Multicore Programming

Lock-free data structures

15 Nov 2010

Peter Sewell Jaroslav Ševčík Tim Harris

What's wrong with locks?

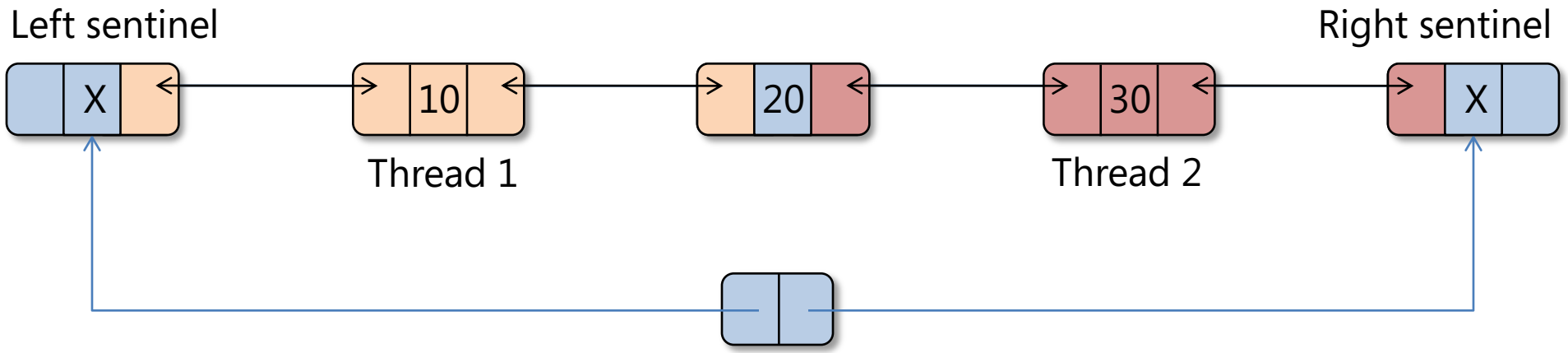
Lists without locks & linearizability

Lock-free progress

Hashtables

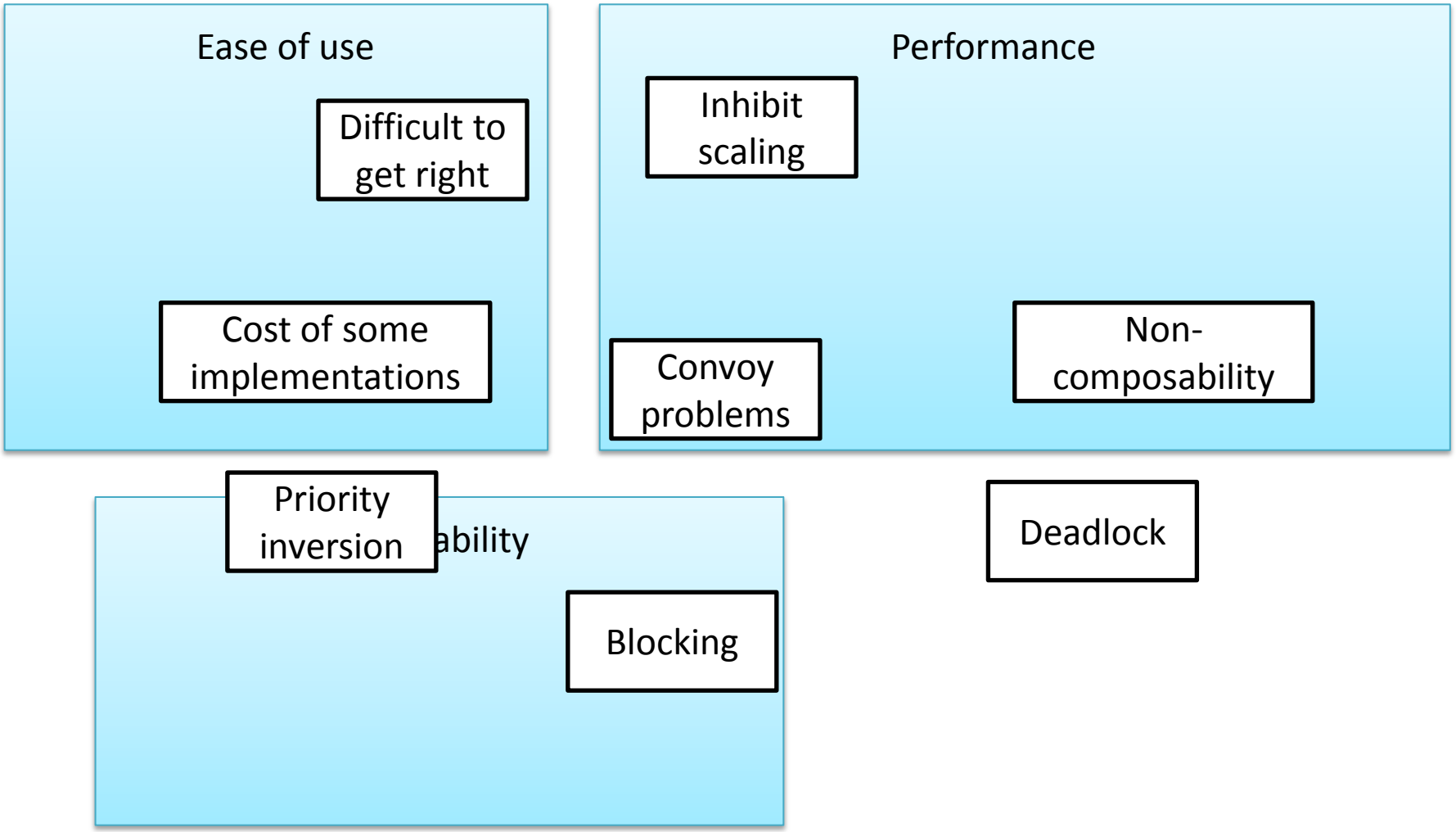
Skiplists

Example: double-ended queue



- "Do the right thing", even when used by multiple threads
- Support full set of push/pop on both ends
- Allow concurrency where possible

What do people say is wrong with locks?



What's wrong with locks?

Lists without locks & linearizability

Lock-free progress

Hashtables

Skiplists

What we're building

- A set of integers
- Represented by a sorted linked list

- find(int) -> bool
- insert(int) -> bool
- delete(int) -> bool

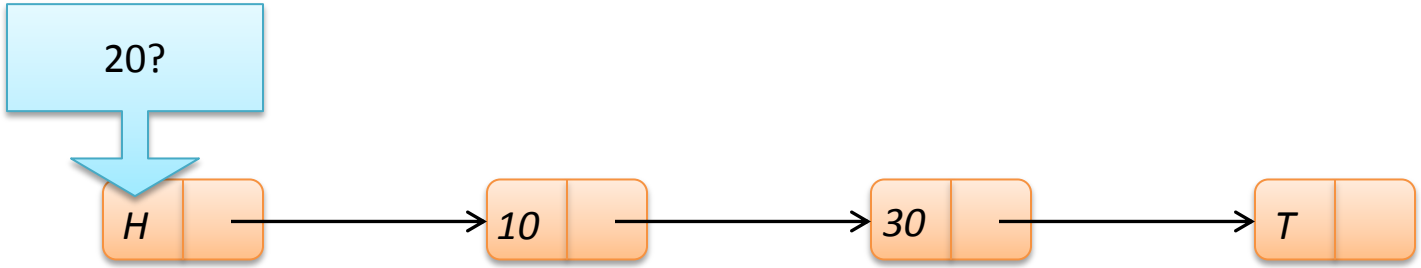
The building blocks

- read(addr) -> val
- write(addr, val)
- cas(addr, old-val, new-val) -> val

(I'll assume that memory is sequentially consistent, and ignore allocation / de-allocation for the moment)

Searching a sorted list

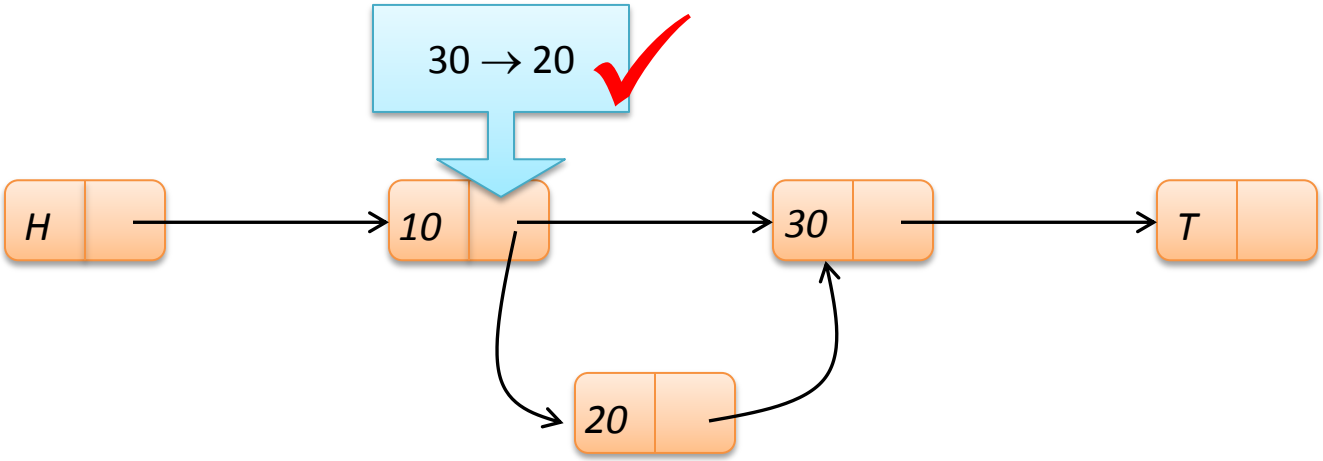
- find(20):



find(20) -> false

Inserting an item with CAS

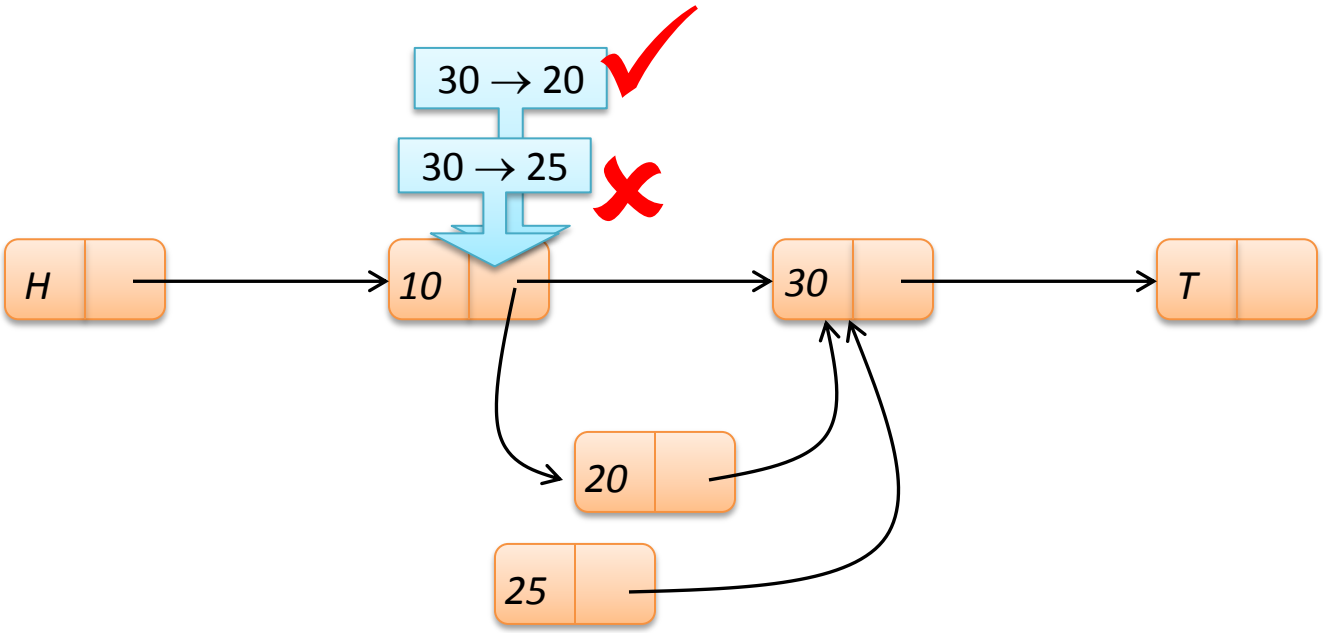
- insert(20):



