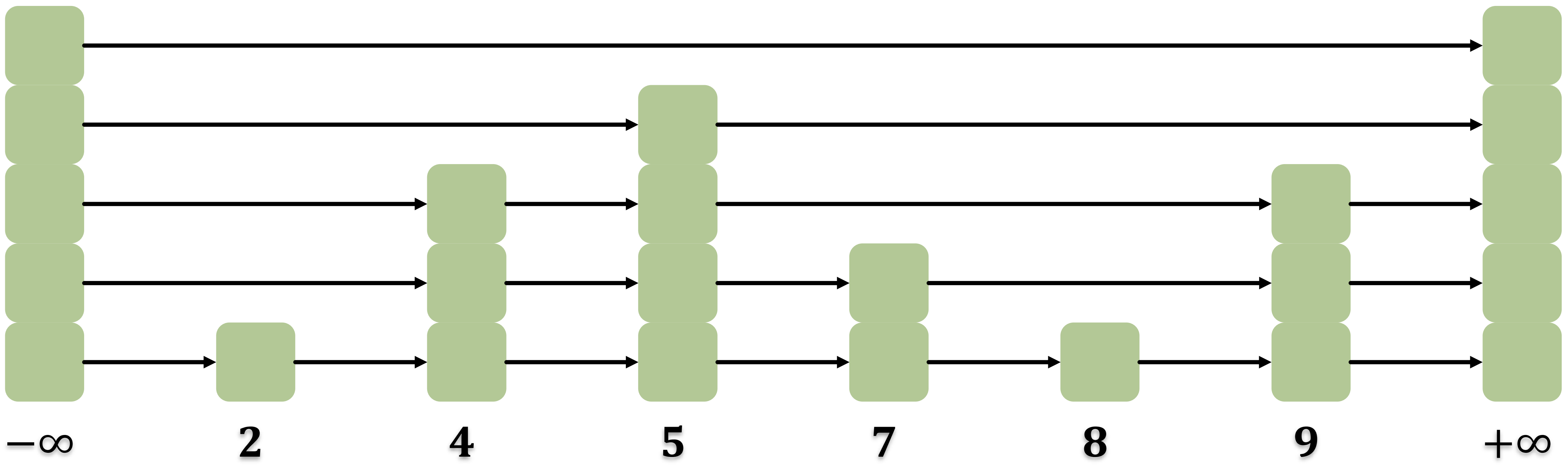
How about balanced trees?

