NON-BLOCKING DATA STRUCTURES AND TRANSACTIONAL MEMORY

Tim Harris, 17 November 2017

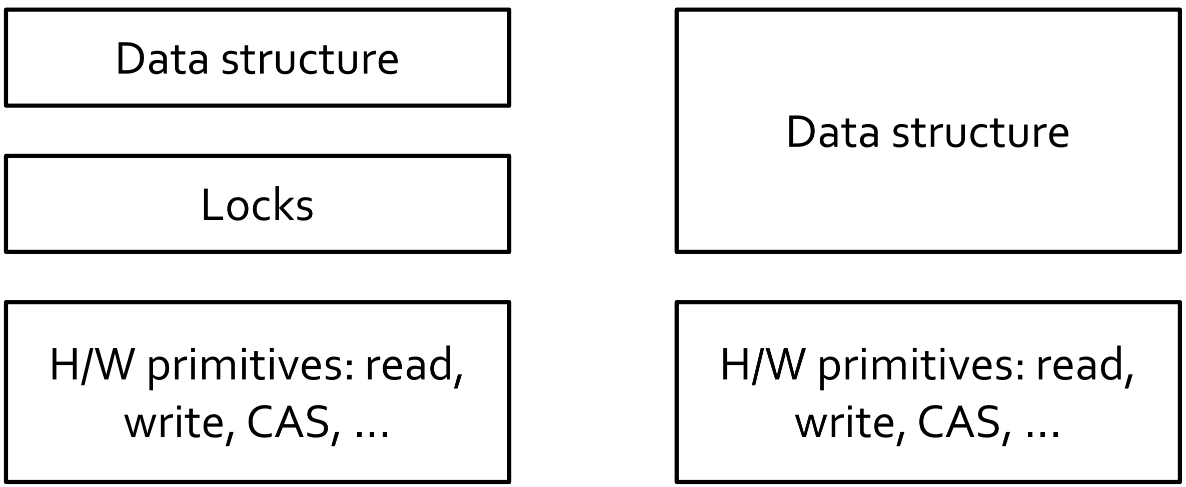
Lecture 7

- Linearizability
- Lock-free progress properties
- Hashtables and skip-lists
- Queues
- Reducing contention
- Explicit memory management

Linearizability

More generally

- Suppose we build a shared-memory data structure directly from read/write/CAS, rather than using locking as an intermediate layer



- Why might we want to do this?
- What does it mean for the data structure to be correct?

What we're building

- A set of integers, represented by a sorted linked list

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

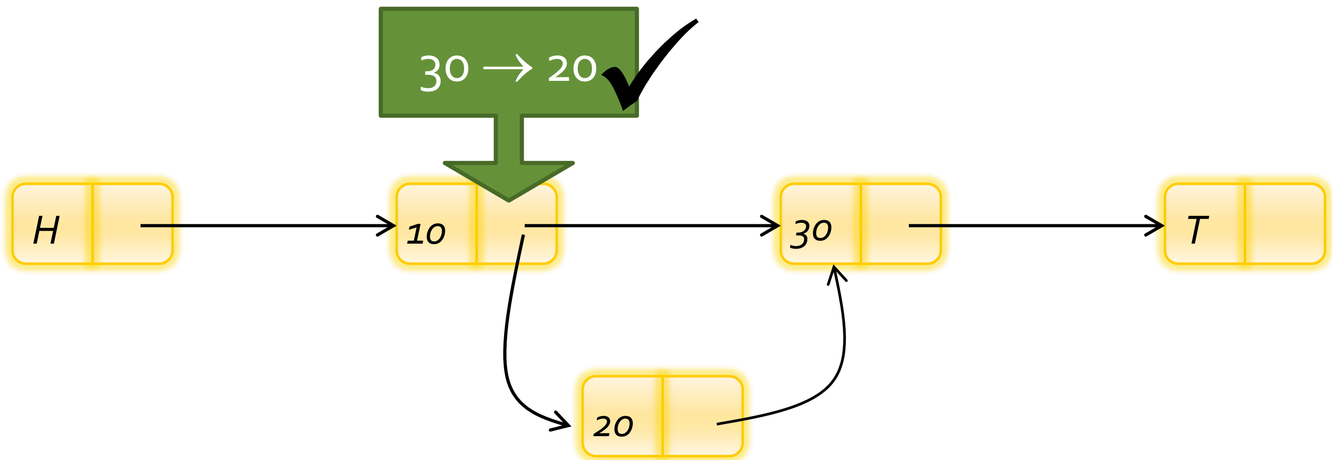
Searching a sorted list

- find(20):

find(20) -> false

Inserting an item with CAS

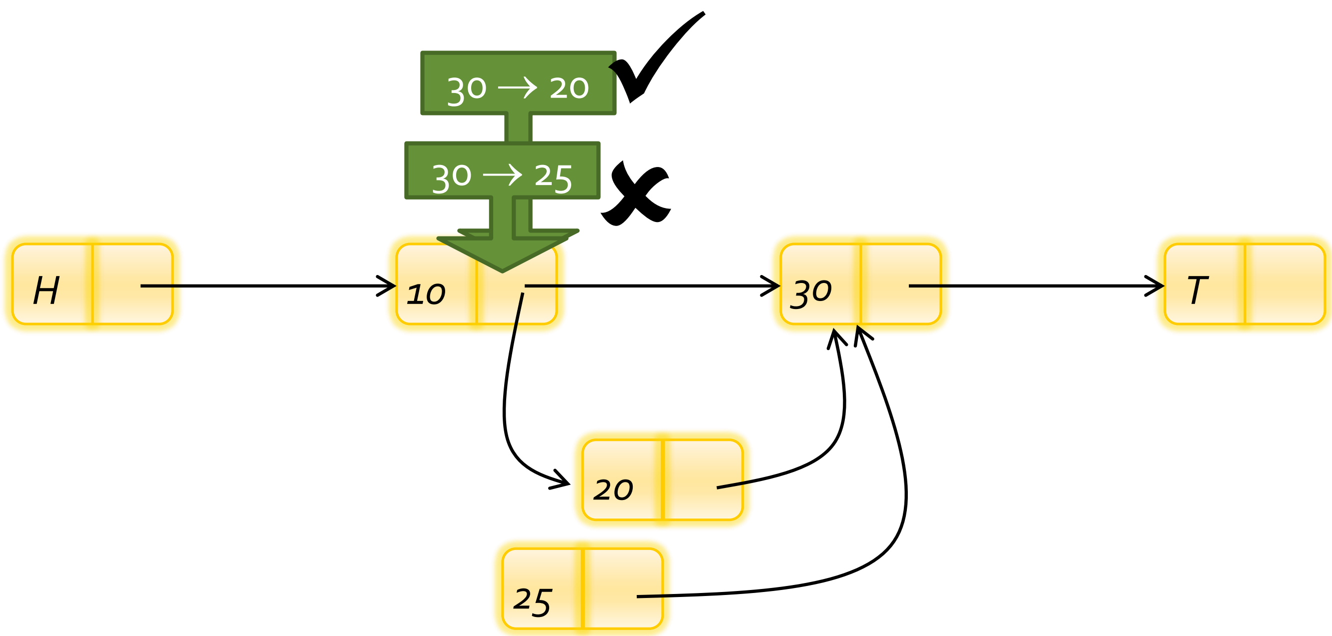
- insert(20):



insert(20) -> true

Inserting an item with CAS

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

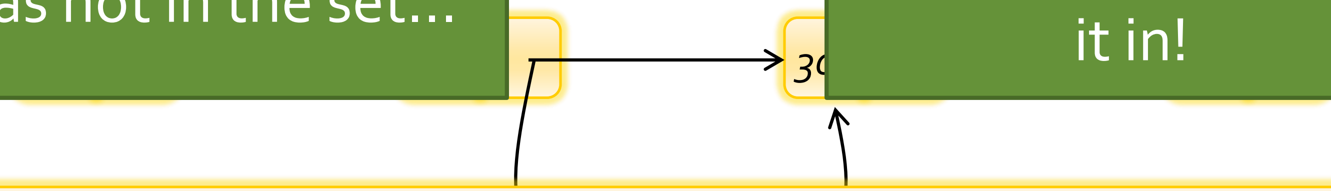


Searching and finding together

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

This thread saw 20 was not in the set...

...but this thread succeeded in putting it in!



- 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

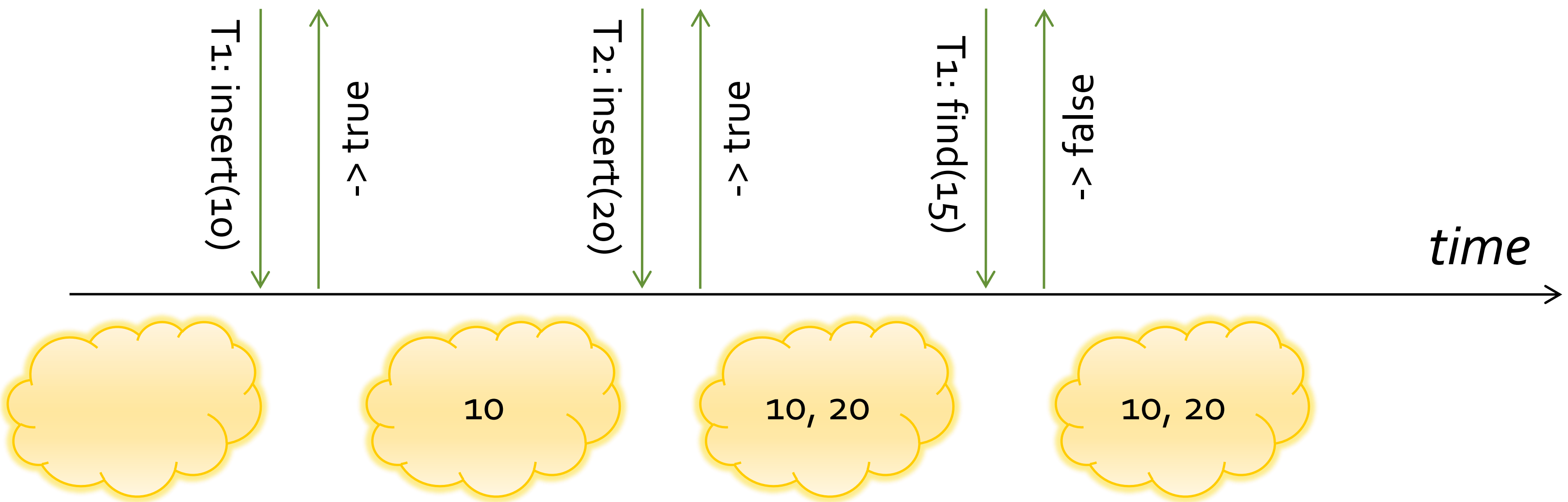
Informally:

Look at the behaviour of the data structure (what operations are called on it, and what their results are).