insert(20) -> true

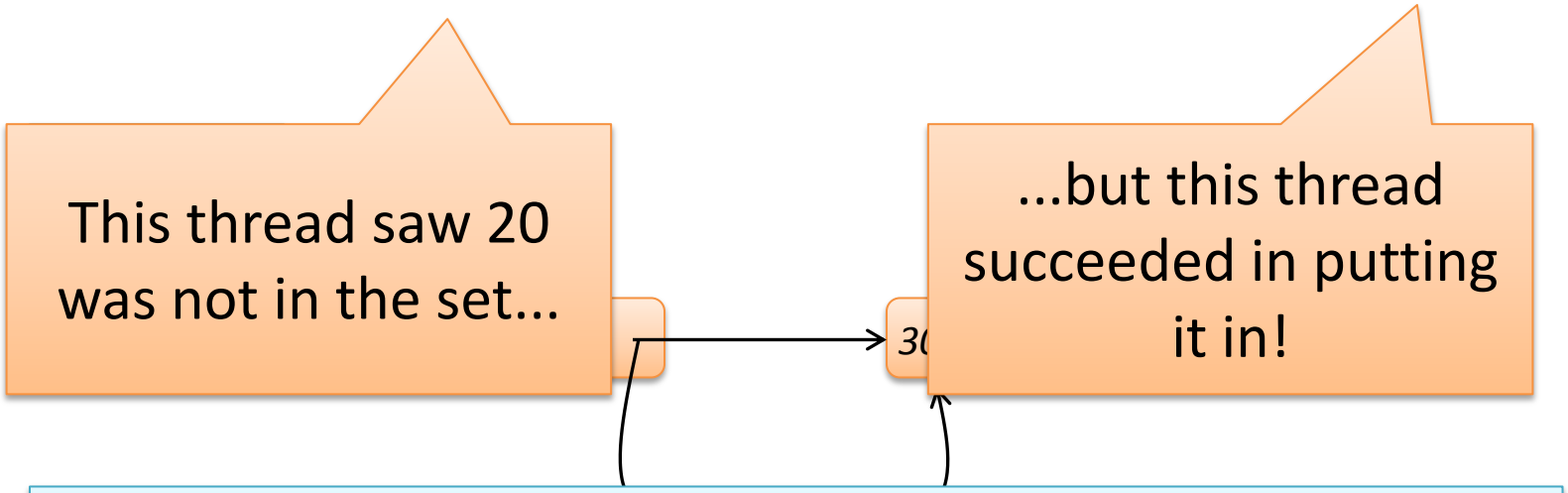
Inserting an item with CAS

- insert(20):
- insert(25):



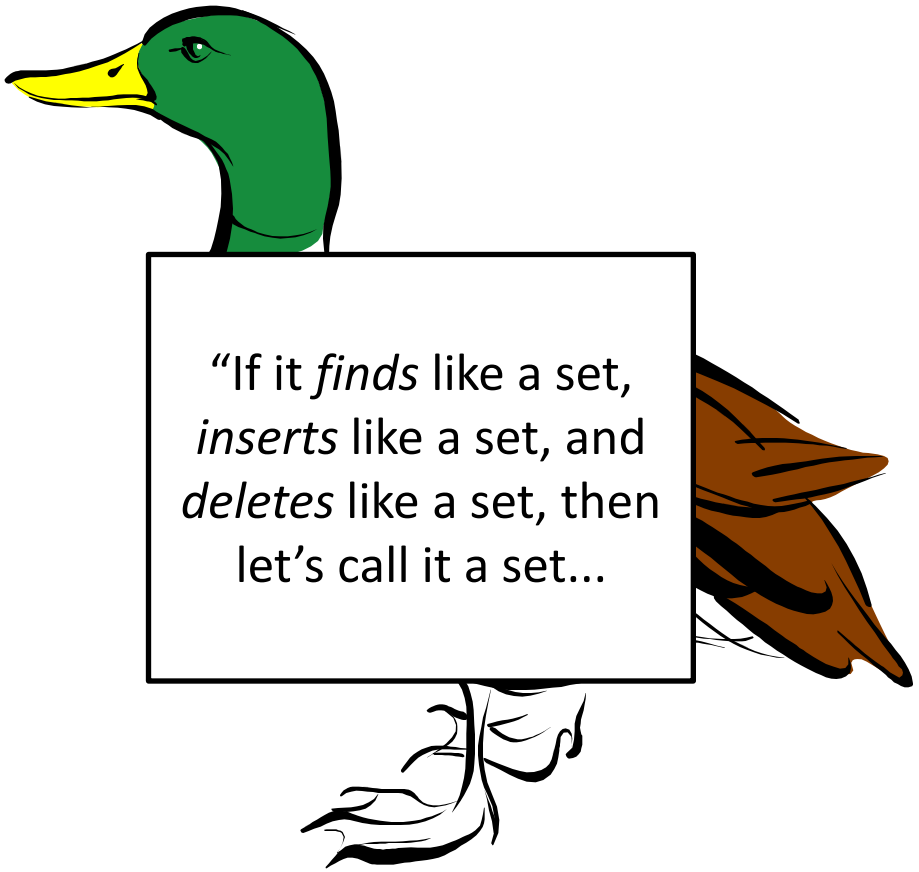
Searching and finding together

- find(20) -> false
- insert(20) -> true



- Is this a correct implementation of a set?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

Correctness criteria



Sequential specification

- Ig… th… cus o…

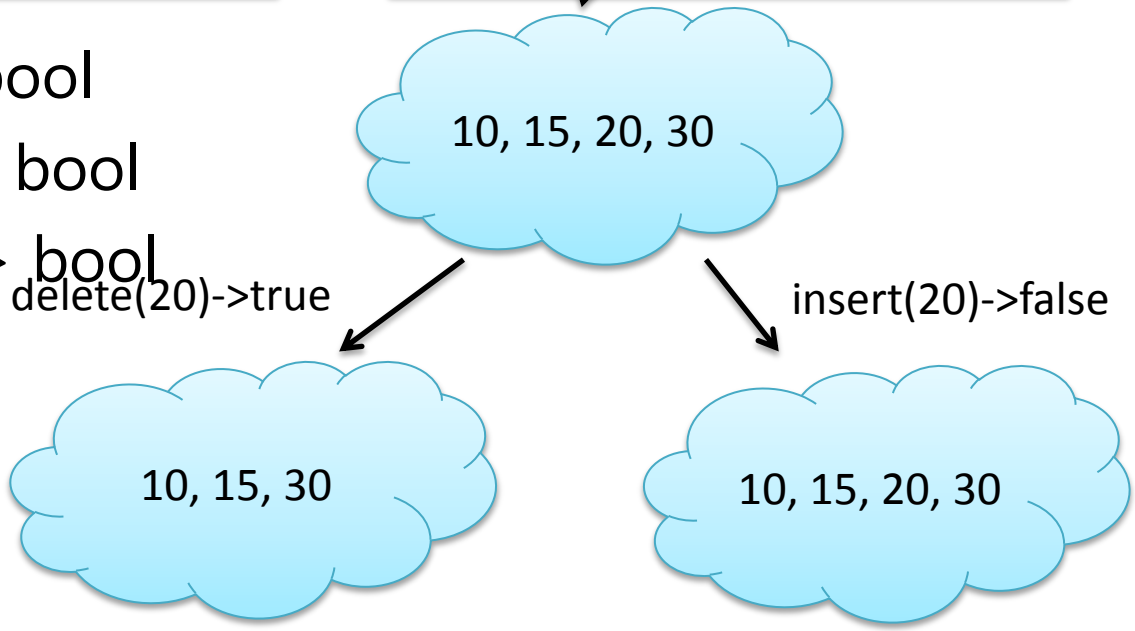
Sequential: we're only considering one operation on the set at a time

Specification: we're saying what a *set* does, not what a list does, or how it looks in memory