- **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)**

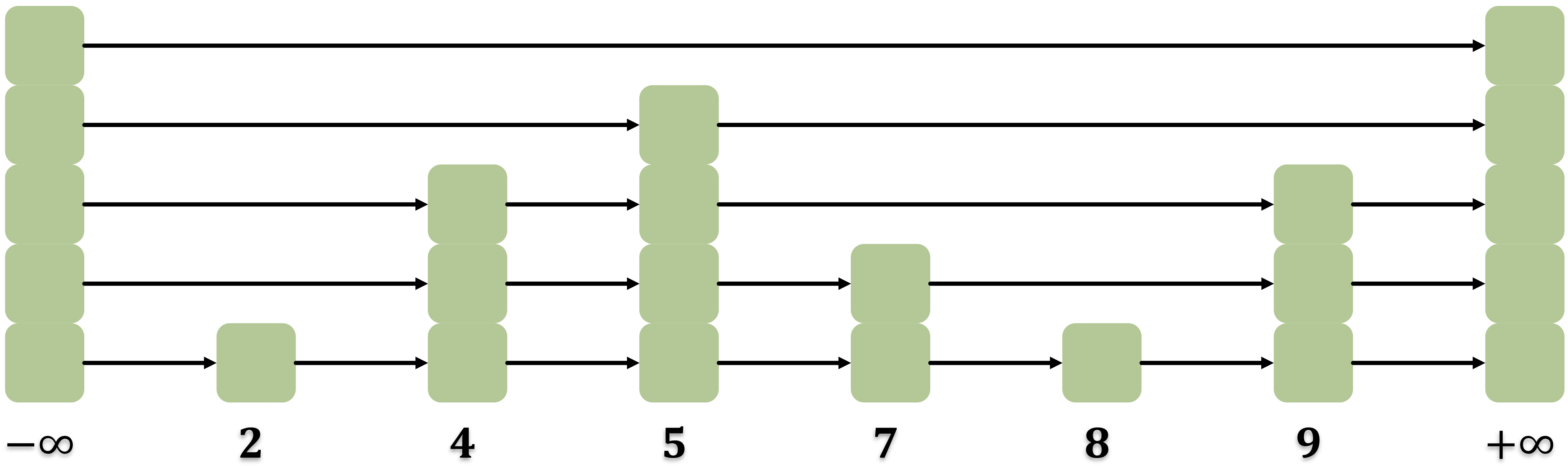
Skip lists

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



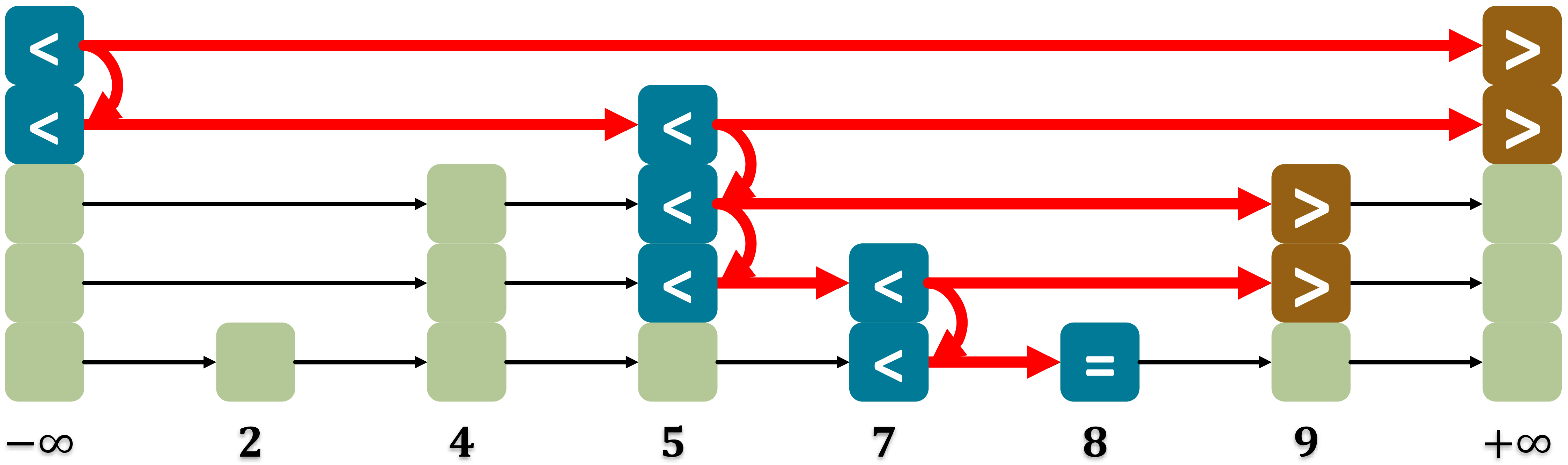
Skip list property

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



Searching

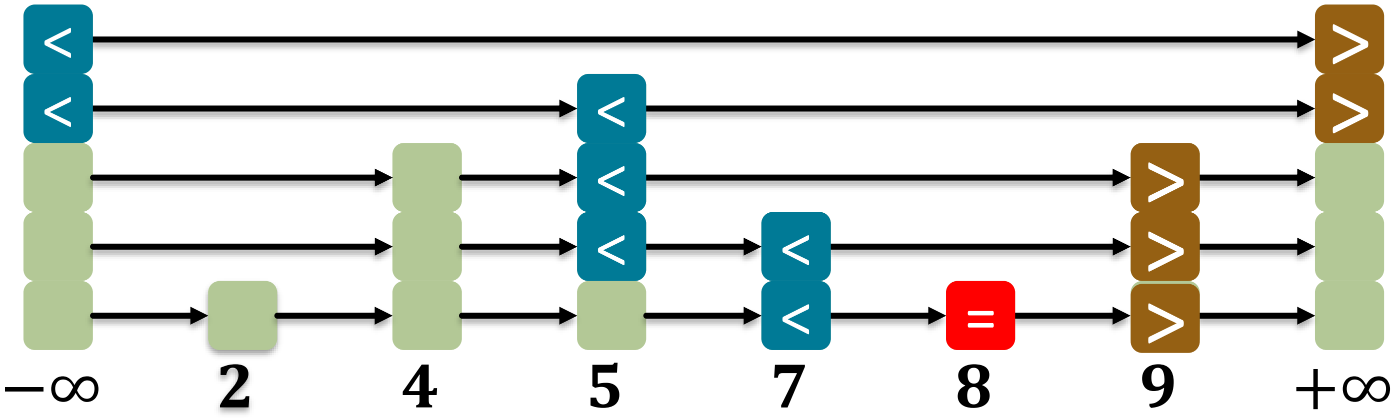
- Logarithmic search (with high probability)
- Example: Search for 8



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)`

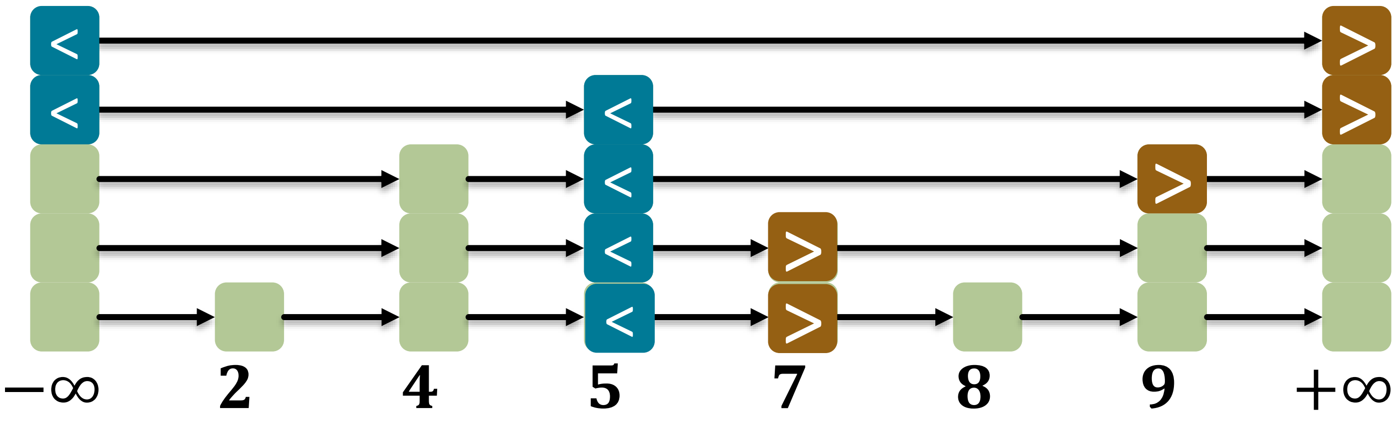
- `e.g., x = 8`
- `returns 0`



Sequential find

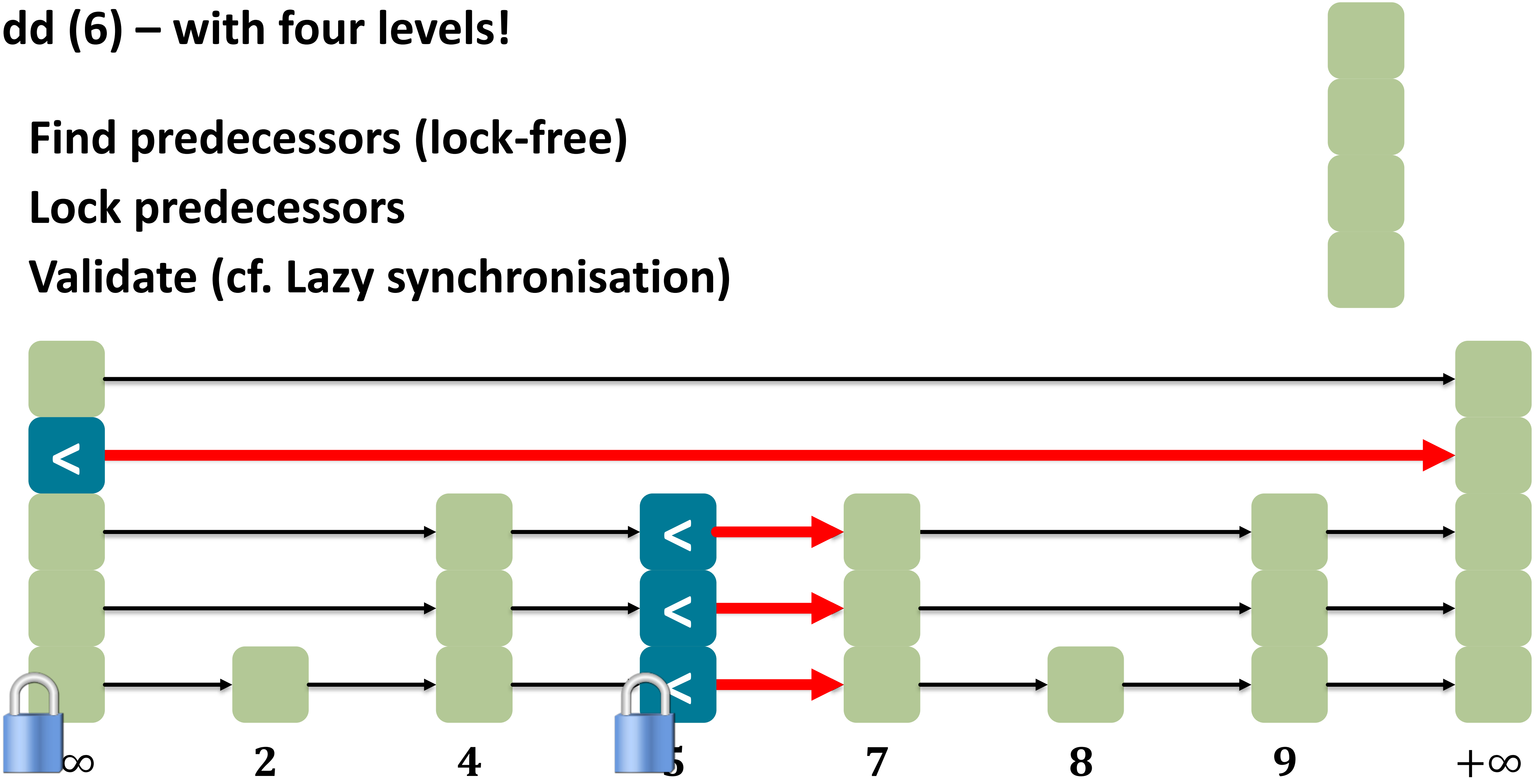
- // find node with value x
- // return -1 if not found, node level, succ, and pre otherwise
- // pre = array of predecessor node for all levels
- // succ = array of successor node for all levels
- **int find(T x, Node<T>[] pre, Node<T>[] succ)**