If this behaviour is indistinguishable from atomic calls to a sequential implementation then the concurrent implementation is correct.

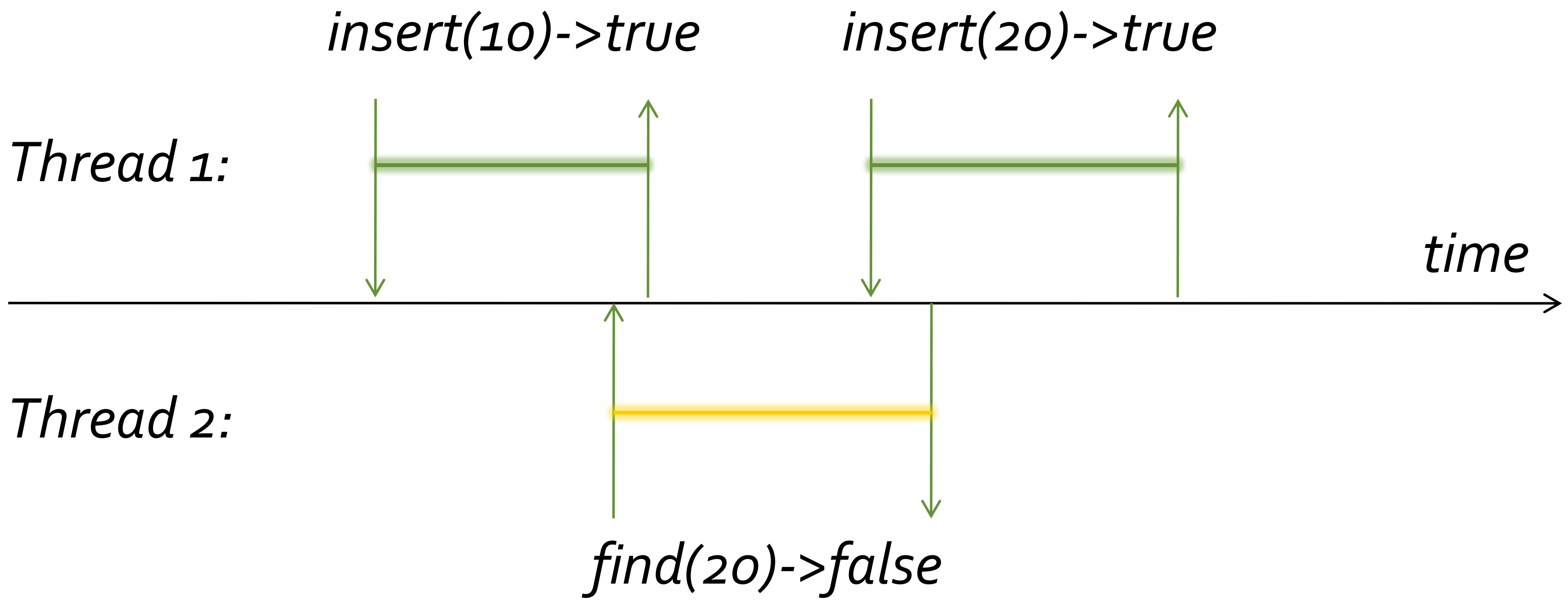
Sequential history

- No overlapping invocations:



Concurrent history

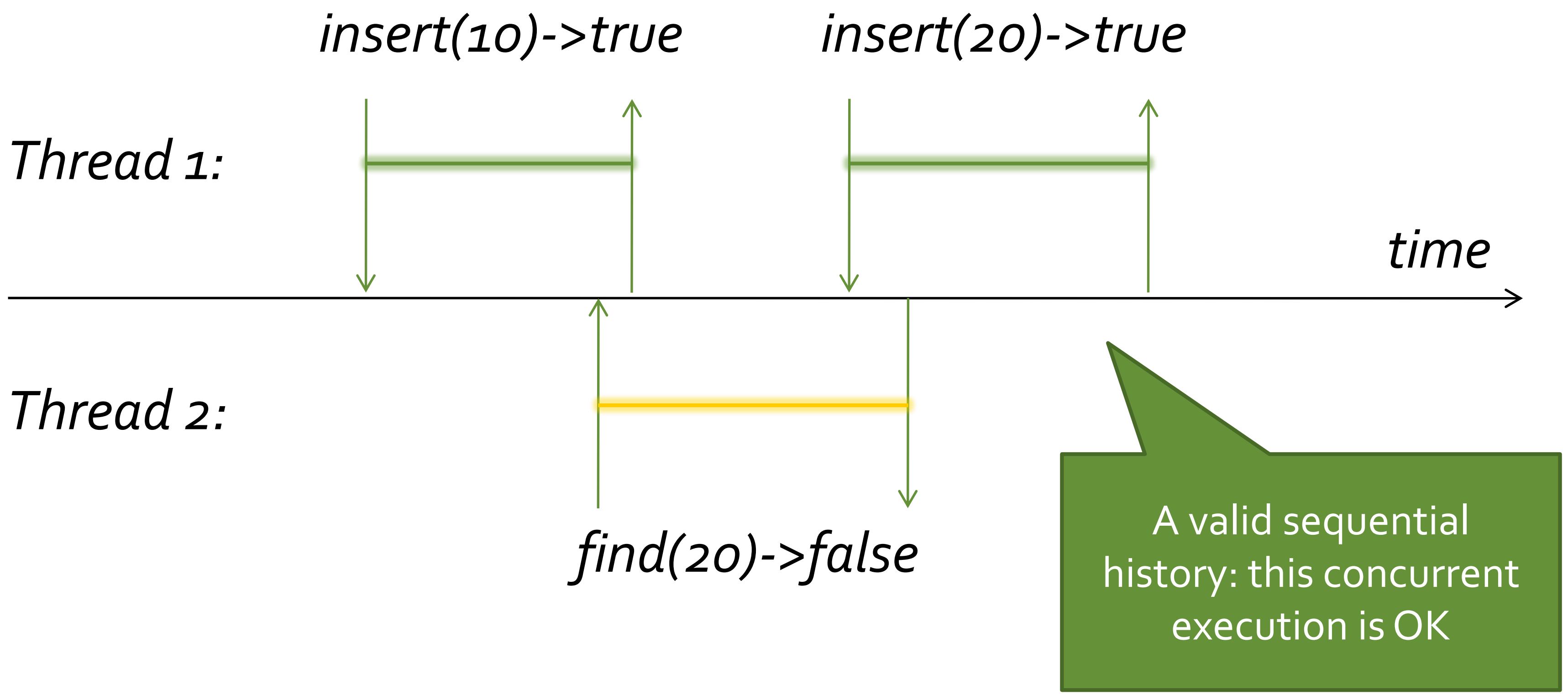
- Allow overlapping invocations:



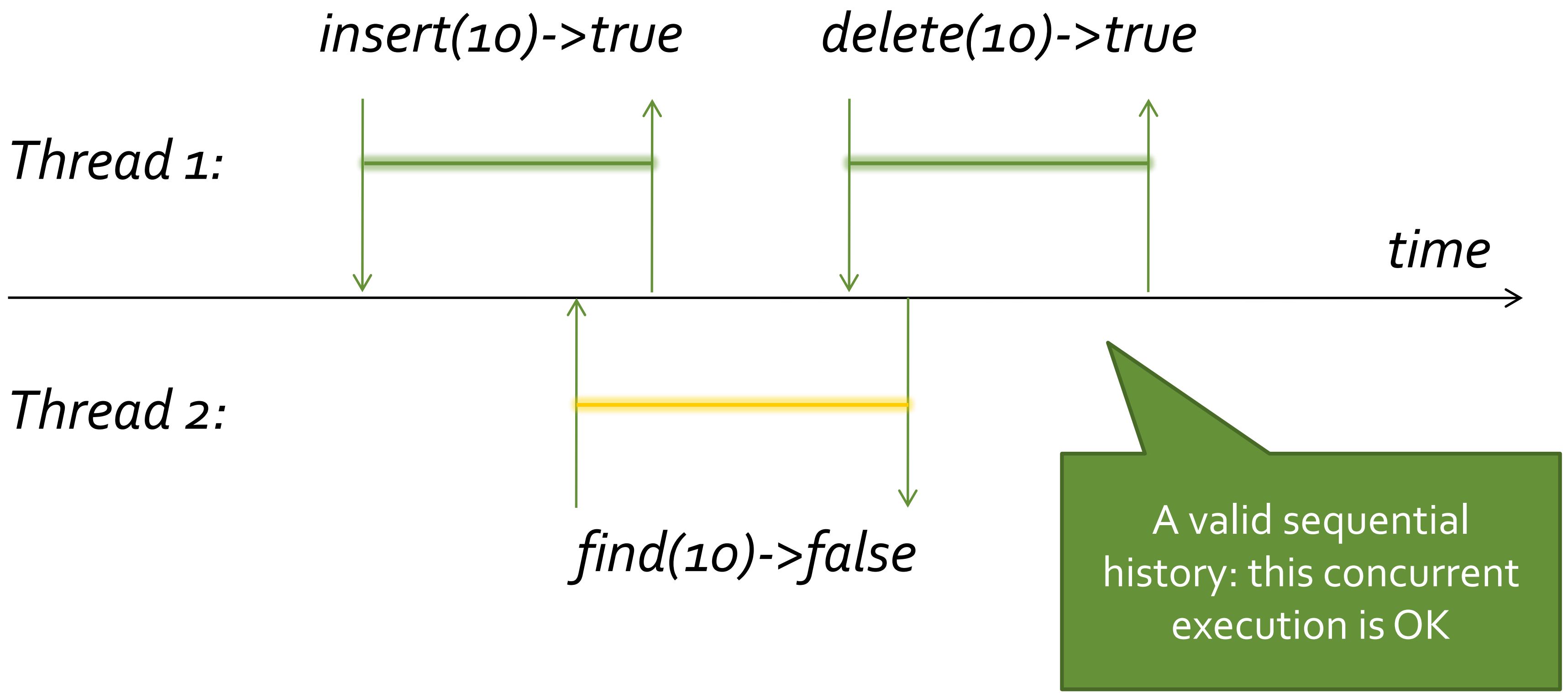
Linearizability

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

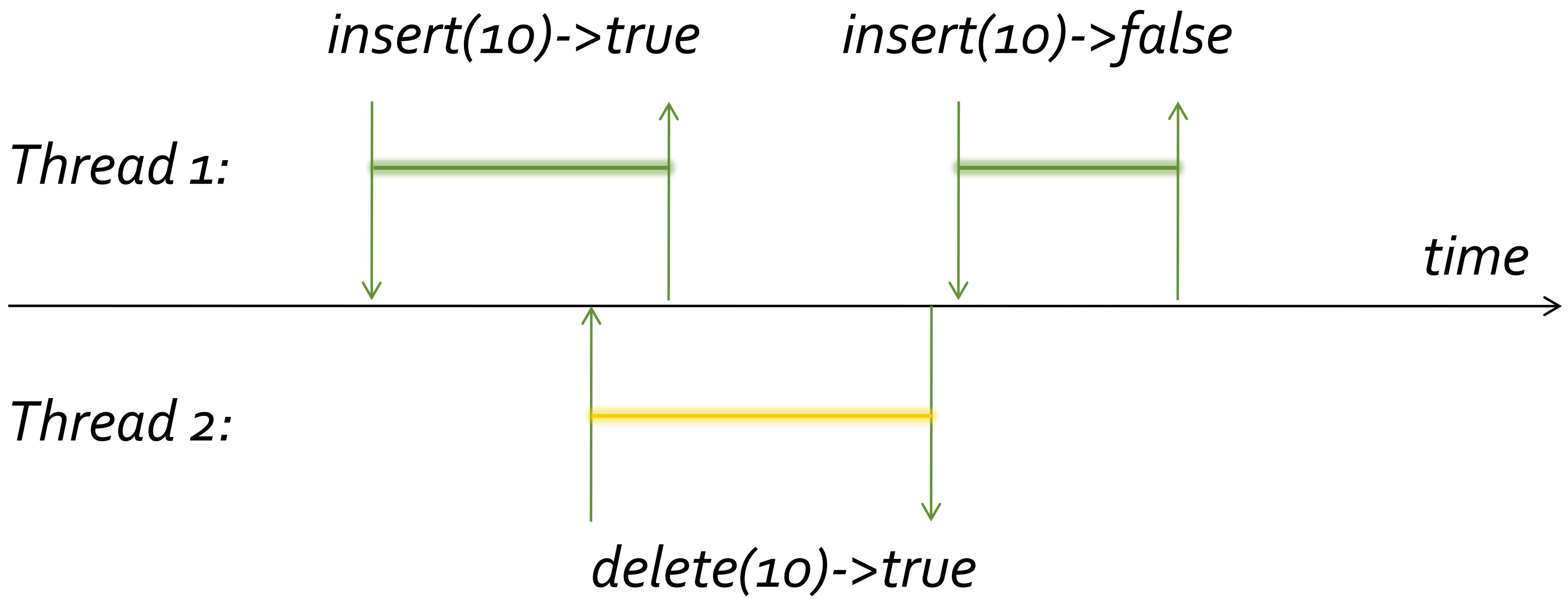
Example: linearizable



Example: linearizable

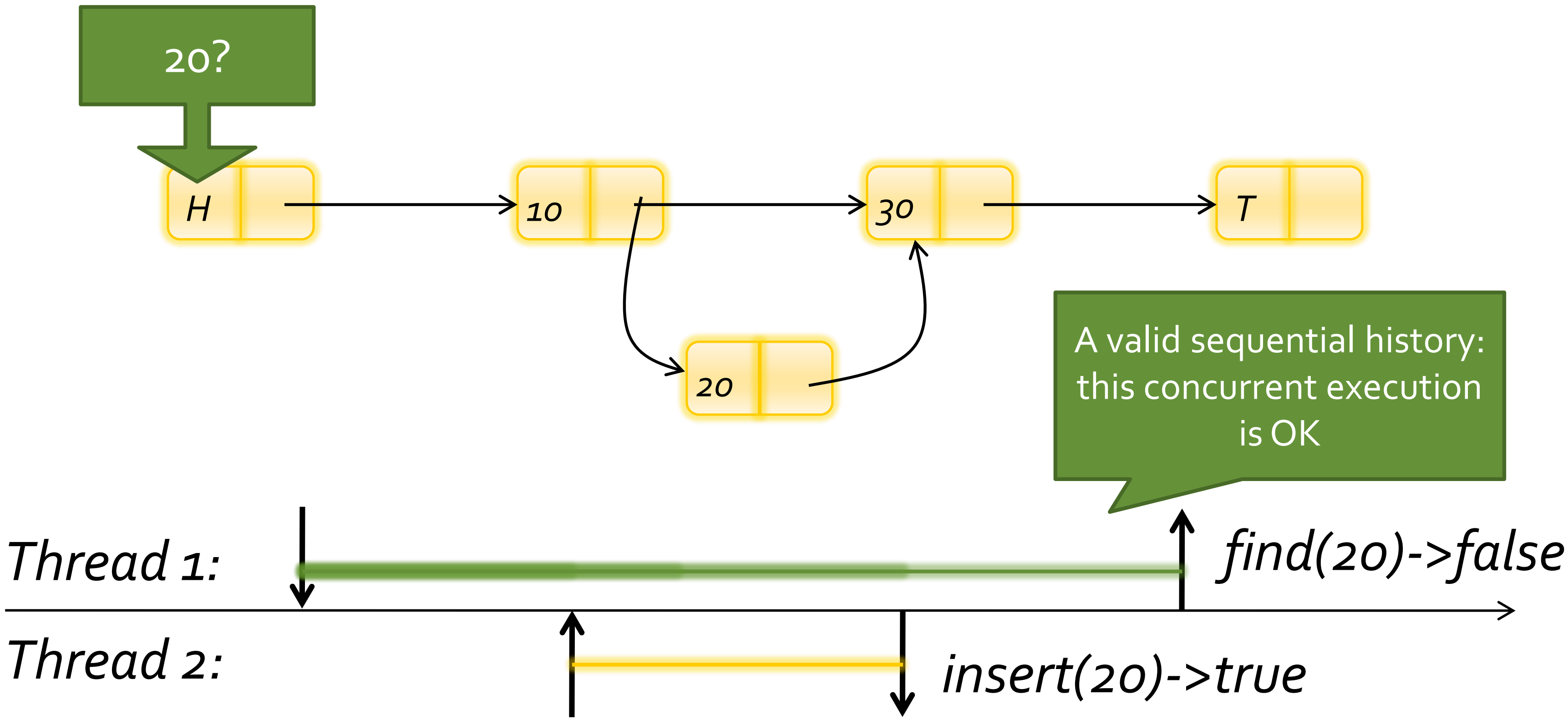


Example: not linearizable



Returning to our example

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

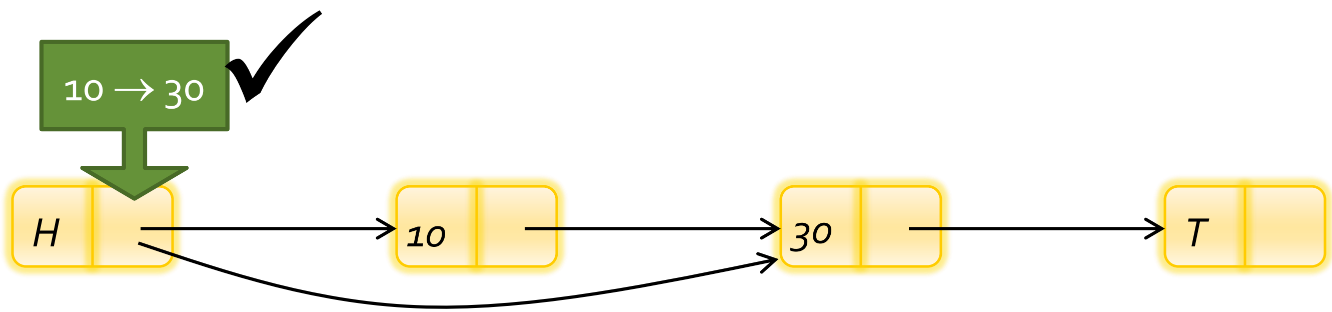


Recurring technique

- 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

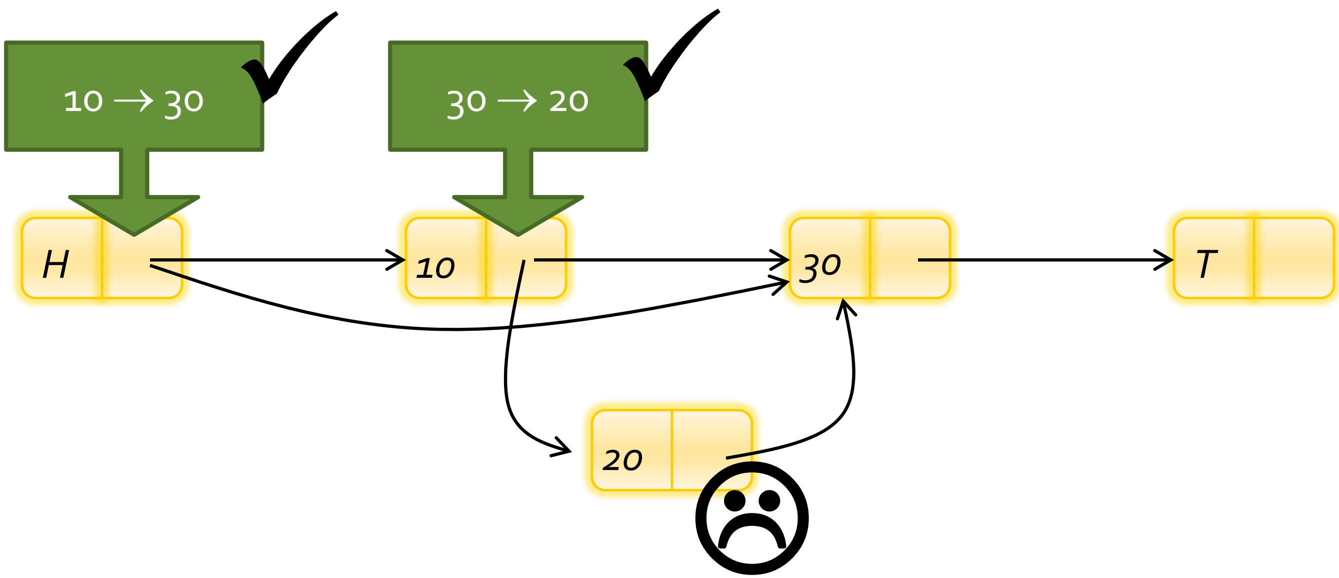
Adding "delete"