find(int) -> bool

insert(int) -> bool

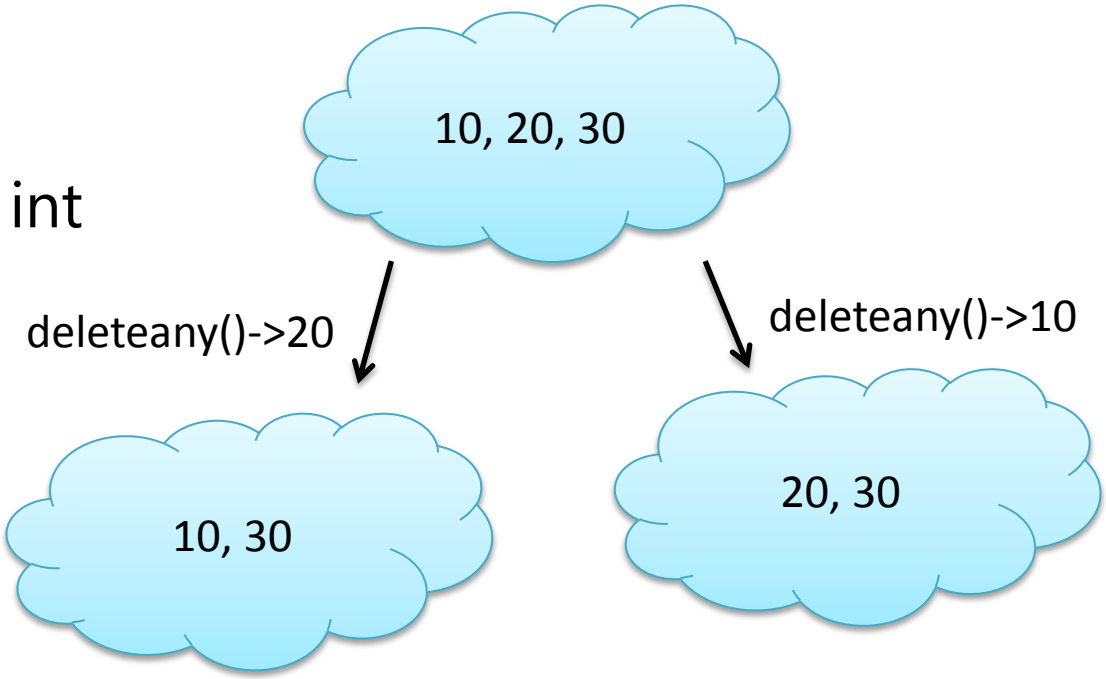
delete(int) -> bool



Sequential specification

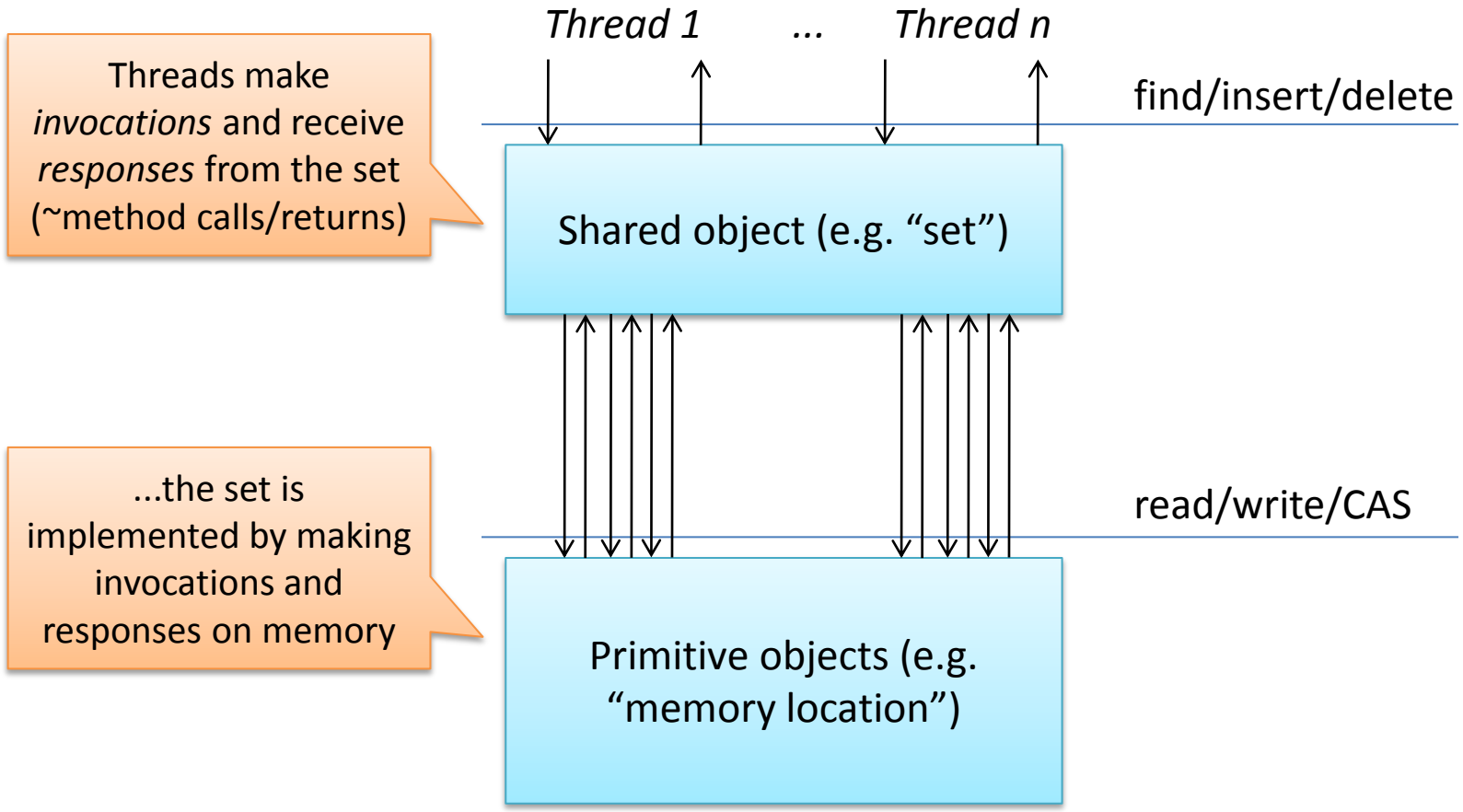
Let's add:

deleteany() -> int



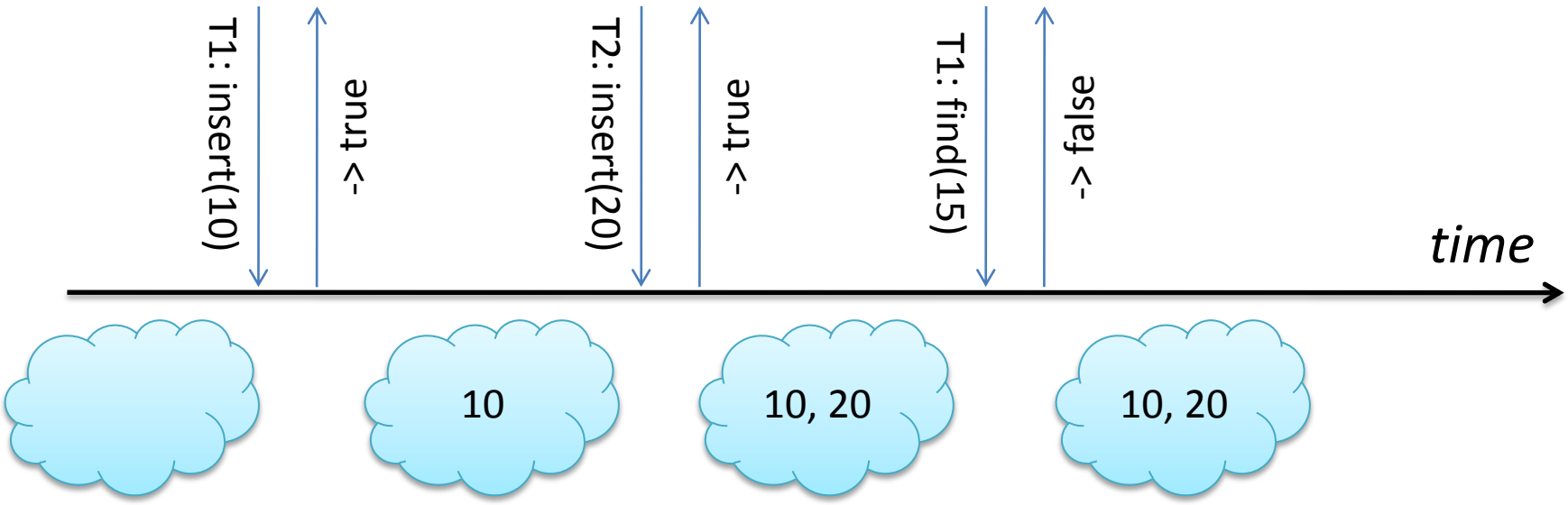
This is still a *sequential* spec... just not a *deterministic* one

System model



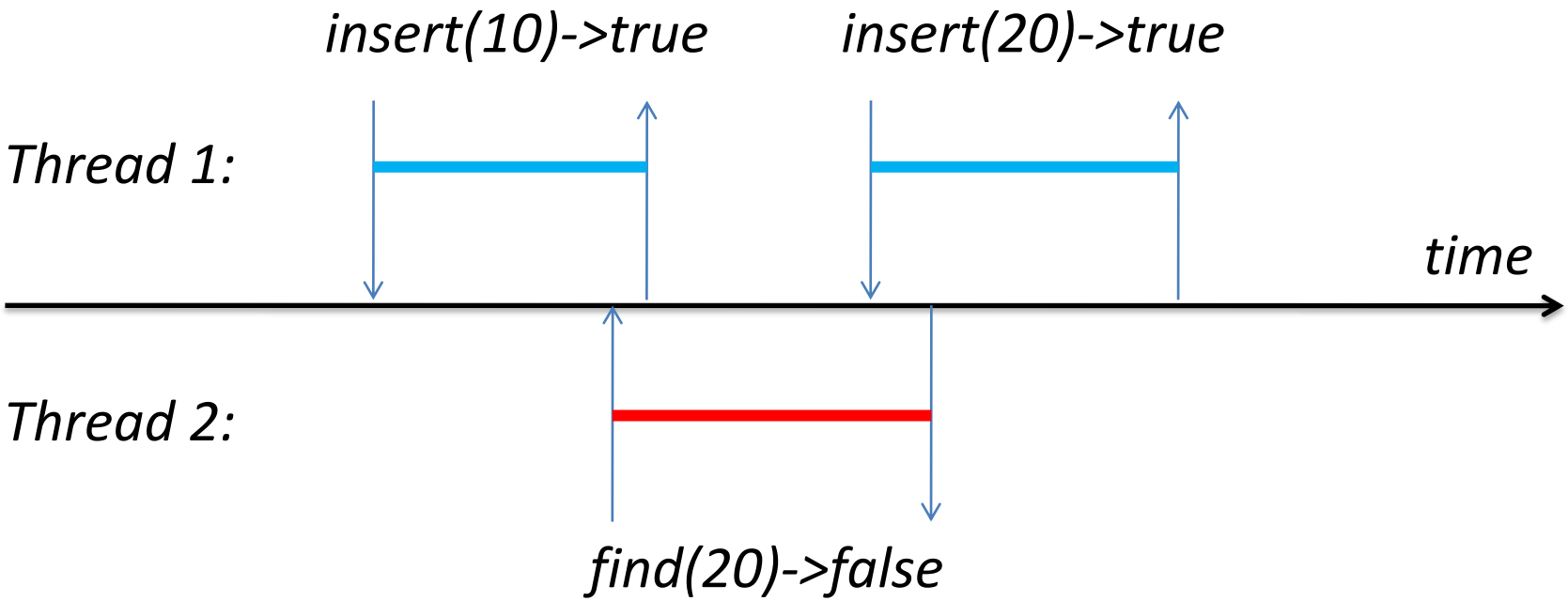
Sequential history

- No overlapping invocations:



Concurrent history

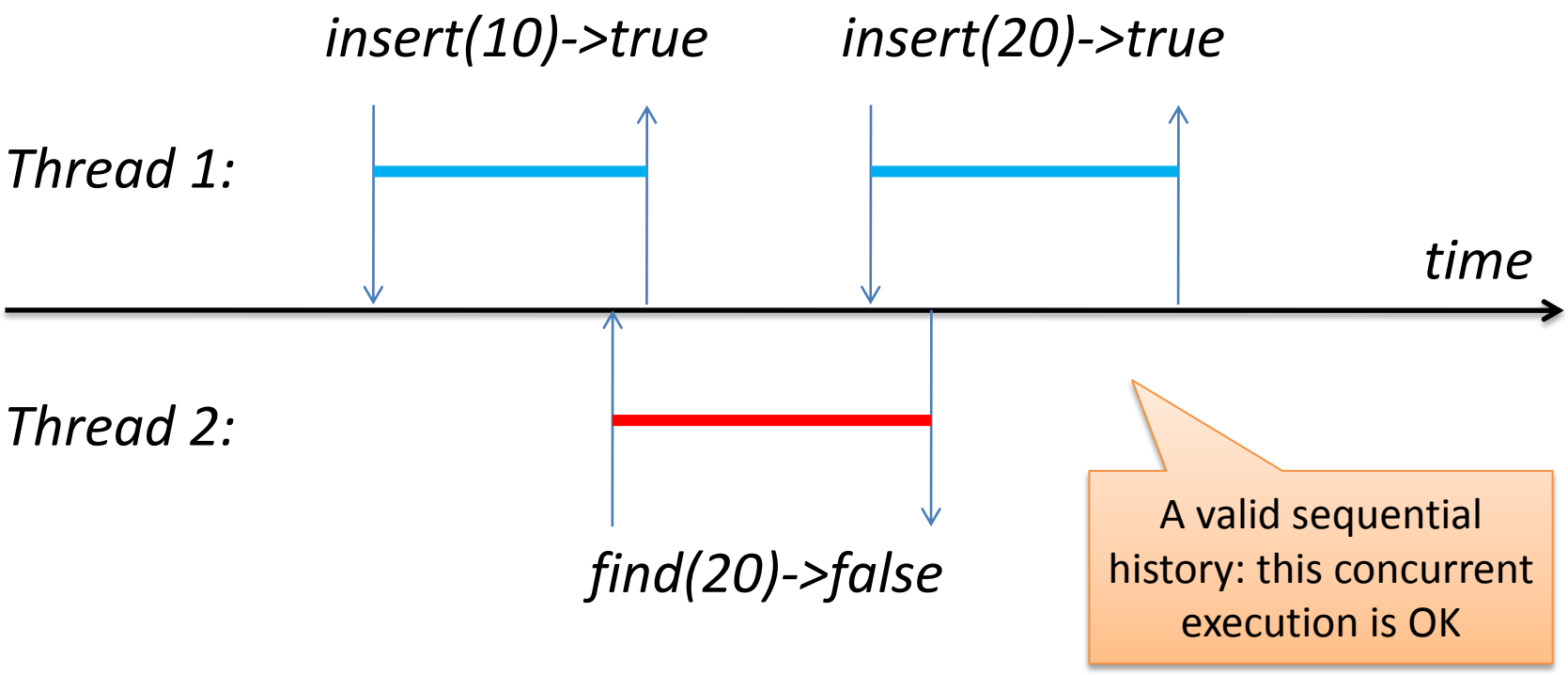
- Allow overlapping invocations:



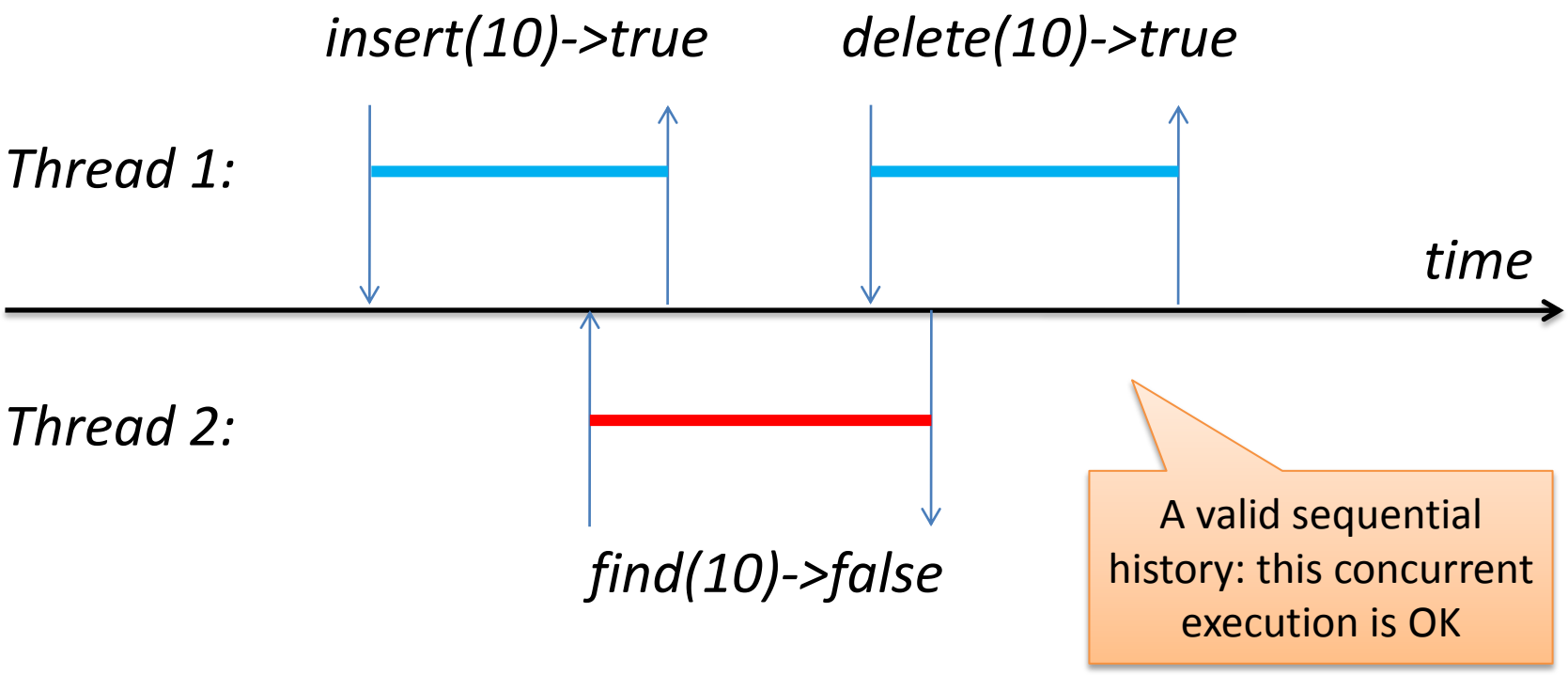
Linearizability

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?

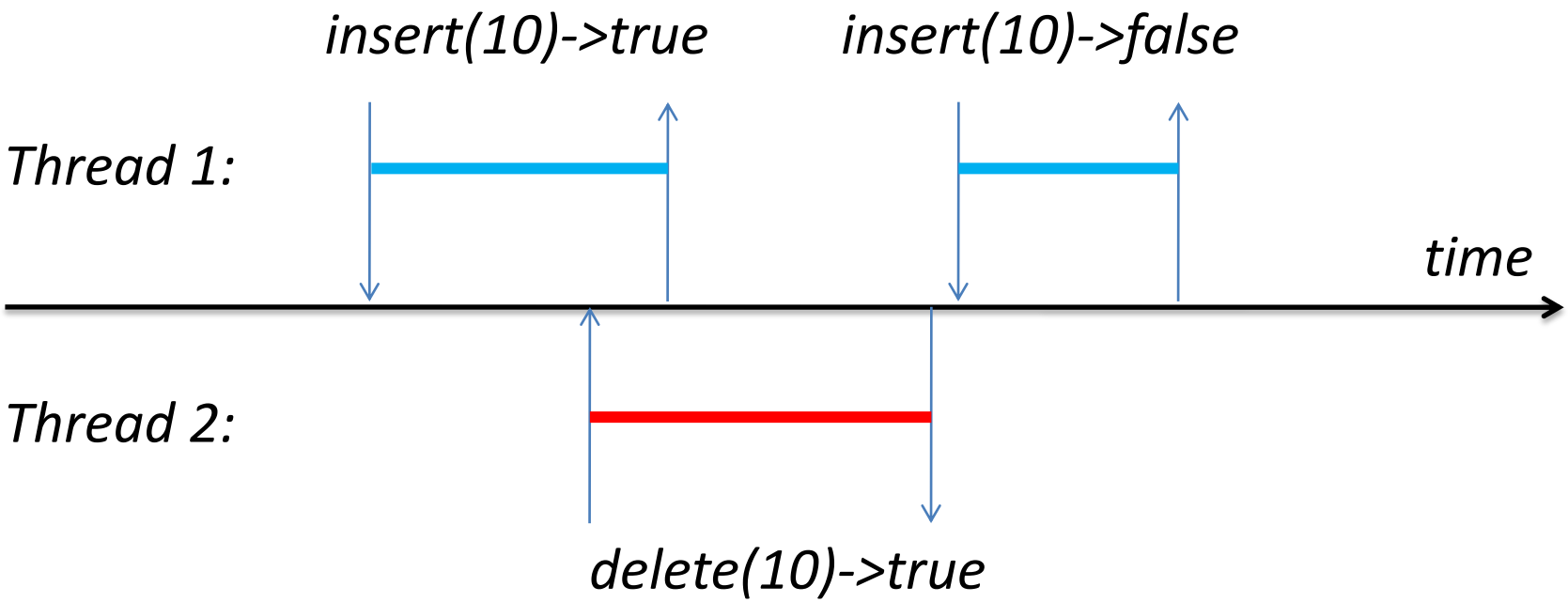
Example: linearizable



Example: linearizable

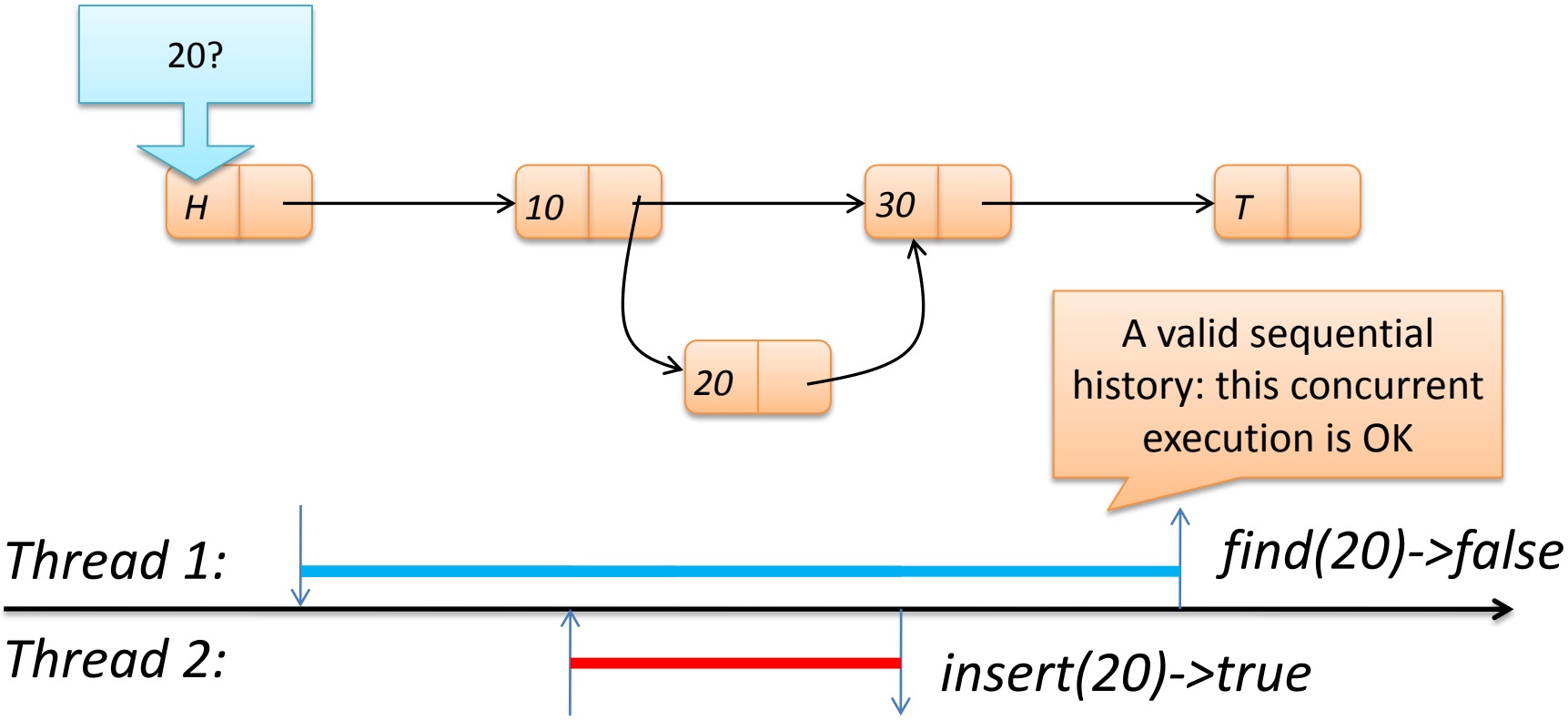


Example: not linearizable



Returning to our example

- find(20) -> false
- insert(20) -> true

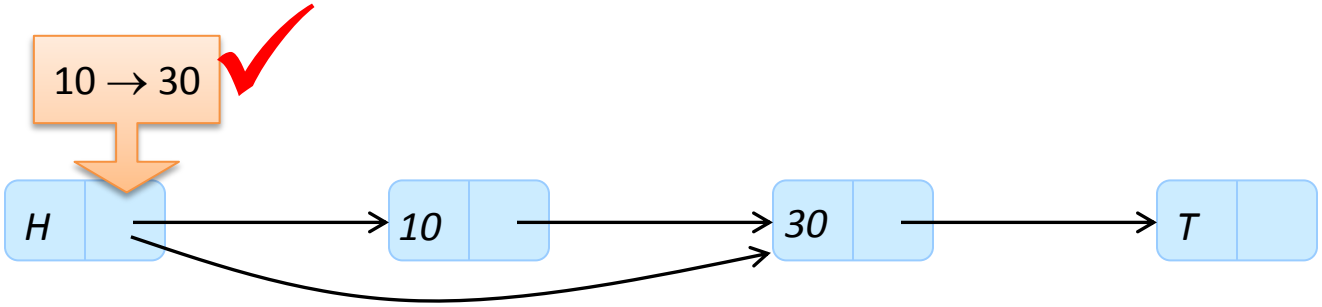


Recurring technique

- For updates:
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads:
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction

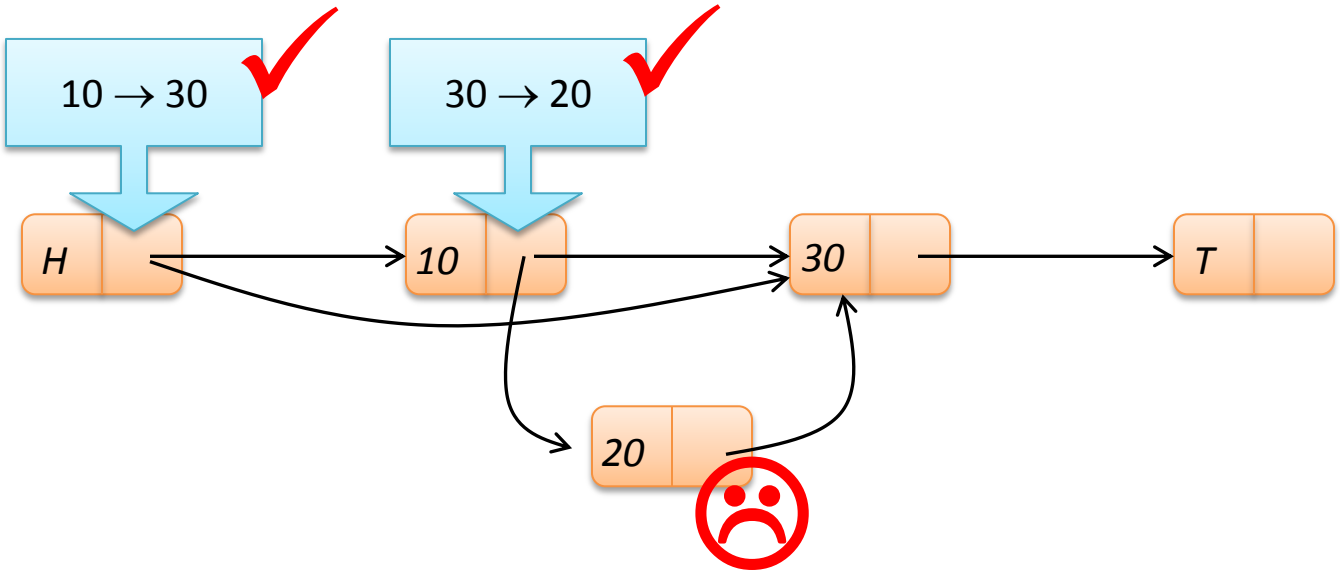
Adding "delete"

- First attempt: just use CAS
  delete(10):



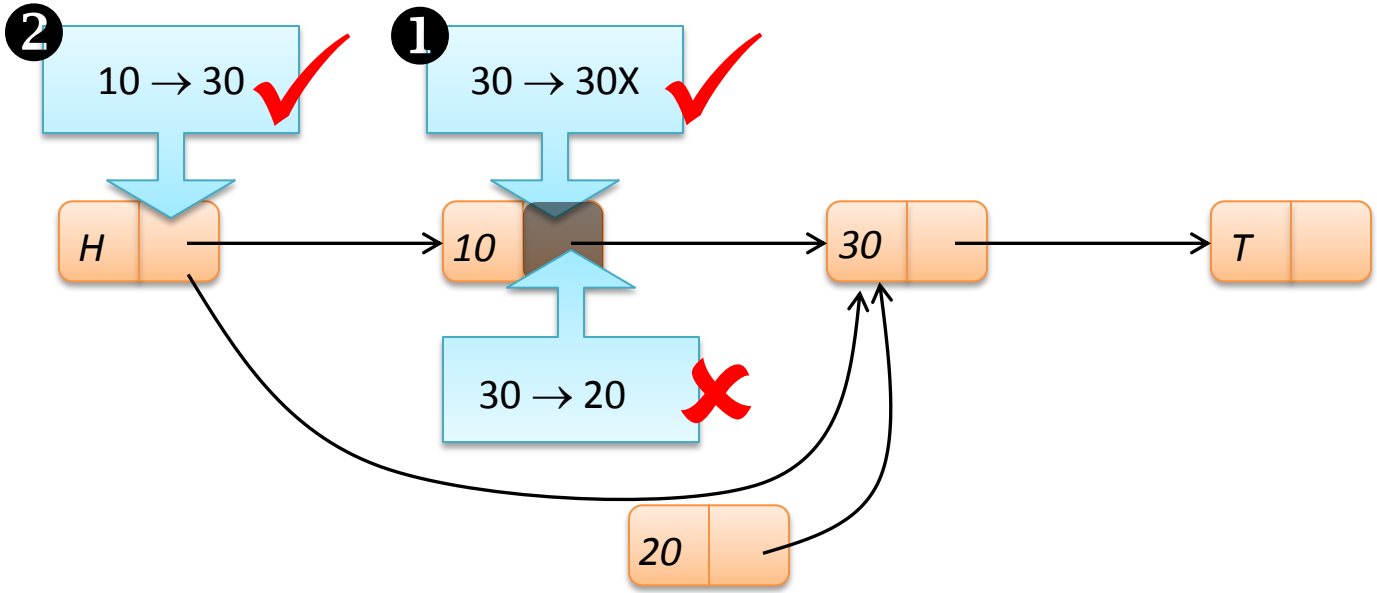
Delete and insert:

- delete(10) & insert(20):



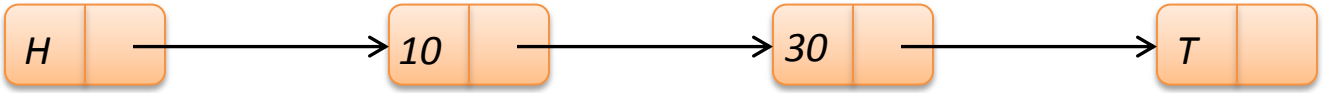
Logical vs physical deletion

- Use a ‘spare’ bit to indicate logically deleted nodes:

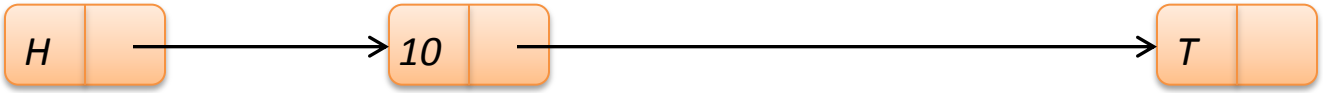


Delete-greater-than-or-equal

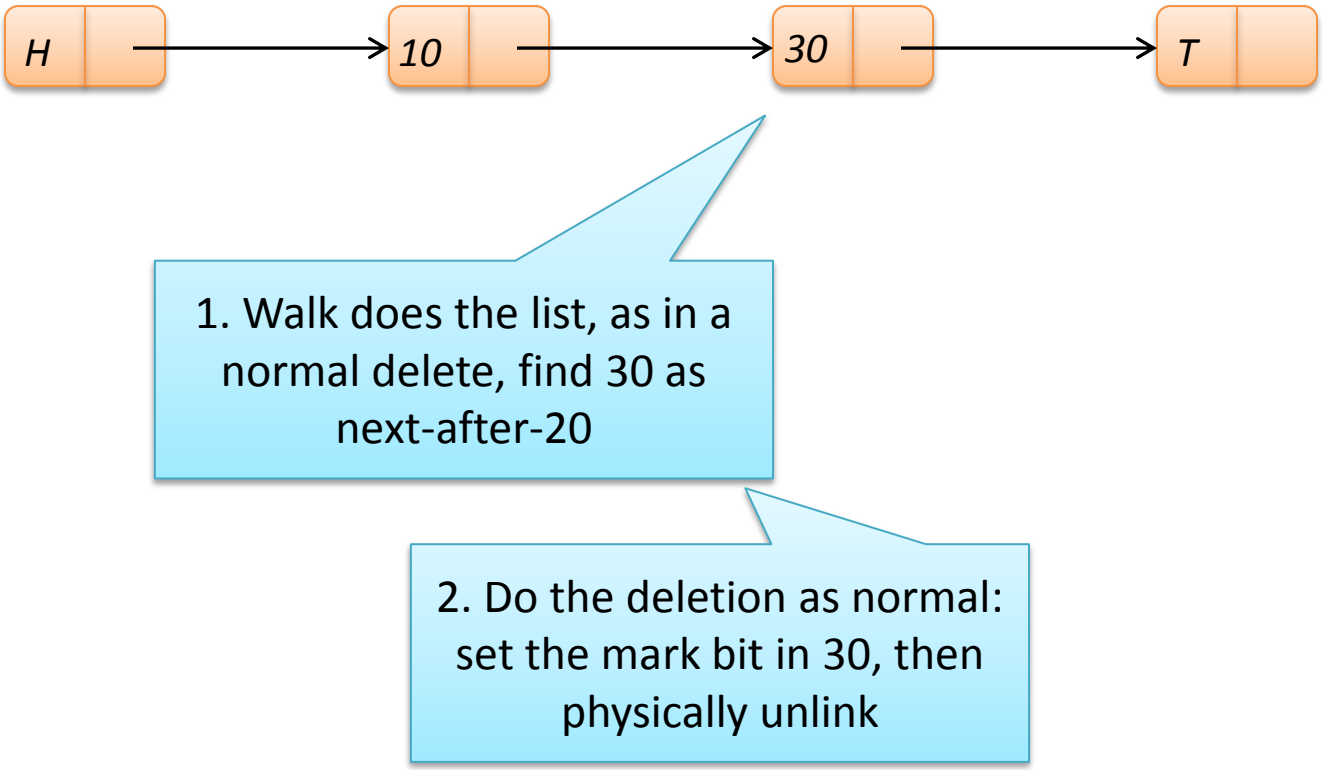
- DeleteGE(int x) -> int
  - Remove “x”, or next element above “x”



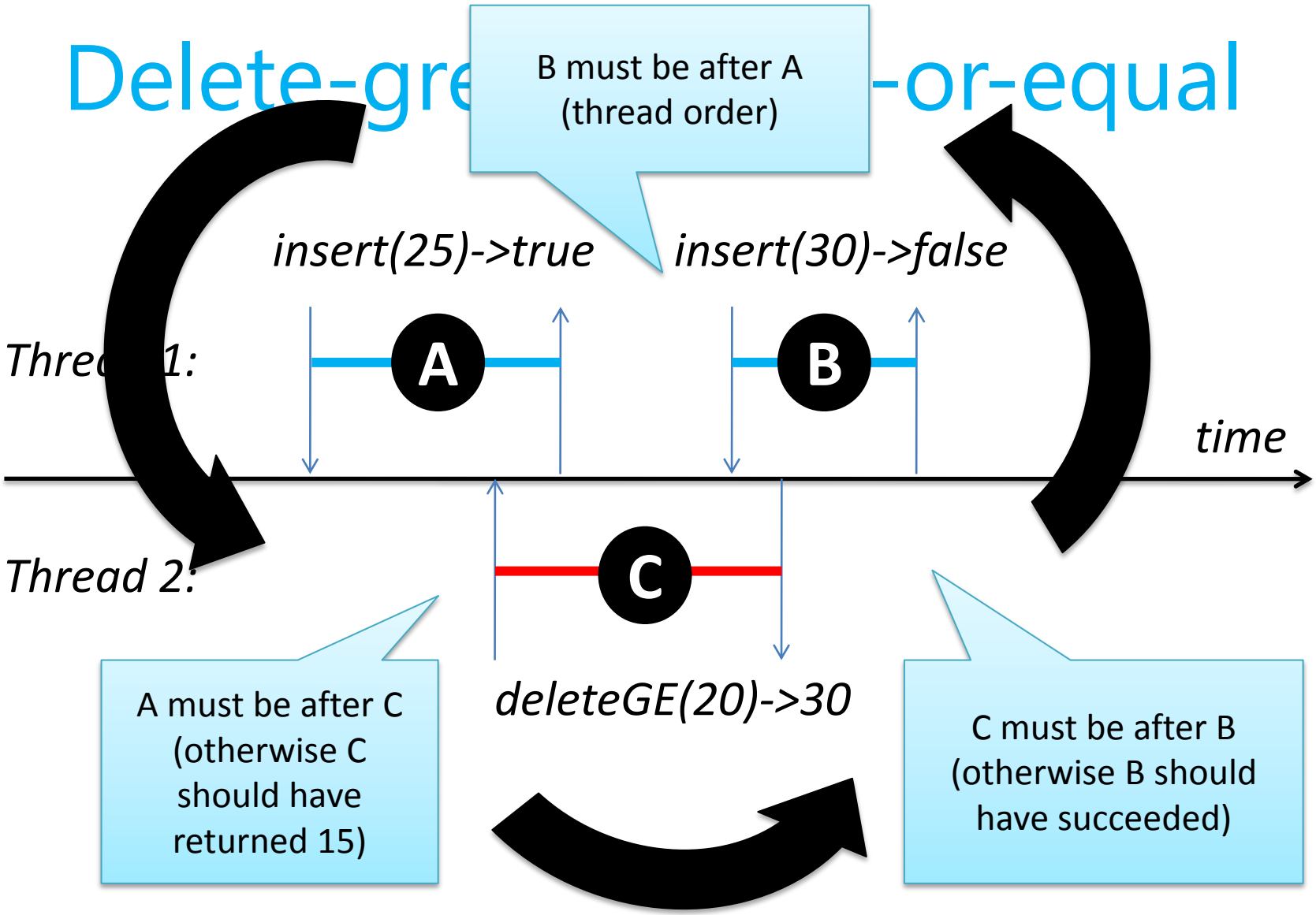
- DeleteGE(20) -> 30



Does this work: DeleteGE(20)