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



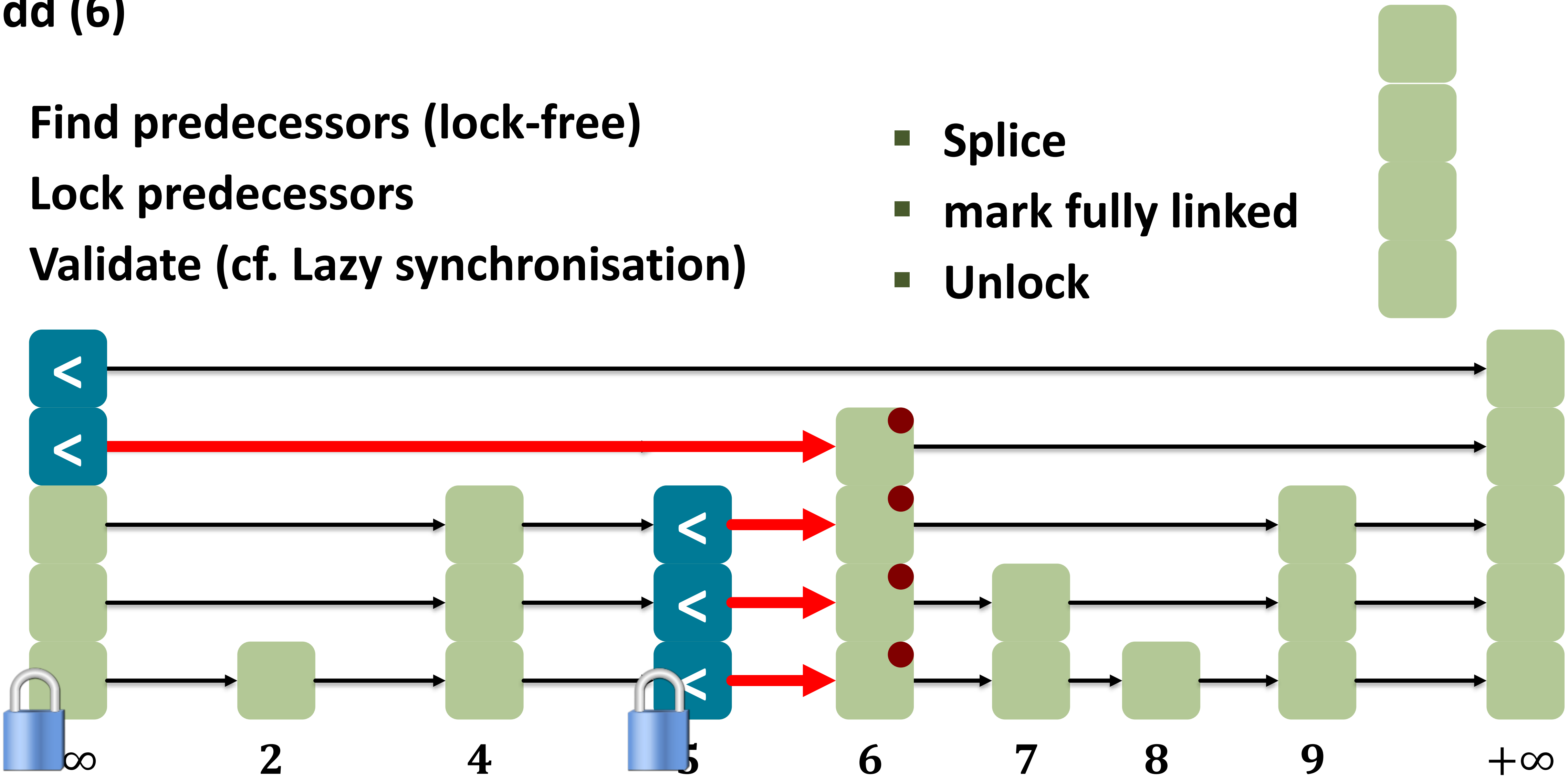
add (6) – with four levels!

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



add (6)

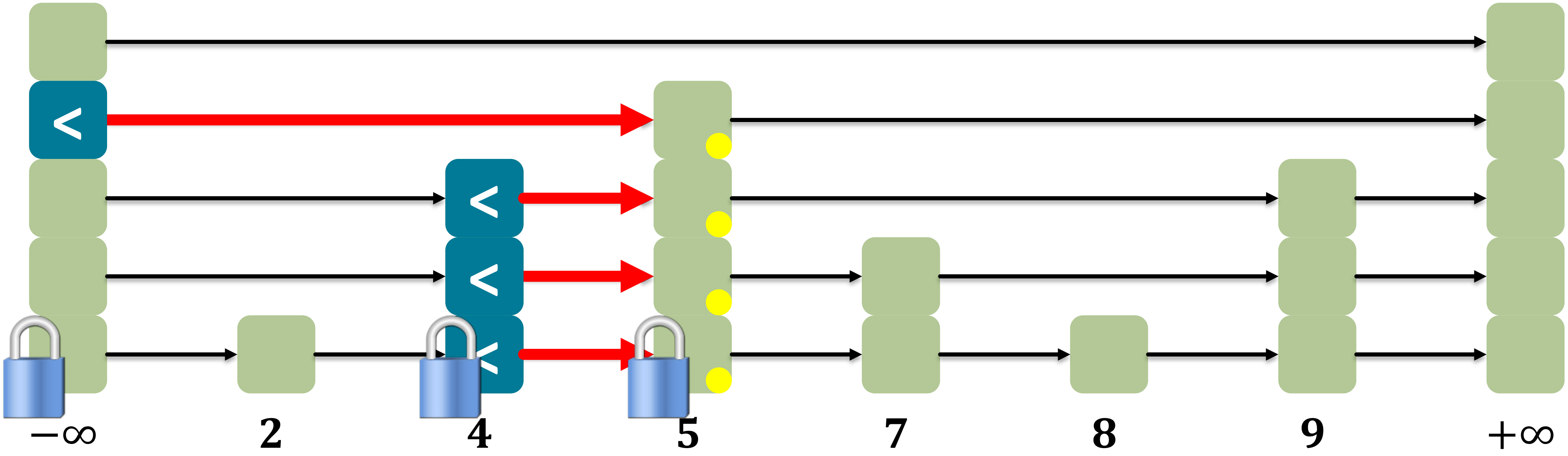
- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy synchronisation)
- Splice
- mark fully linked
- Unlock



remove(5)