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



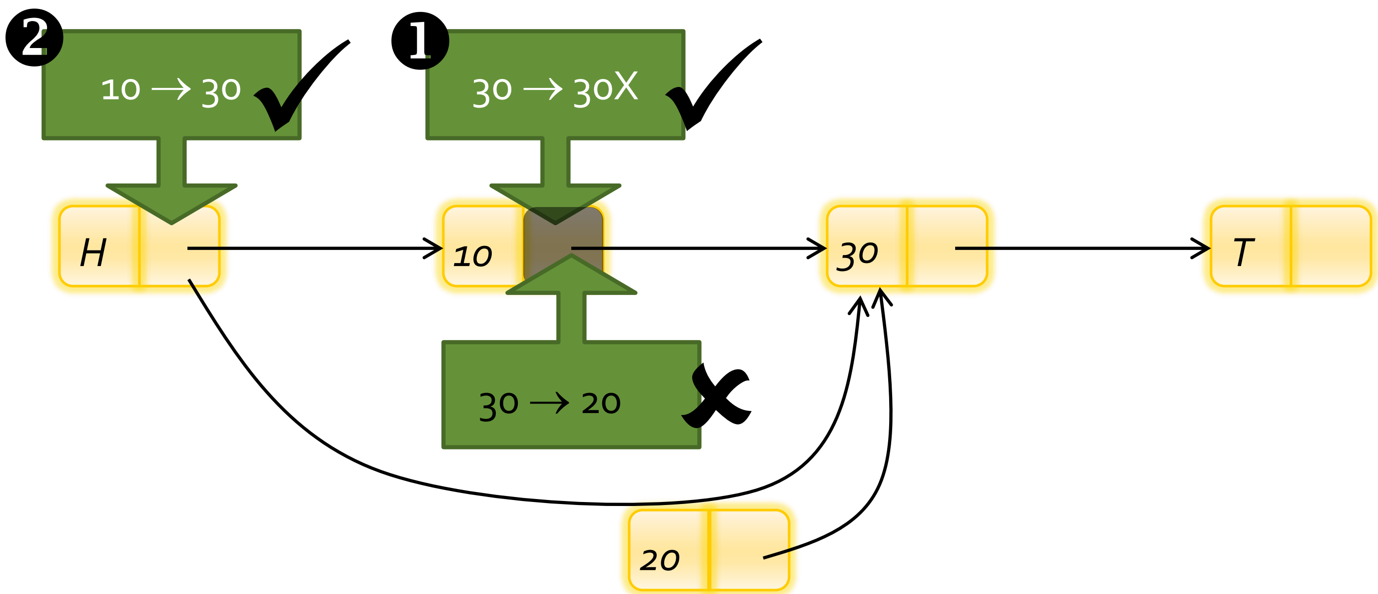
Delete and insert:

- delete(10) & insert(20):



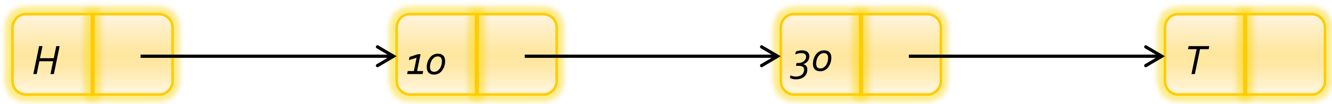
Logical vs physical deletion

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

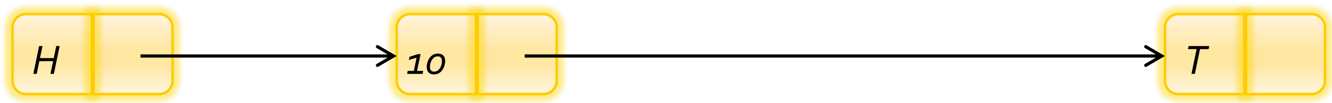


Delete-greater-than-or-equal

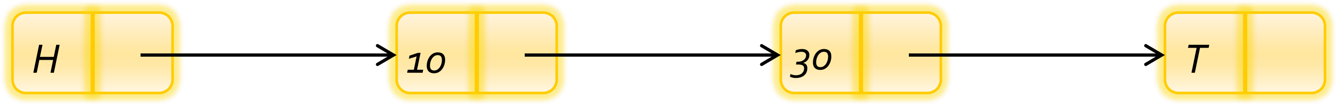
- DeleteGE(int x) -> int
  - Remove "x", or next element above "x"



- DeleteGE(20) -> 30



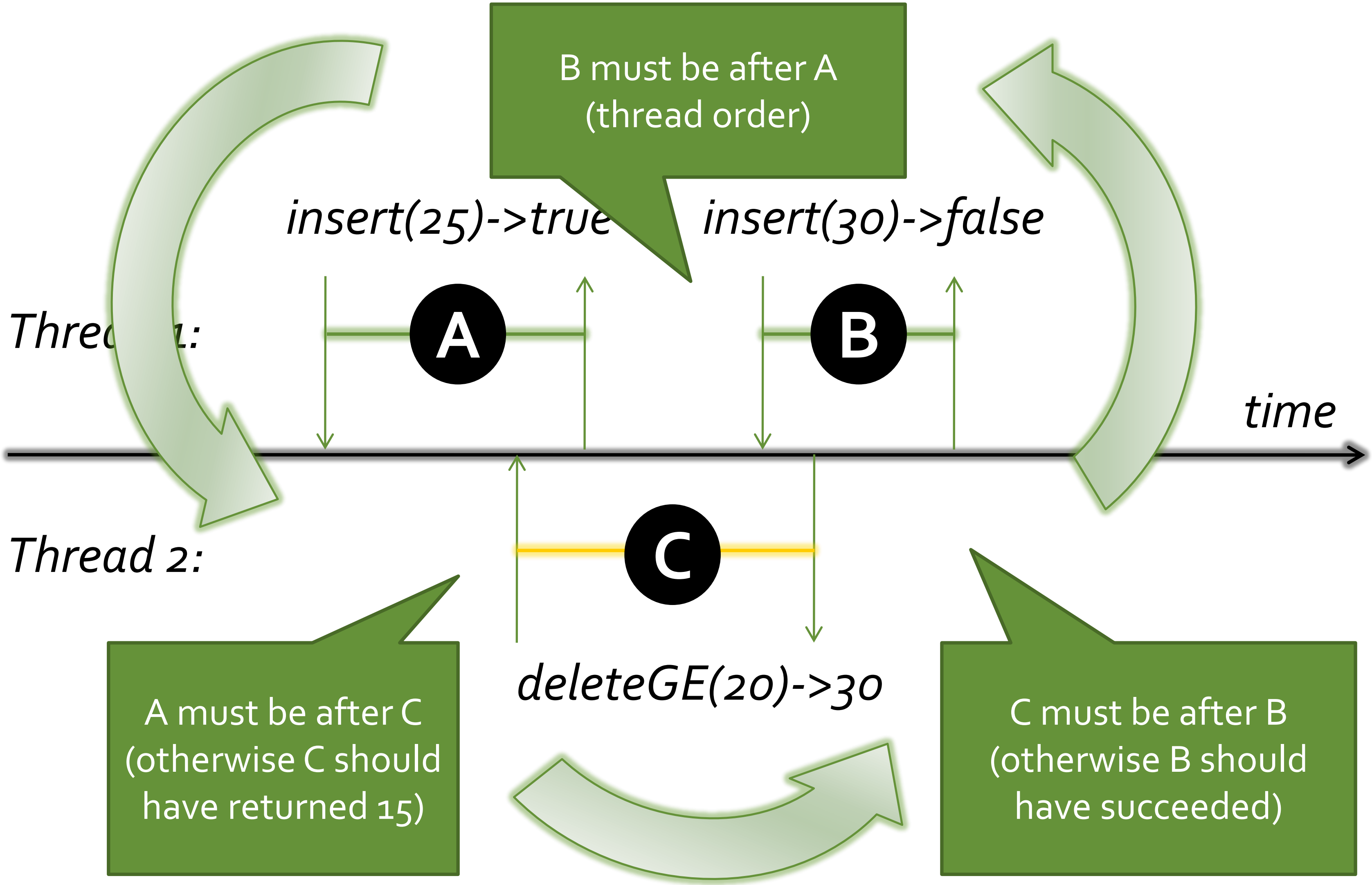
Does this work: DeleteGE(20)