Delete-gre -or-equal



How to realise this is wrong

- See operation which determines result
- Consider a delay at that point
- Is the result still valid?
  - Delayed read: is the memory still accessible (more of this next week)
  - Delayed write: is the write still correct to perform?
  - Delayed CAS: does the value checked by the CAS determine the result?

What's wrong with locks?

Lists without locks & linearizability

Lock-free progress

Hashtables

Skiplists

Progress: is this a good "lock-free" list?

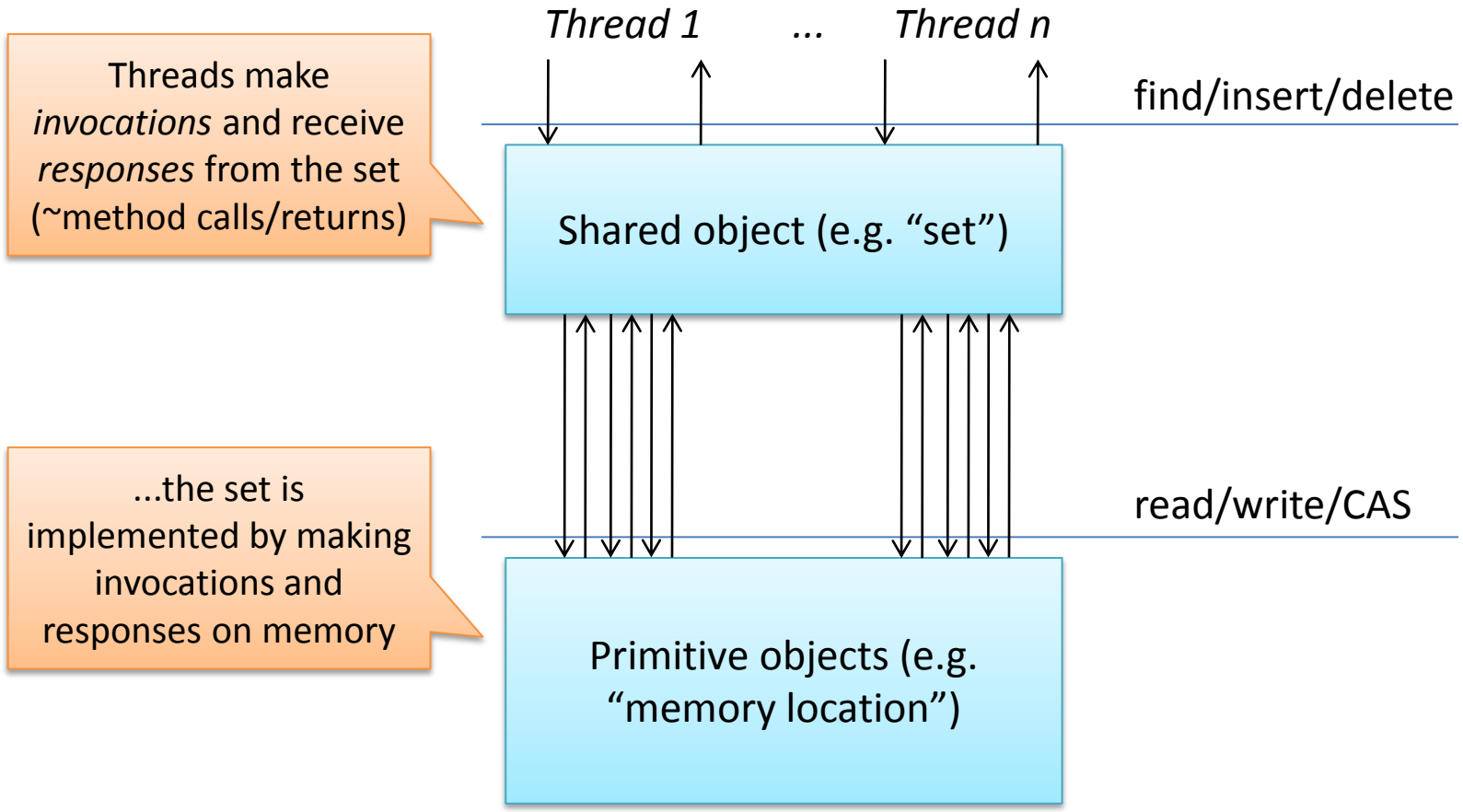
```
static volatile int MY_LIST = 0;

bool find(int key) {
  // Wait until list available
  while (CAS(&MY_LIST, 0, 1) == 1) {
  }

  ...

  // Release list
  MY_LIST = 0;
}
```

OK, we're not calling pthread_mutex_lock... but we're essentially doing the same thing

"Lock-free"

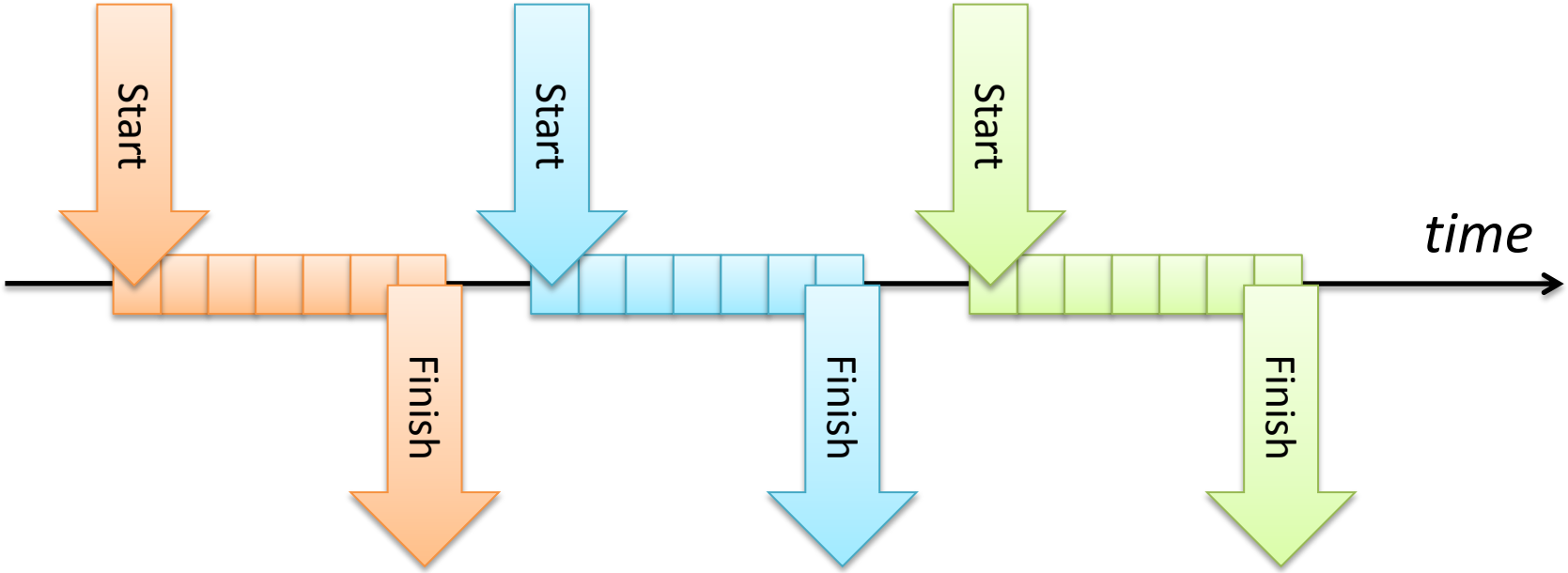
- A specific kind of *non-blocking* progress guarantee
- Precludes the use of typical locks
  - From libraries
  - Or "hand rolled"
- Often mis-used informally as a synonym for
  - Free from calls to a locking function
  - Fast
  - Scalable

Extending the system model



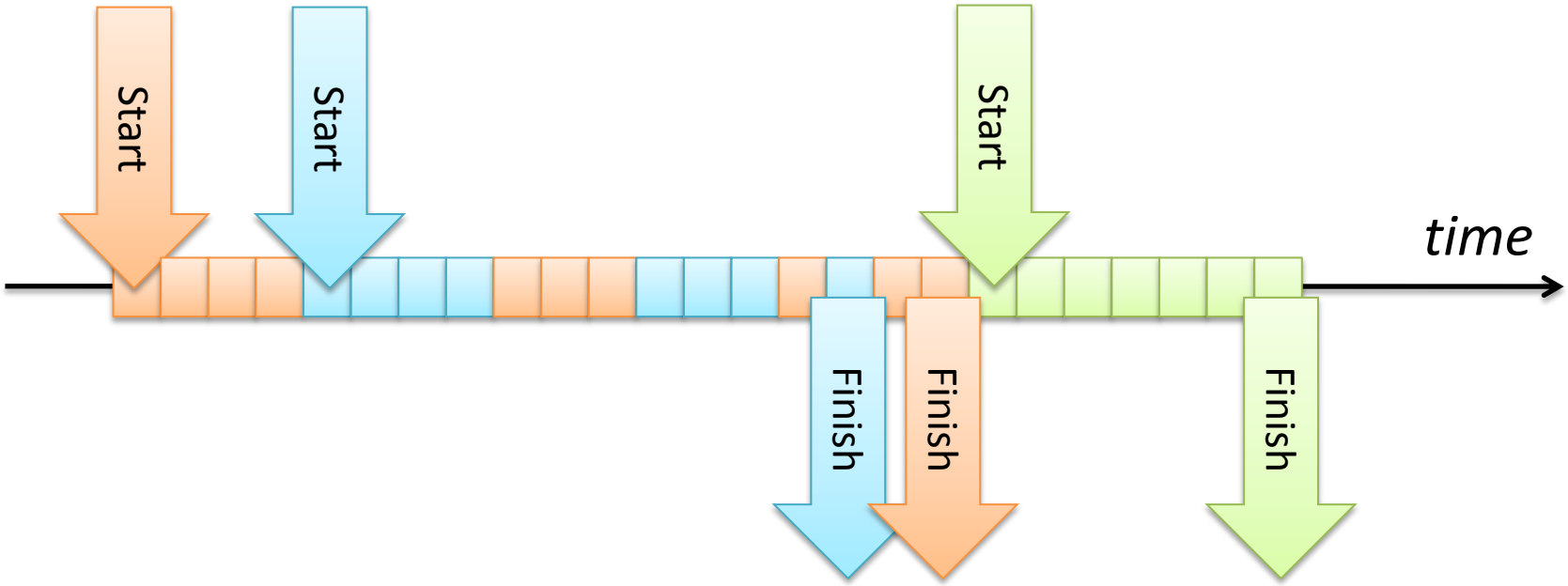
Execution model

- Threads start/finish *operations*
- Threads execute *steps* in the implementation