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

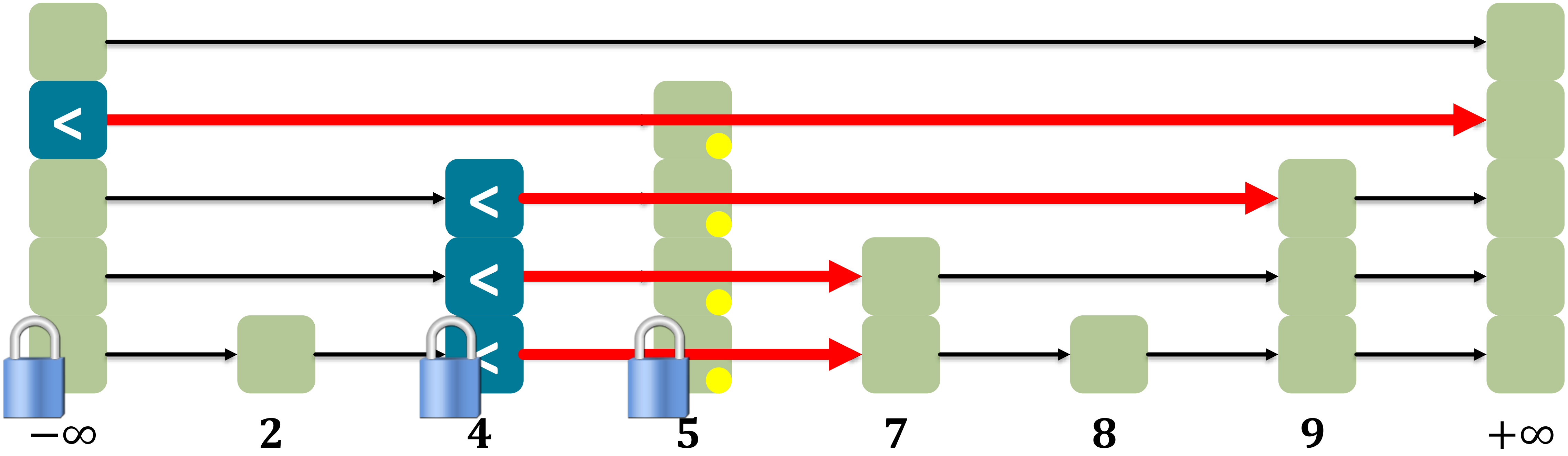
- Lock predecessors and validate



remove(5)

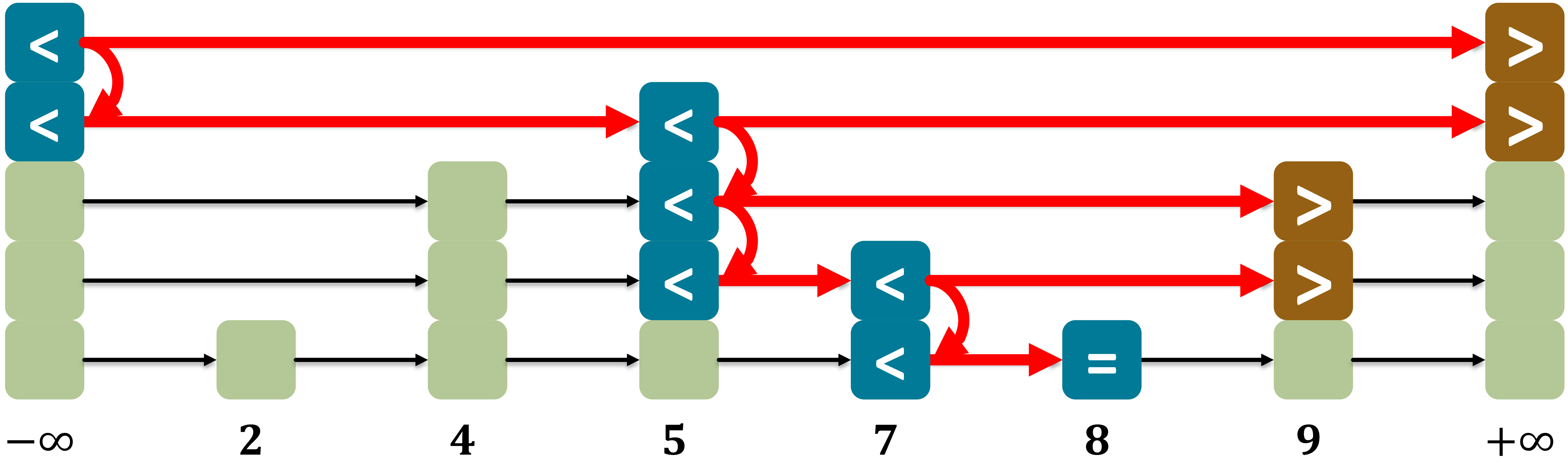
- find predecessors
- lock victim
- logically remove victim (mark)

- 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**

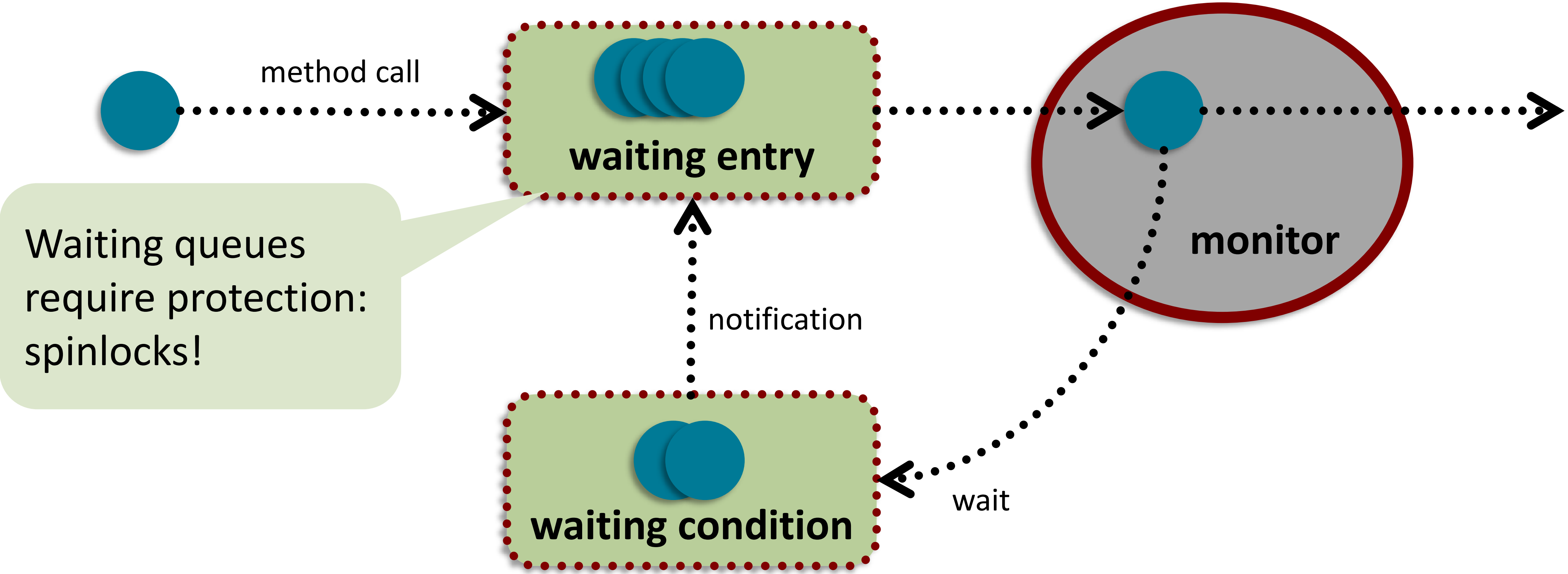
Literature:
-Herlihy Chapter 11.1 – 11.3
-Herlihy Chapter 9.8