1. Walk down the list, as in a normal delete, find 30 as next-after-20

2. Do the deletion as normal: set the mark bit in 30, then physically unlink

Delete-greater-than-or-equal



Lock-free progress properties

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”

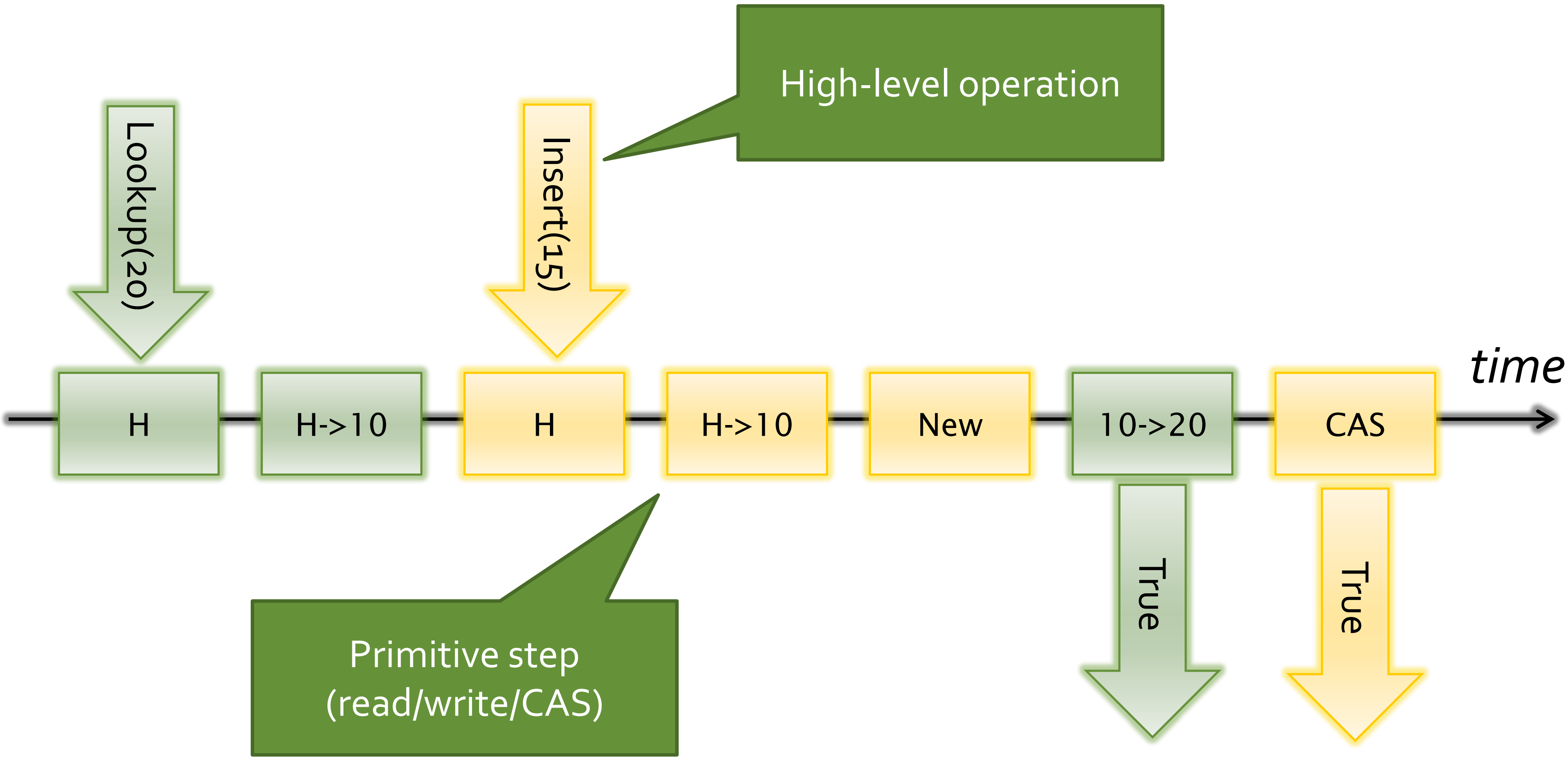
- A specific kind of *non-blocking* progress guarantee
- Precludes the use of typical locks
  - From libraries
  - Or “hand rolled”
- Often mis-used informally as a synonym for
  - Free from calls to a locking function
  - Fast
  - Scalable

"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

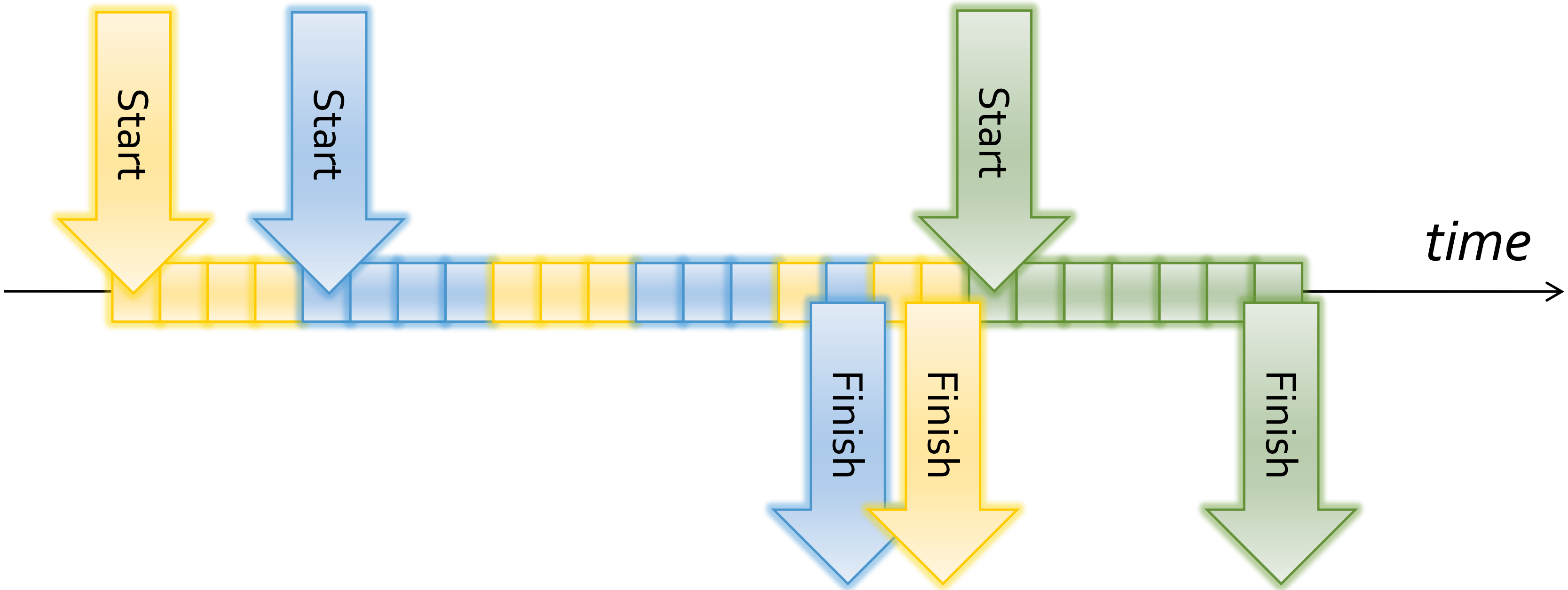
The version number mechanism is an example of a technique that is often effective in practice, does not use locks, but is not lock-free in this technical sense