Wait-free

- A thread finishes its own operation if it continues executing steps

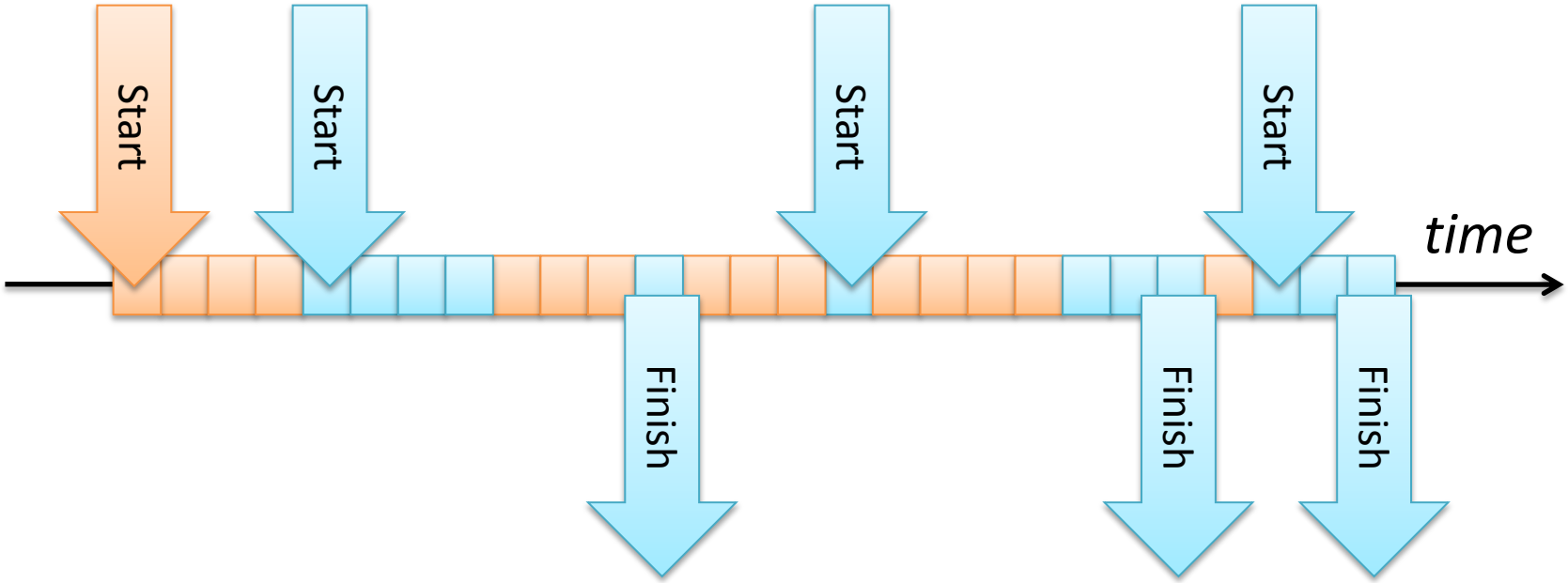


Implementing wait-free algorithms

- A few special cases
- Hybrids (e.g., wait-free find)
- Queuing and helping strategies: everyone ensures oldest operation makes progress
- Niches, e.g., bounded-wait-free in real-time systems

Lock-free

- Some thread finishes its operation if threads continue taking steps

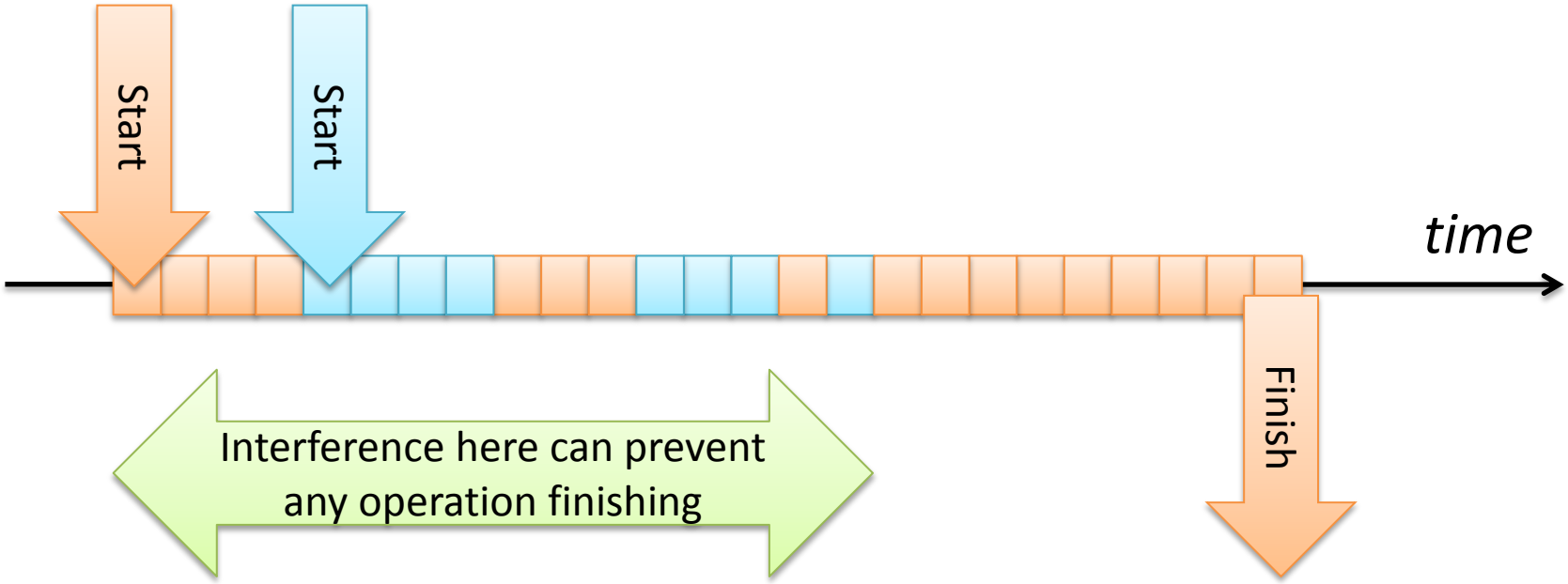


Implementing lock-free algorithms

- Ensure that one thread (A) only has to repeat work if some other thread (B) has made "real progress"
  - e.g., insert(x) starts again if it finds that a conflicting update has occurred
- Use helping to let one thread finish another's work
  - e.g., physically deleting a node on its behalf

Obstruction-free

- A thread finishes its own operation if it runs in isolation



Building obstruction-free algorithms

- Ensure that none of the low-level steps leave a data structure "broken"
- On detecting a conflict:
  - Help the other party finish
  - Get the other party out of the way
- Use *contention management* to reduce likelihood of live-lock

Lock-freedom

- Lock-free (progress criteria)
- Written without using locks
- Written for scalable and perf

What's wrong with locks?