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

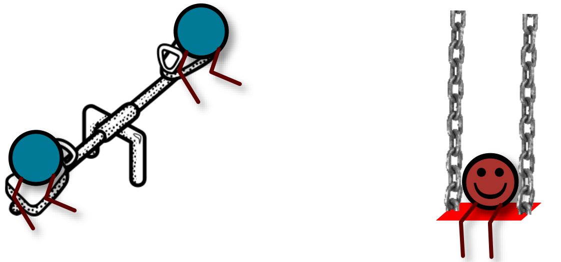
- **Locks that suspend the execution of threads while they wait. Semaphores, mutexes and monitors are typically implemented using a scheduled lock.**



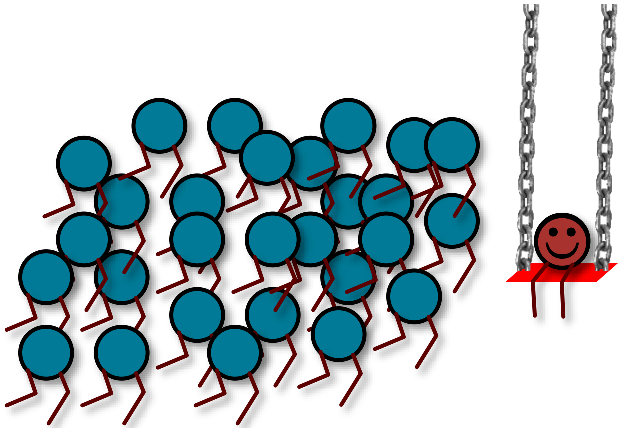
Locks with waiting/scheduling

- **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.

implies

- **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 <u>cannot</u> 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 whether CAS succeeded)

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)

Why not “guarantees”?

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.

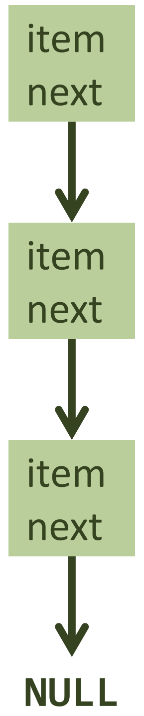
Sidenotes:

- **maybe transactional memory will become competitive at some point**
- **LL/SC or variants thereof may give stronger semantics avoiding ABA**

Lock-Free Stack

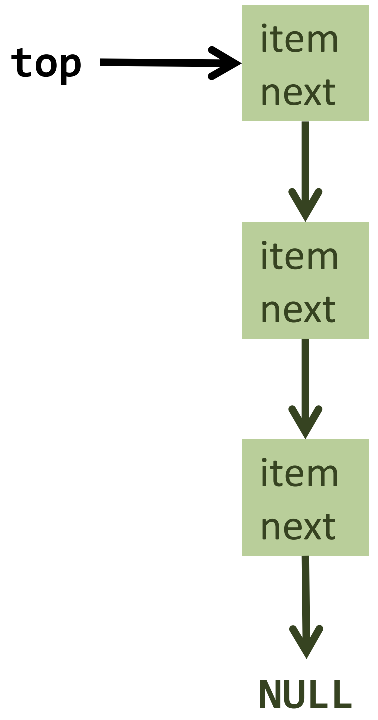
Stack Node

```
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;
   }

}
```



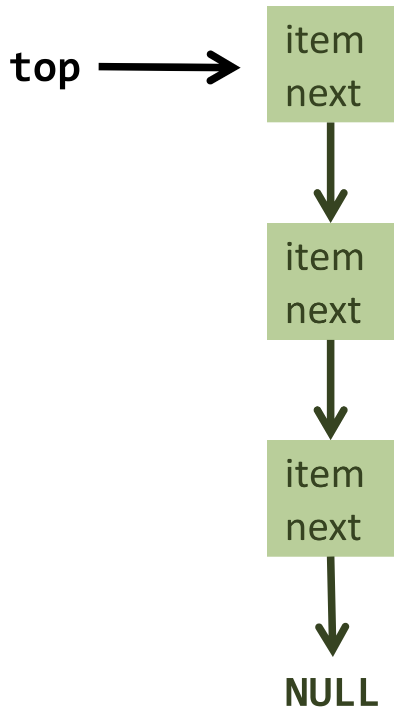
Blocking Stack

```
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;
   }
}
```