System model



Wait-free

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

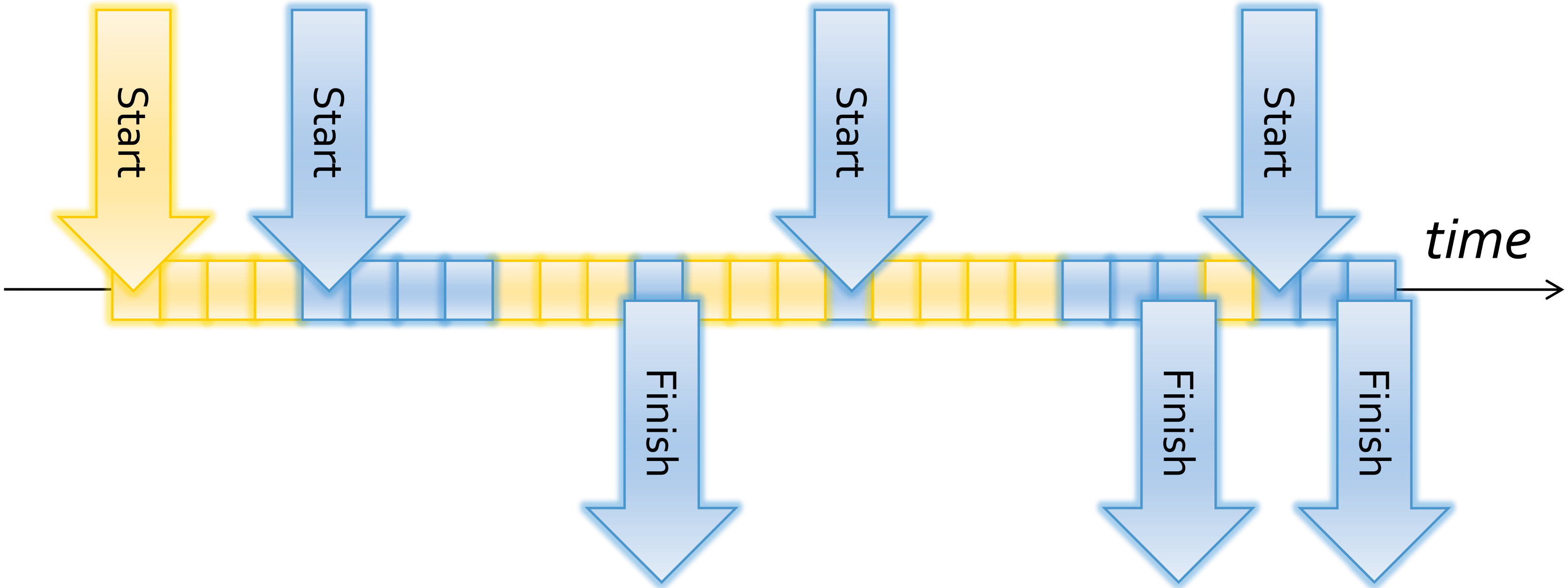


Implementing wait-free algorithms

- Important in some significant niches
  - Worst-case execution time guarantees
- General construction techniques exist ("universal constructions")
- Queuing and helping strategies: everyone ensures oldest operation makes progress
  - Often a high sequential overhead
  - Often limited scalability
- Fast-path / slow-path constructions
  - Start out with a faster lock-free algorithm
  - Switch over to a wait-free algorithm if there is no progress
  - ...if done carefully, obtain wait-free progress overall
- In practice, progress guarantees can vary between operations on a shared object
  - e.g., wait-free find + lock-free delete

Lock-free

- Some thread finishes its operation if threads continue taking steps



A (poor) lock-free counter

```
int getNext(int *counter) {
  while (true) {
    int result = *counter;
    if (CAS(counter, result, result+1)) {
      return result;
    }
  }
}
```

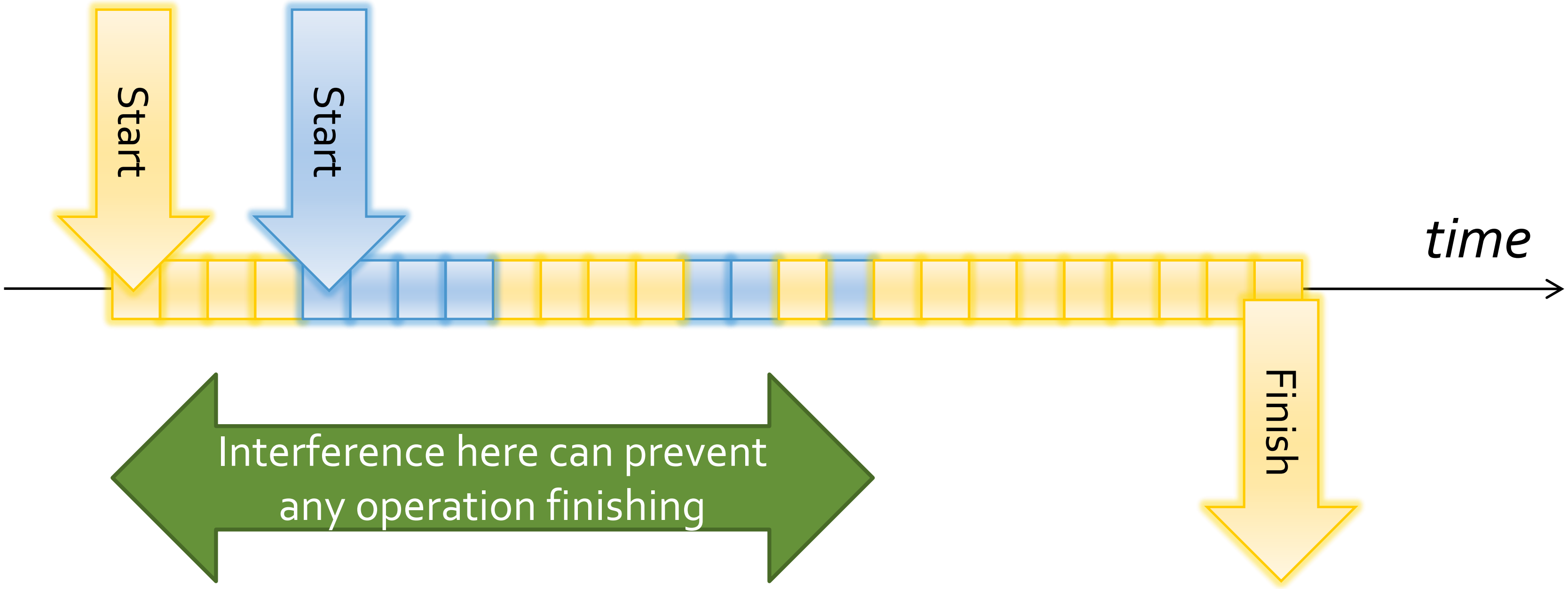
Not wait free: no guarantee that any particular thread will succeed

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



A (poor) obstruction-free counter

```
int getNext(int *counter) {
  while (true) {
    int result = LL(counter);
    if (SC(counter, result+1)) {
      return result;
    }
  }
}
```

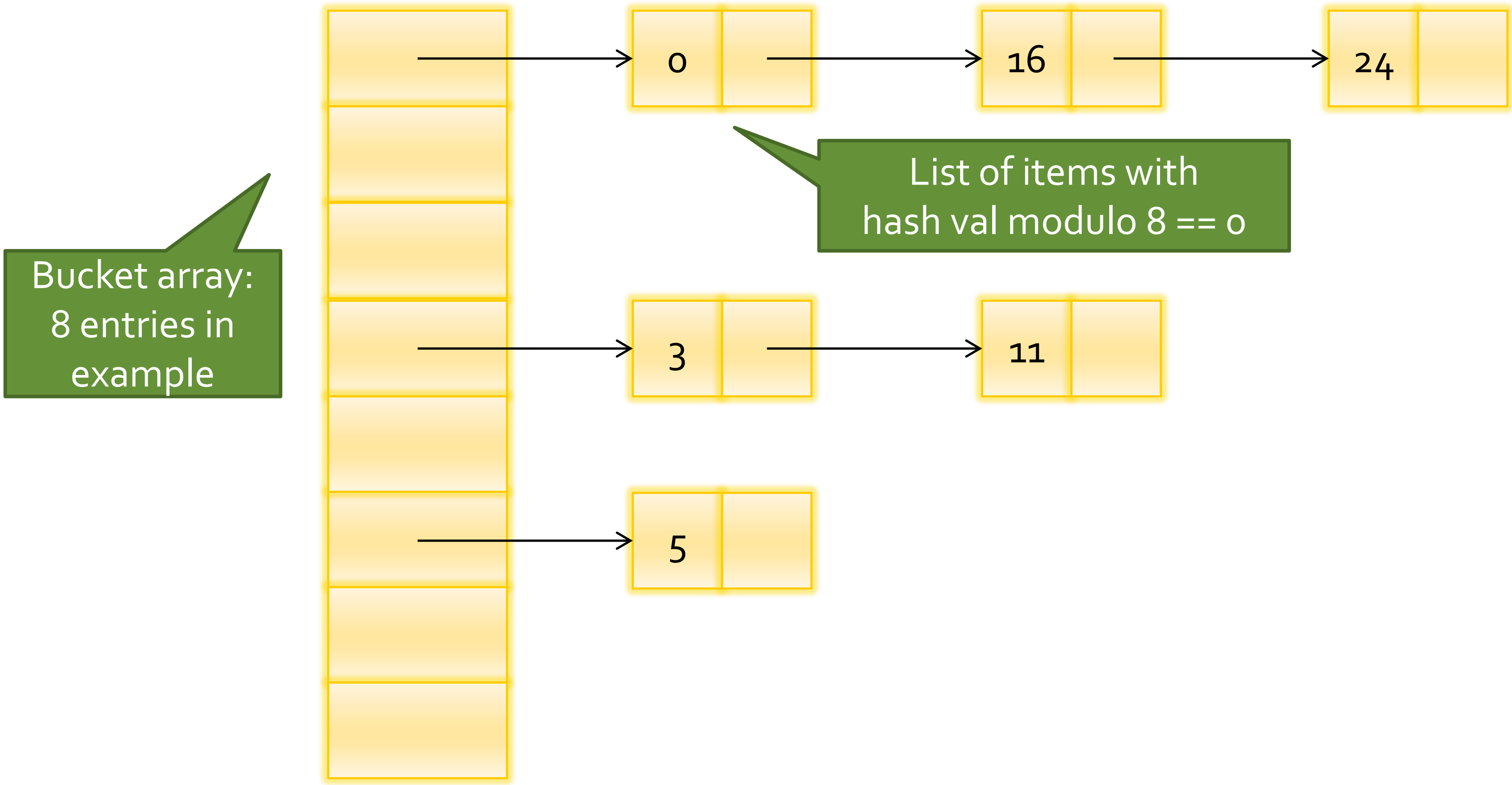
Assuming a very weak load-linked (LL) store-conditional (SC): LL on one thread will prevent an SC on another thread succeeding