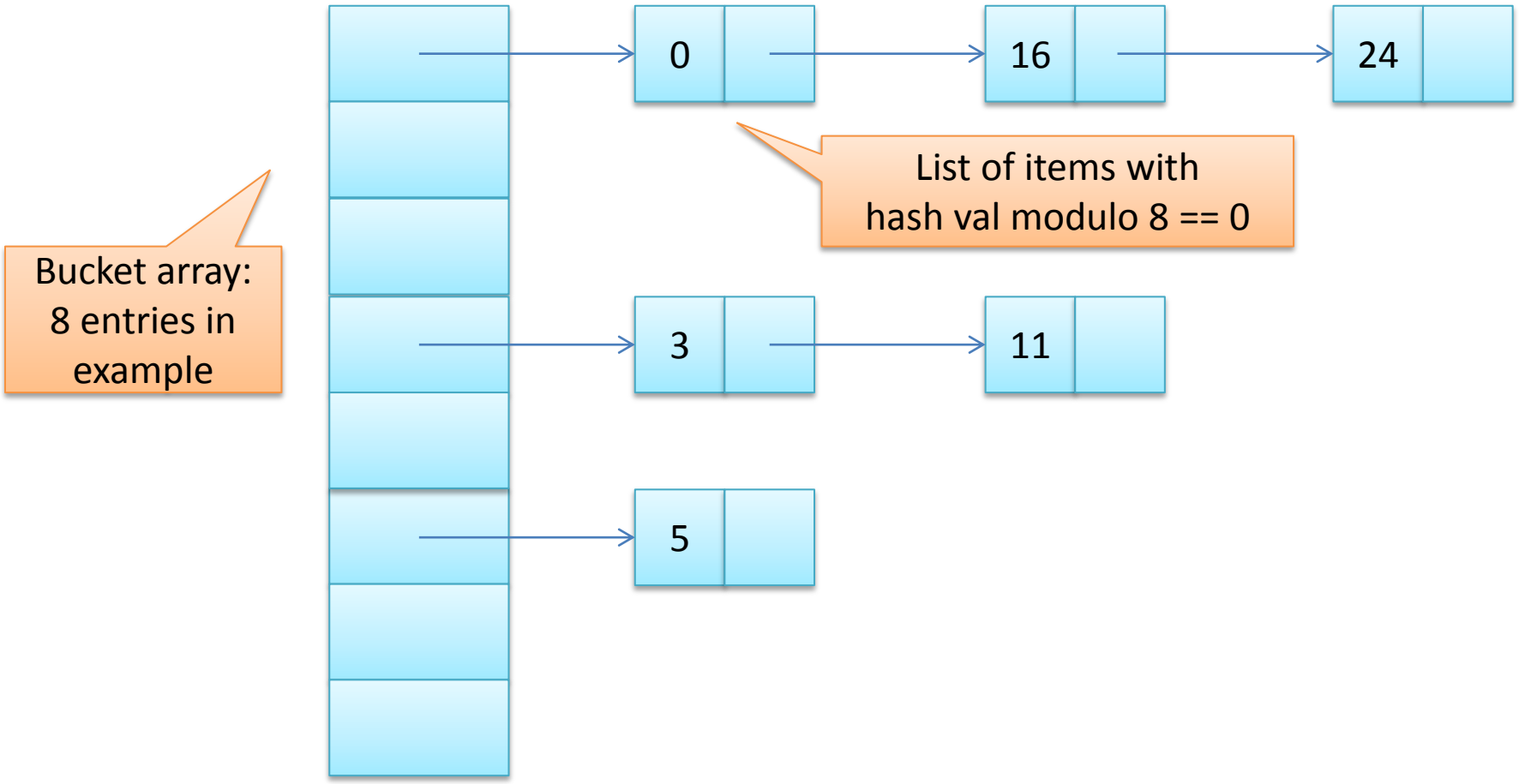
Lists without locks & linearizability

Lock-free progress

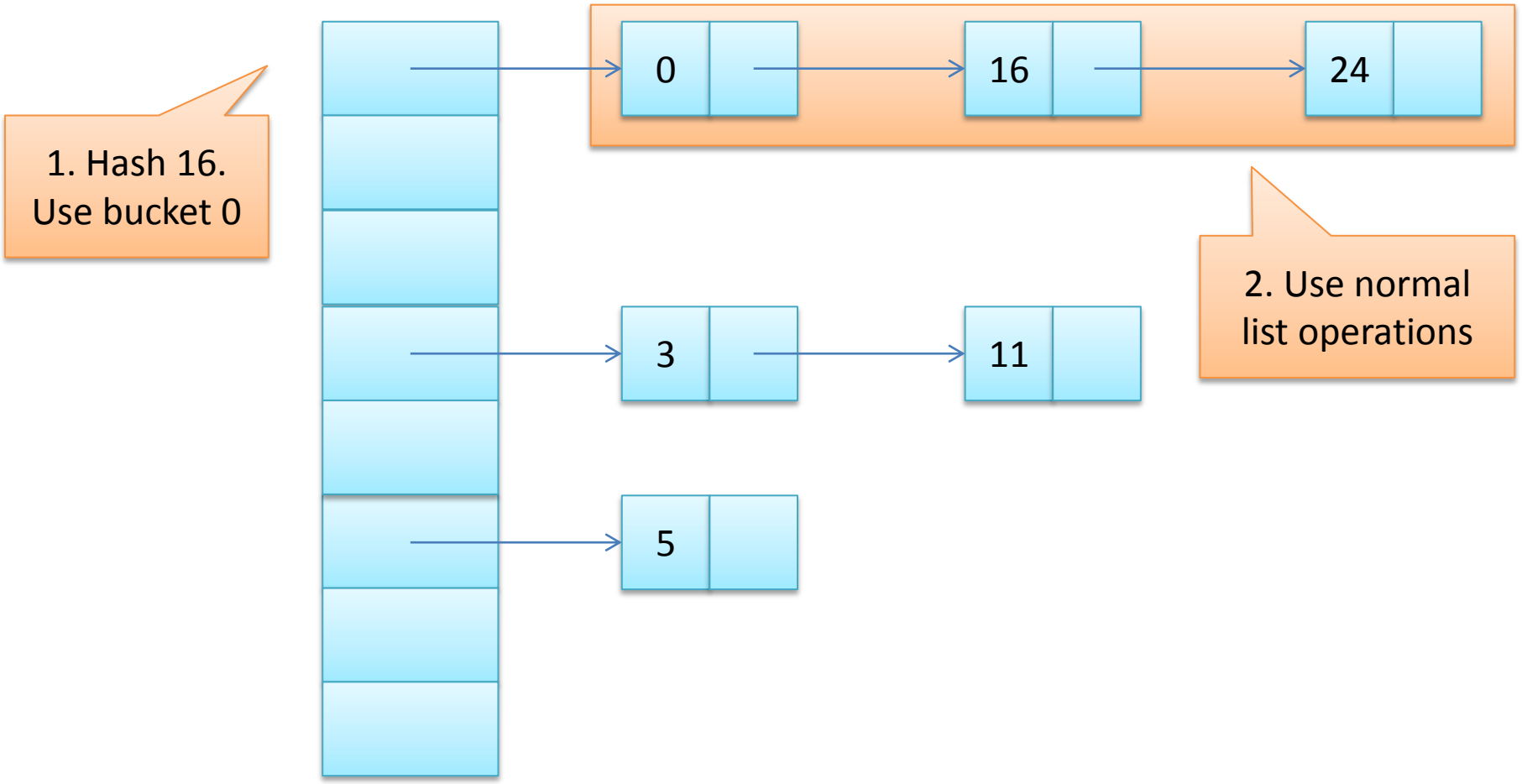
Hashtables

Skiplists

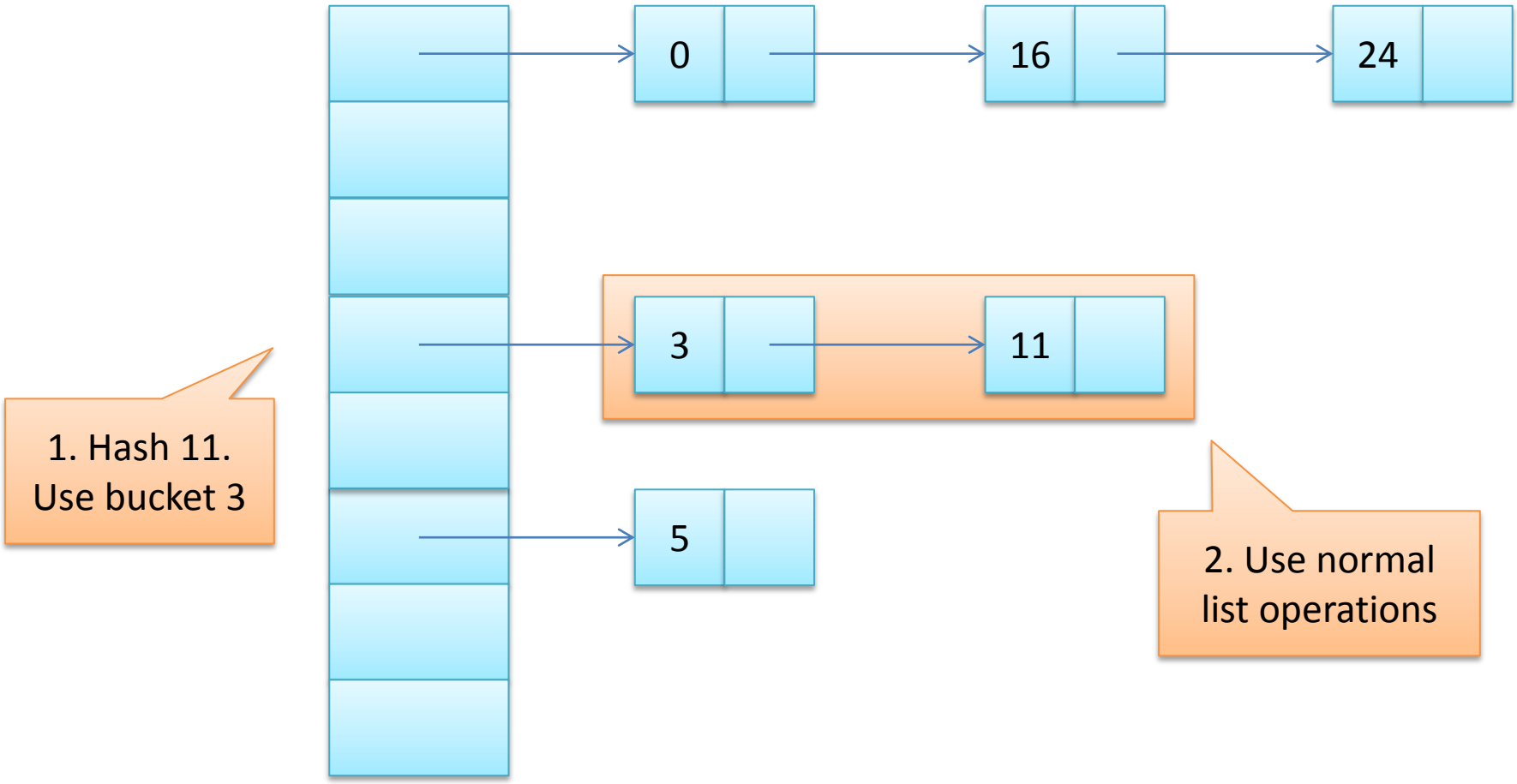
Hash tables


Bucket array:
8 entries in
example
0
16
24
List of items with
hash val modulo 8 == 0
3
11
5

Hash tables: Contains(16)



Hash tables: Delete(11)



Lessons from this hashtable

- Informal correctness argument:
  - Operations on different buckets don't conflict: no extra concurrency control needed
  - Operations appear to occur atomically at the point where the underlying list operation occurs
- (Not specific to lock-free lists: could use whole-table lock, or per-list locks, etc.)

Practical difficulties:

- Key
- Pop
- Iter
- Res

Options to consider when implementing a "difficult" operation:

- Relax the semantics (e.g., non-exact count, or non-linearizable count)
- Fall back to a simple implementation (e.g., lock the whole table for resize)
- Design a clever implementation (e.g., split-ordered lists)
- Use a different data structures (e.g., skip lists)

What's wrong with locks?

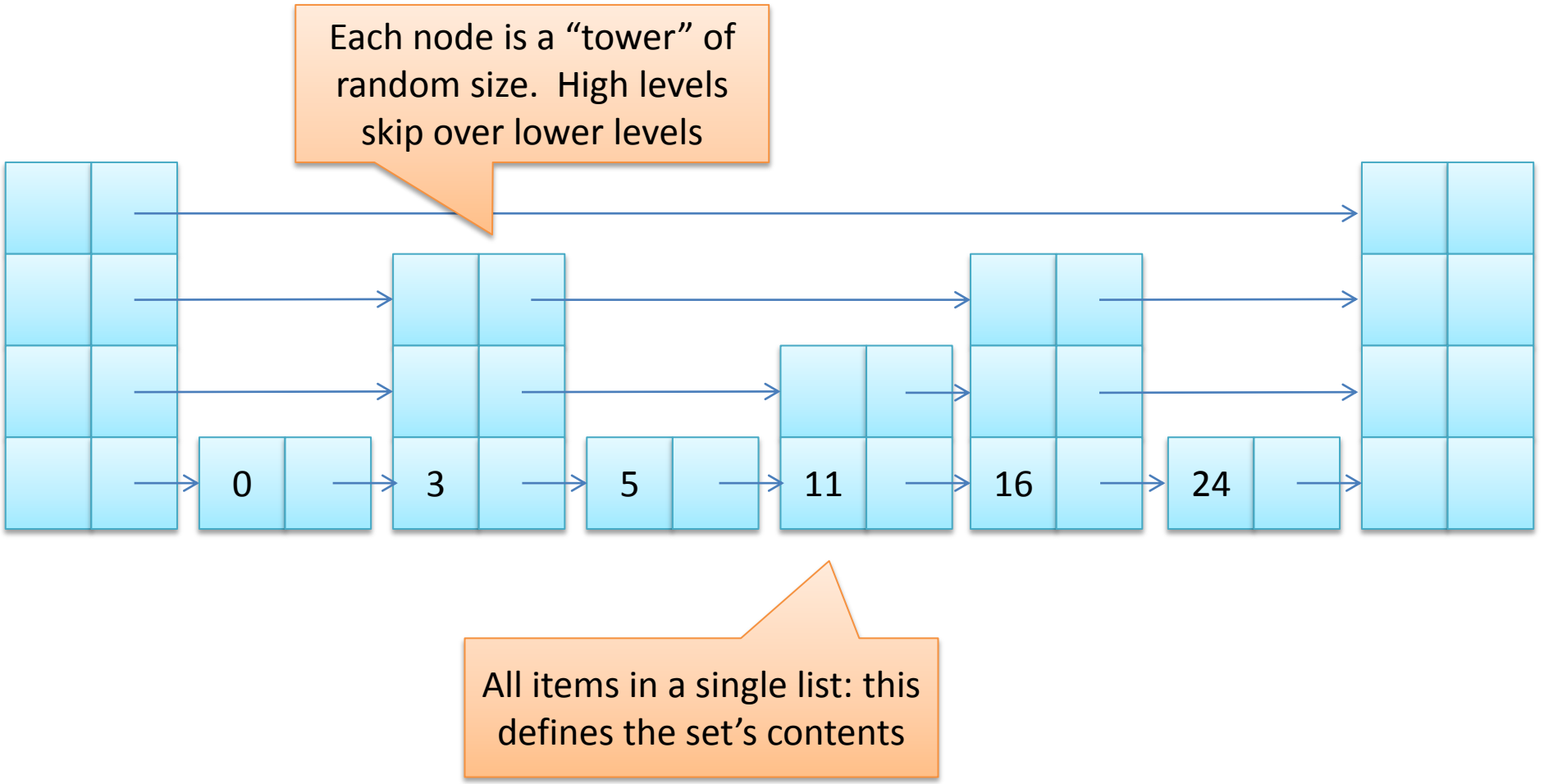
Lists without locks & linearizability

Lock-free progress

Hashtables

Skiplists

Skip lists


Each node is a “tower” of random size. High levels skip over lower levels
0
3
5
11
16
24
All items in a single list: this defines the set’s contents

Skip lists: Delete(11)

Principle: lowest list is the truth