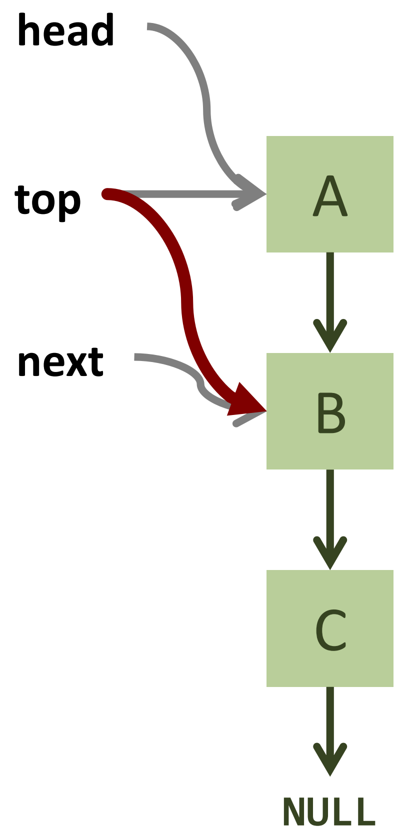
Non-blocking Stack

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

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



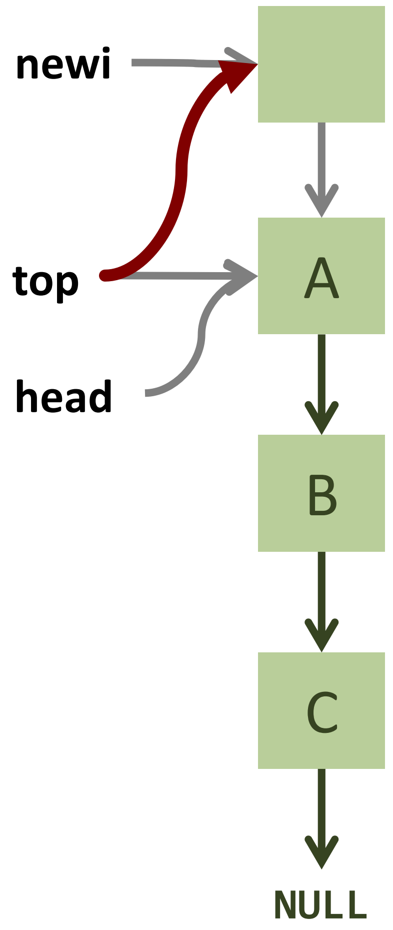
Pop

```
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?

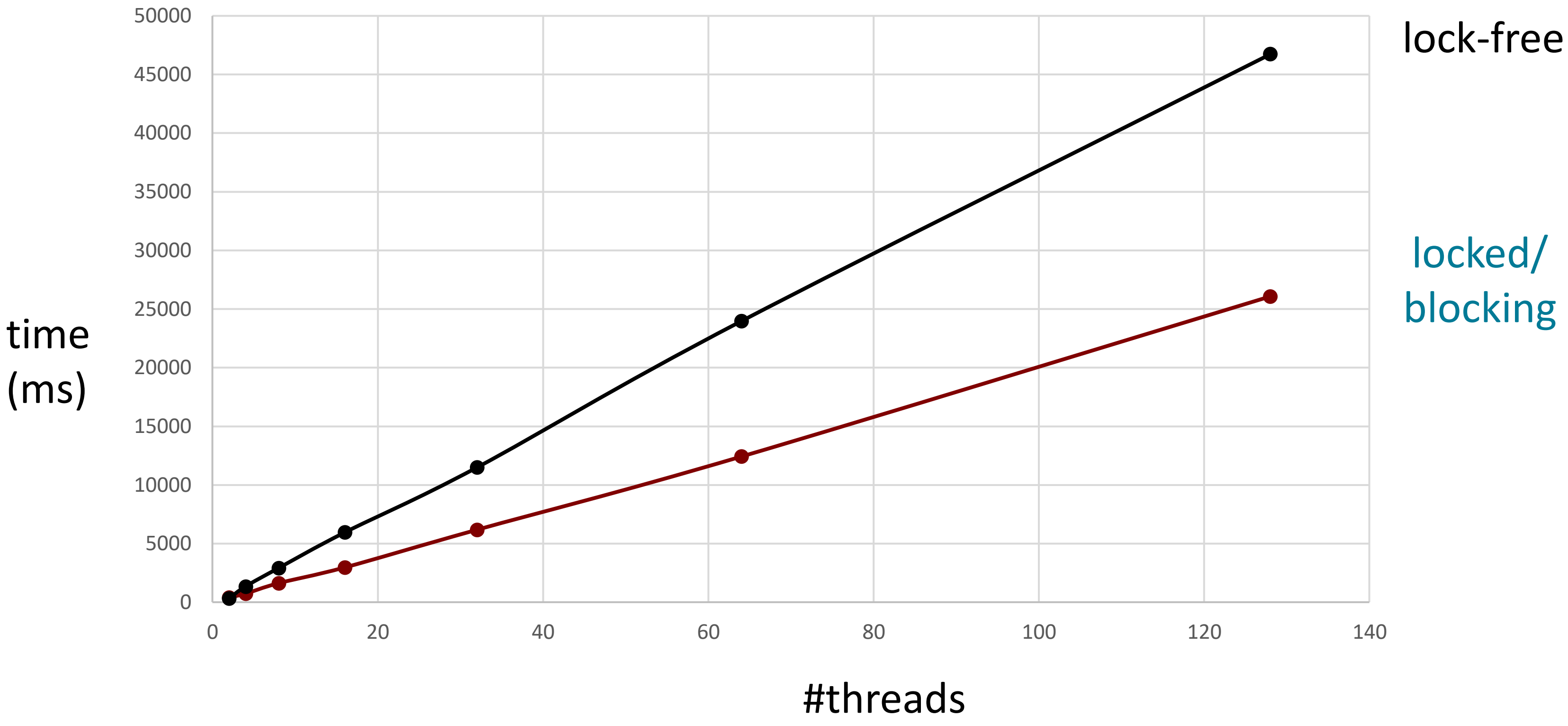
Lock-free programs are deadlock-free by design.

How about performance?

n threads

100,000 push/pop operations

10 times

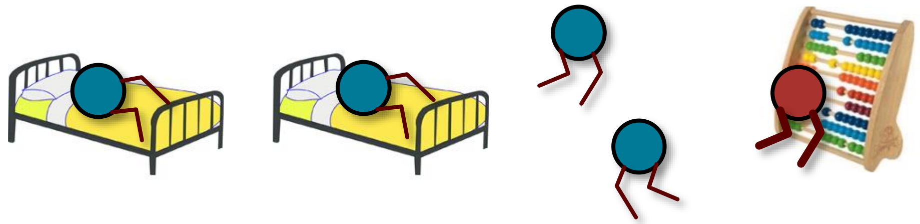


Performance

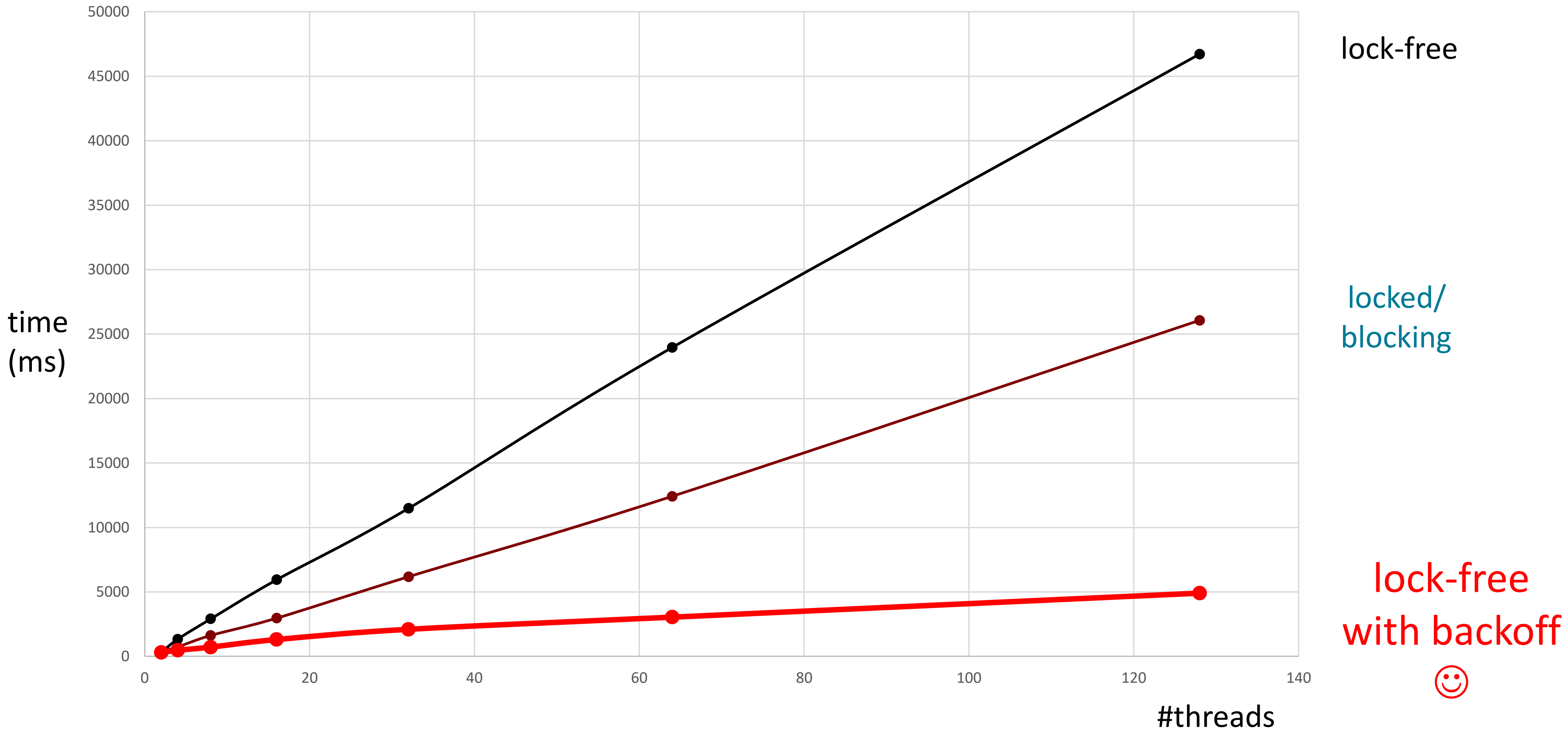
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

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

→ Backoff, again.



With backoff



LOCK FREE LIST SET

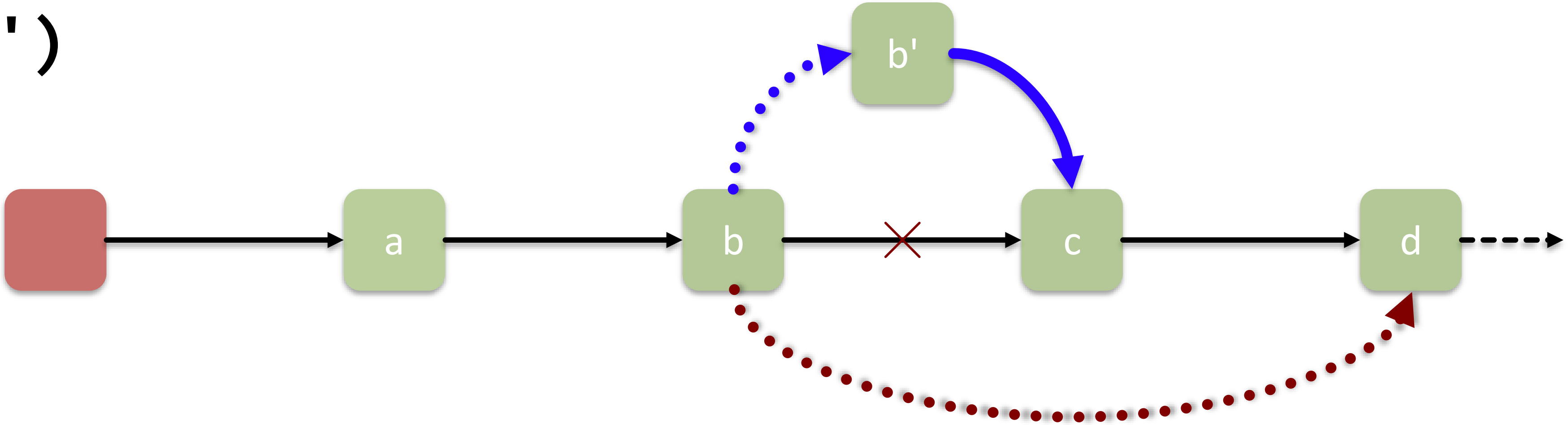
(NOT SKIP LIST!)

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

Does this work?

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

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



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

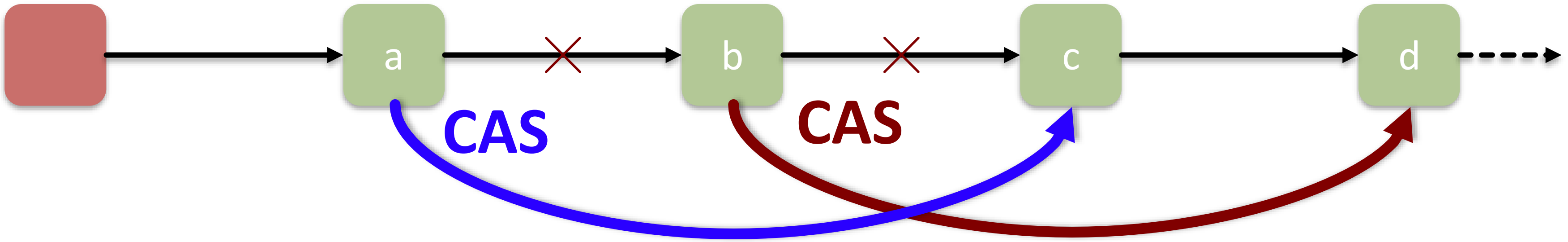
ok?

CAS decides who wins → this seems to work

So does this CAS approach work generally??

Another scenario

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



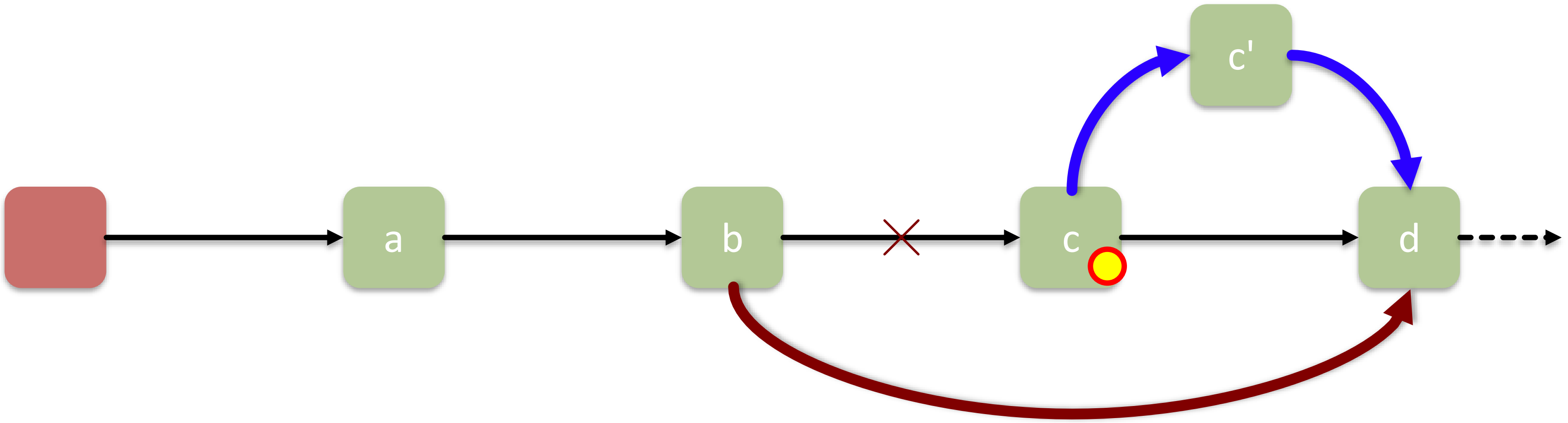
c not deleted! ☹

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

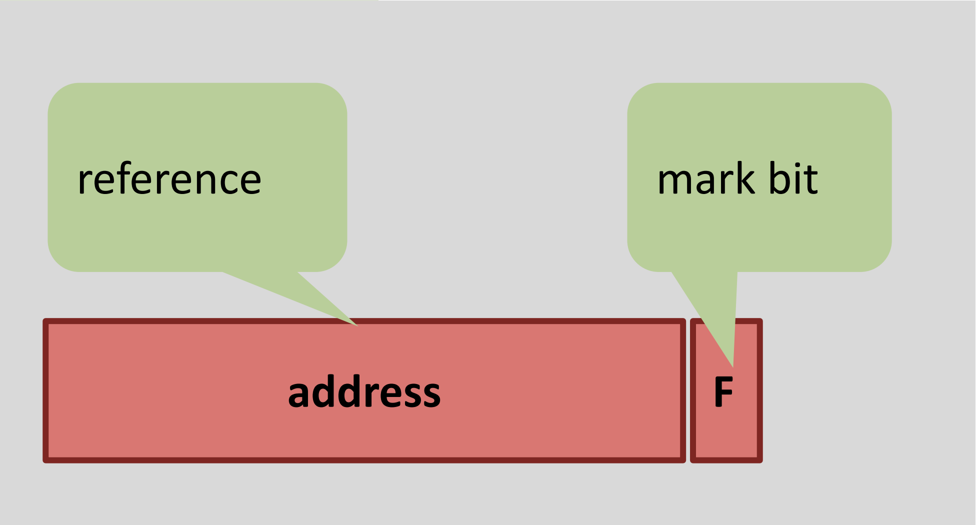
The difficulty that arises in this and many other problems is:

- 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

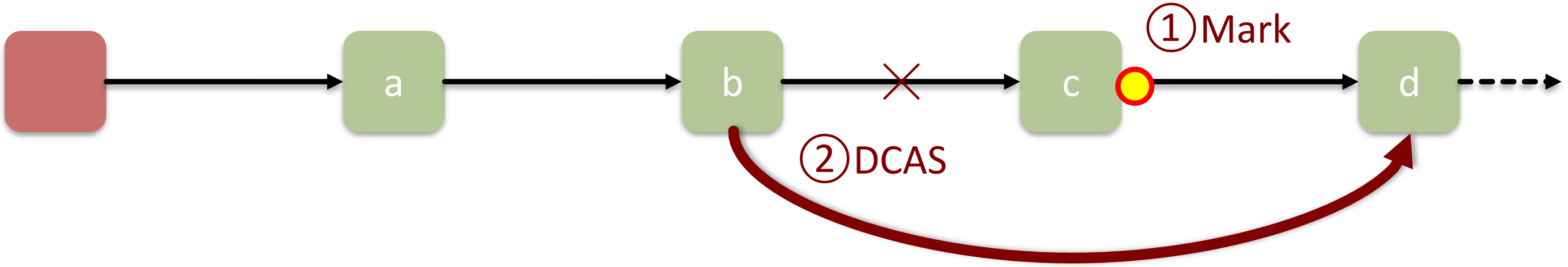
- **Atomically**
  - Swing reference *and*
  - Update flag
- **Remove in two steps**
  - Set mark bit in next field
  - Redirect predecessor's pointer

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]);

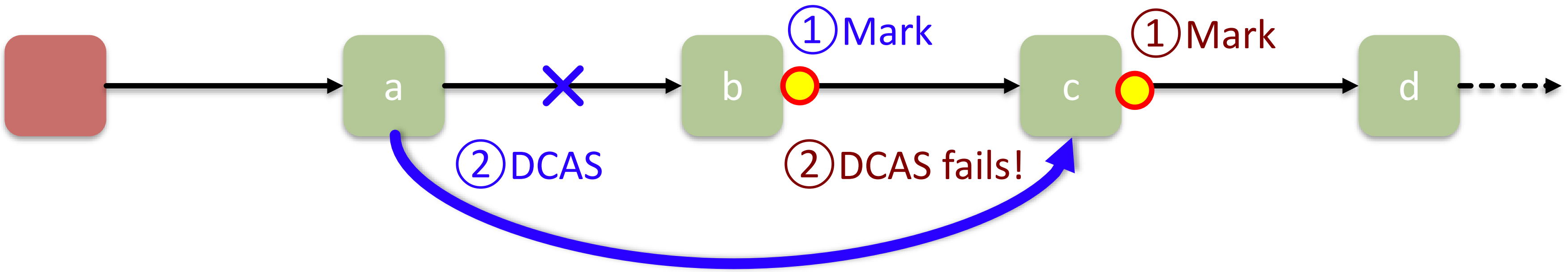


It helps!

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]);

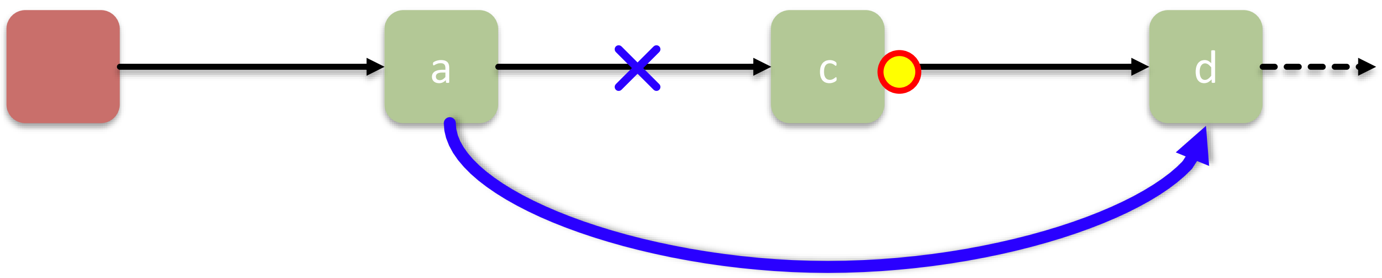
Traversing the list

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

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

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

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;
      }
   }
}
```

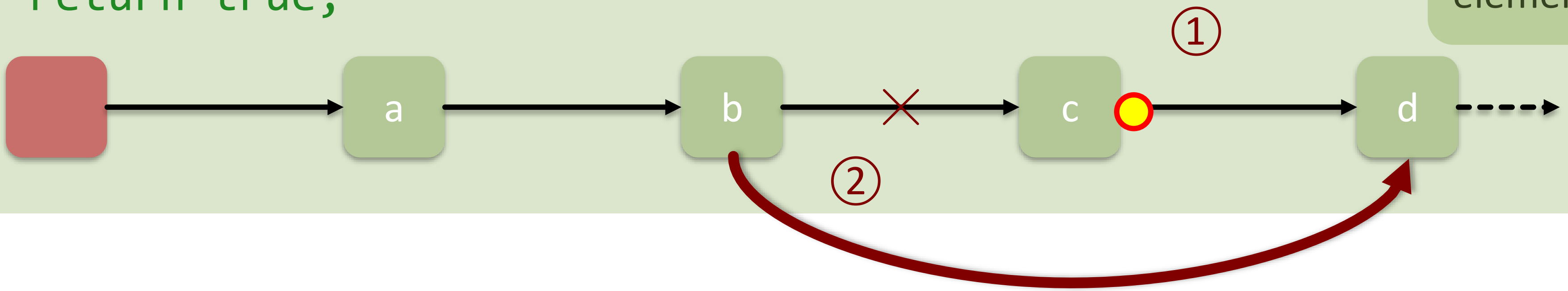
Find element and prev element from key

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

```java
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 ☺)