Building obstruction-free algorithms

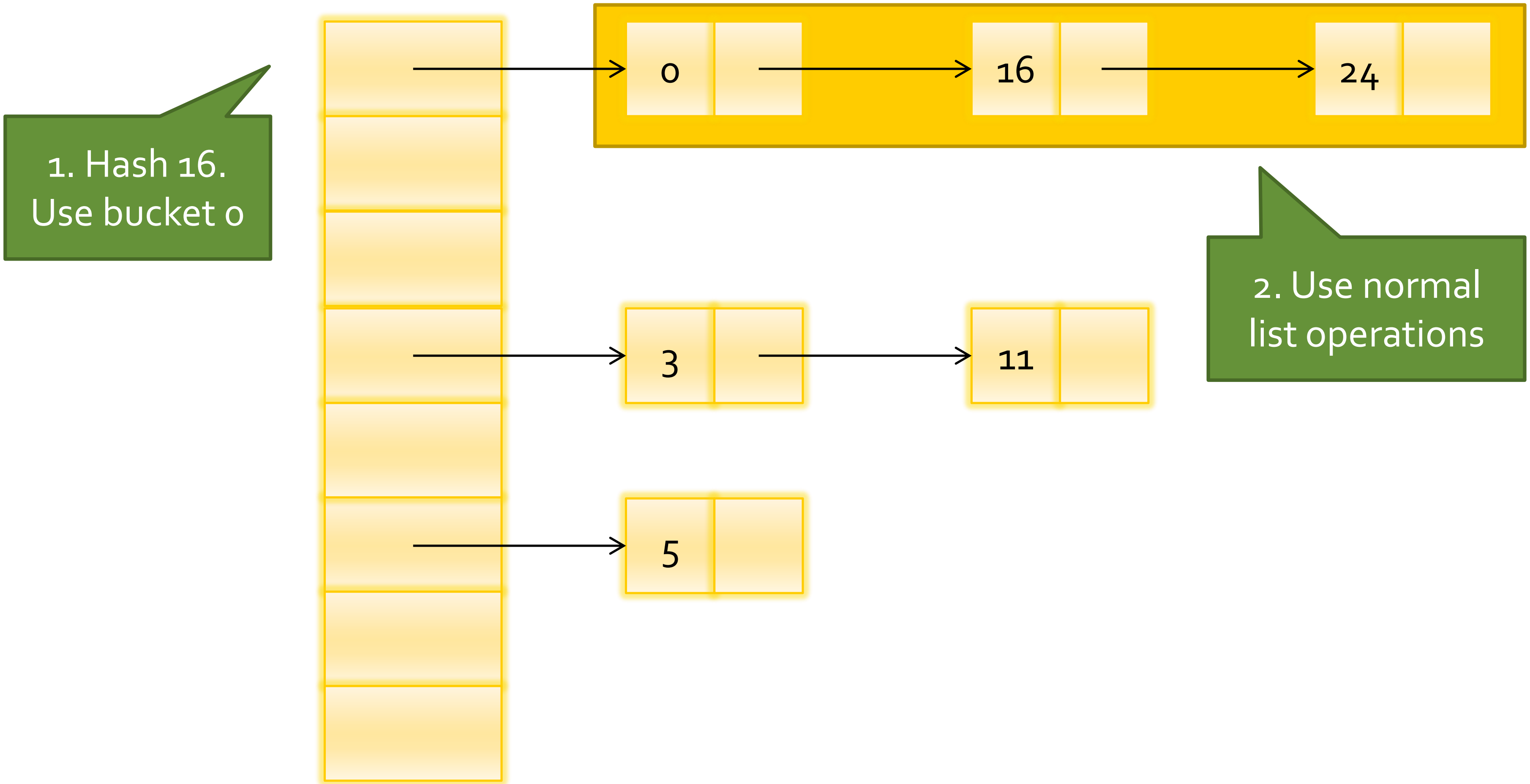
- Ensure that none of the low-level steps leave a data structure “broken”
- On detecting a conflict:
  - Help the other party finish
  - Get the other party out of the way
- Use *contention management* to reduce likelihood of live-lock

Hashtables and skiplists

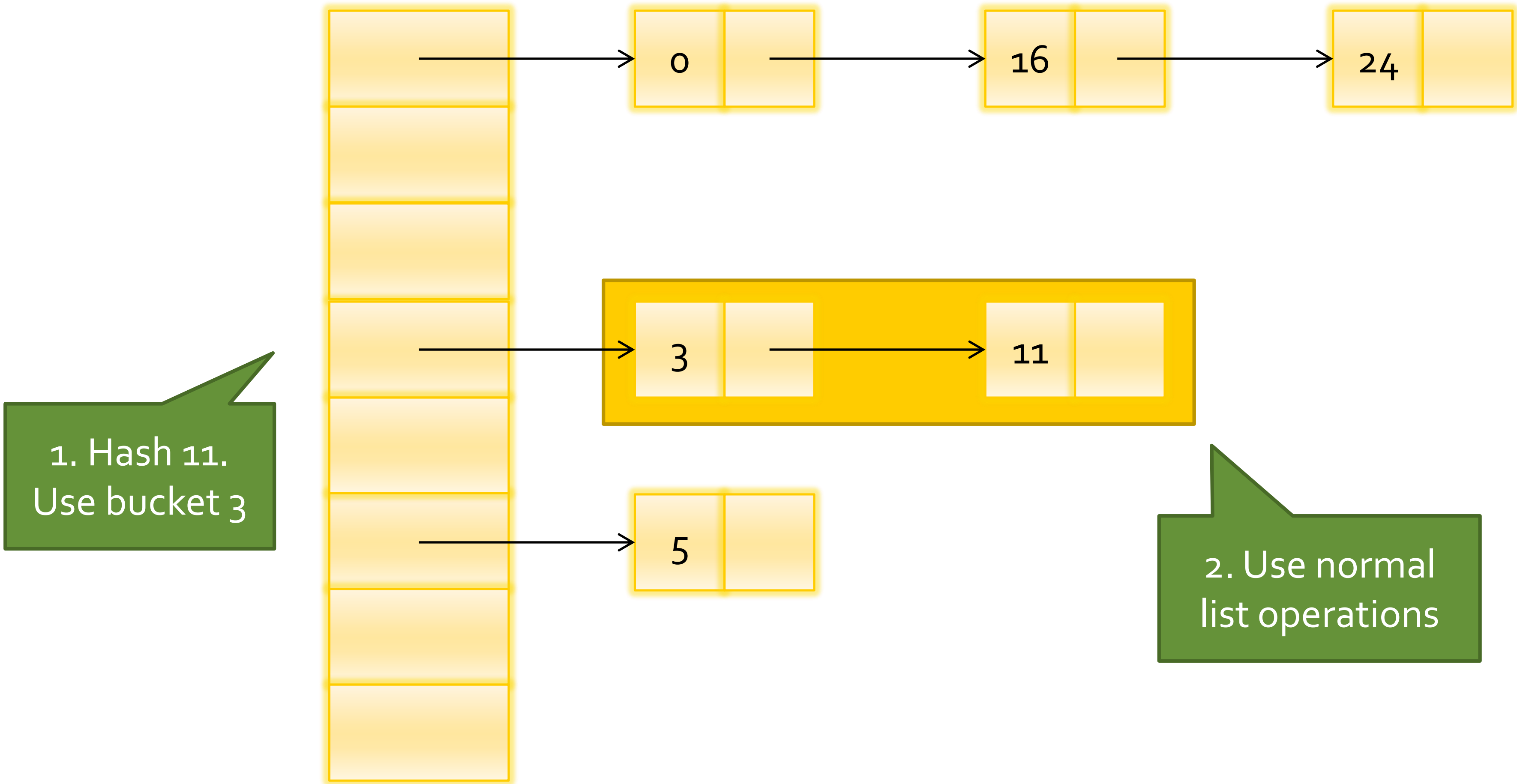
Hash tables


0
16
24
List of items with hash val modulo 8 == 0
Bucket array: 8 entries in example
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-va
- Popu
- Itera
- Resi

Options to consider when implementing a "difficult" operation:

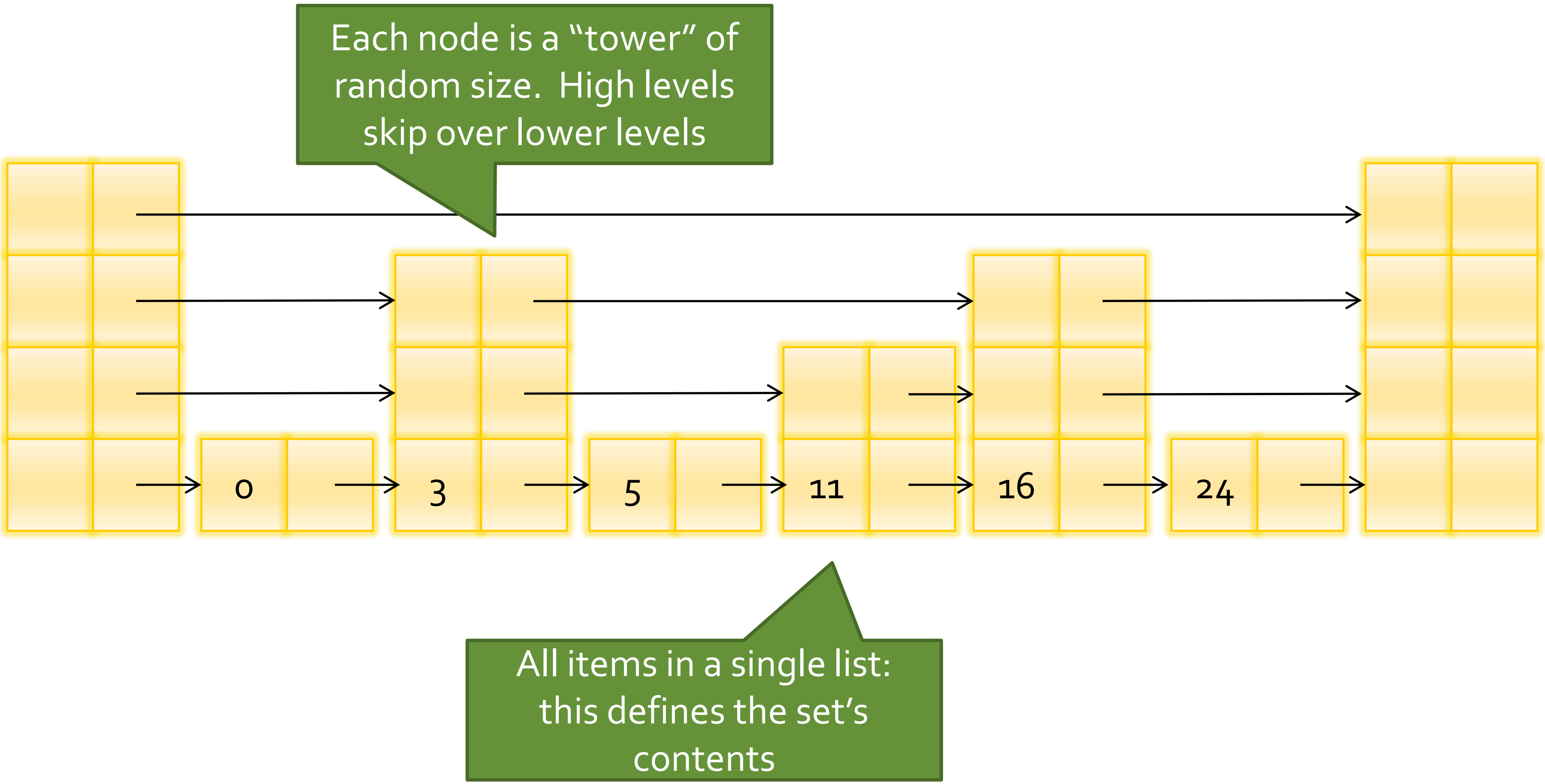
Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted
(e.g., lock the whole table for resize)

Design a clever implementation
(e.g., split-ordered lists)

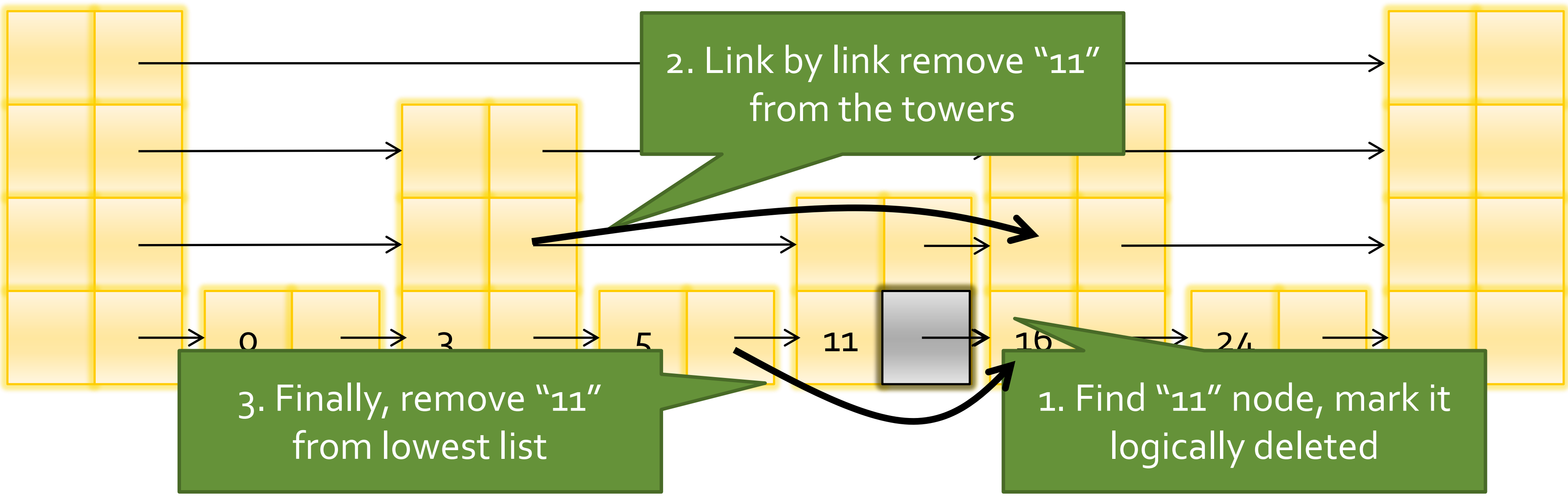
Use a different data structure
(e.g., skip lists)

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

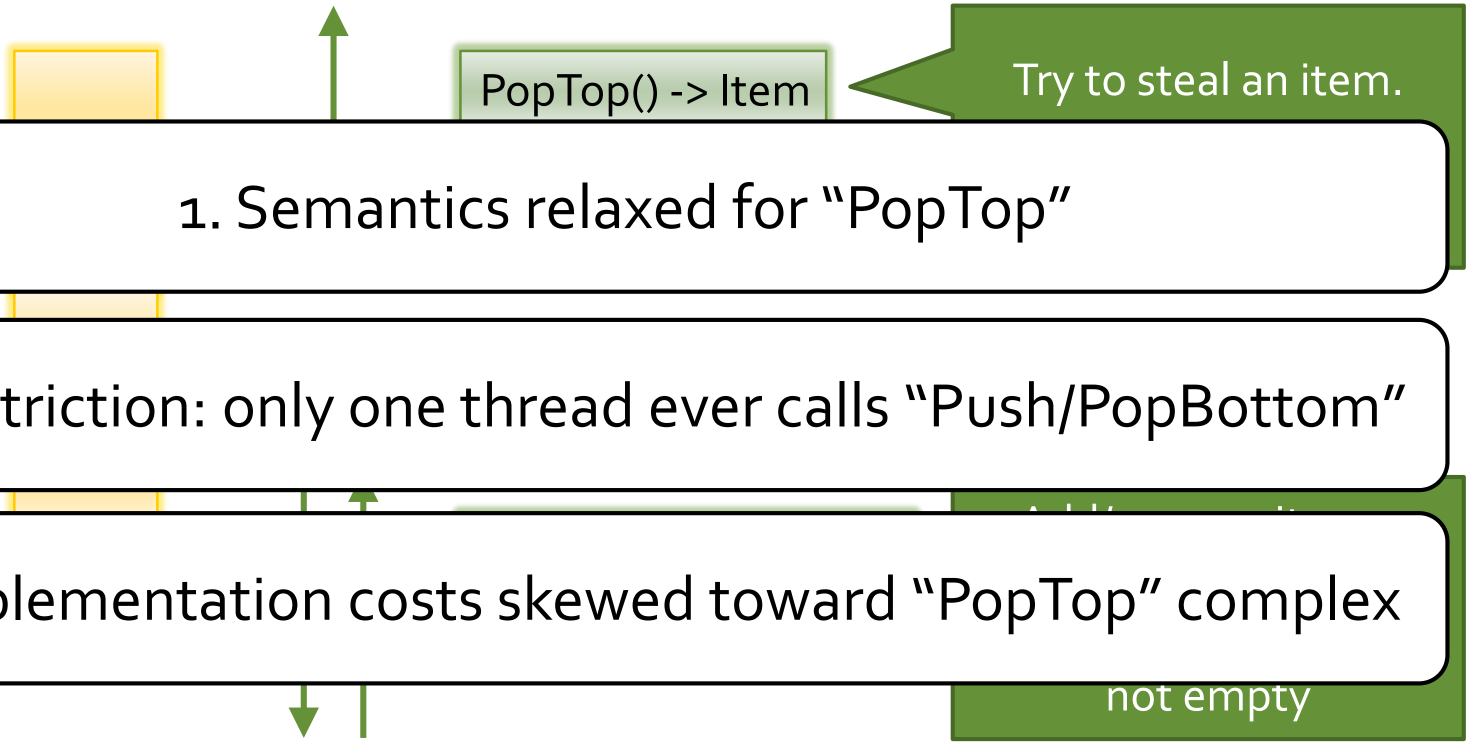
Skip lists: Delete(11)

Principle: lowest list is the truth



Queues

Work stealing queues

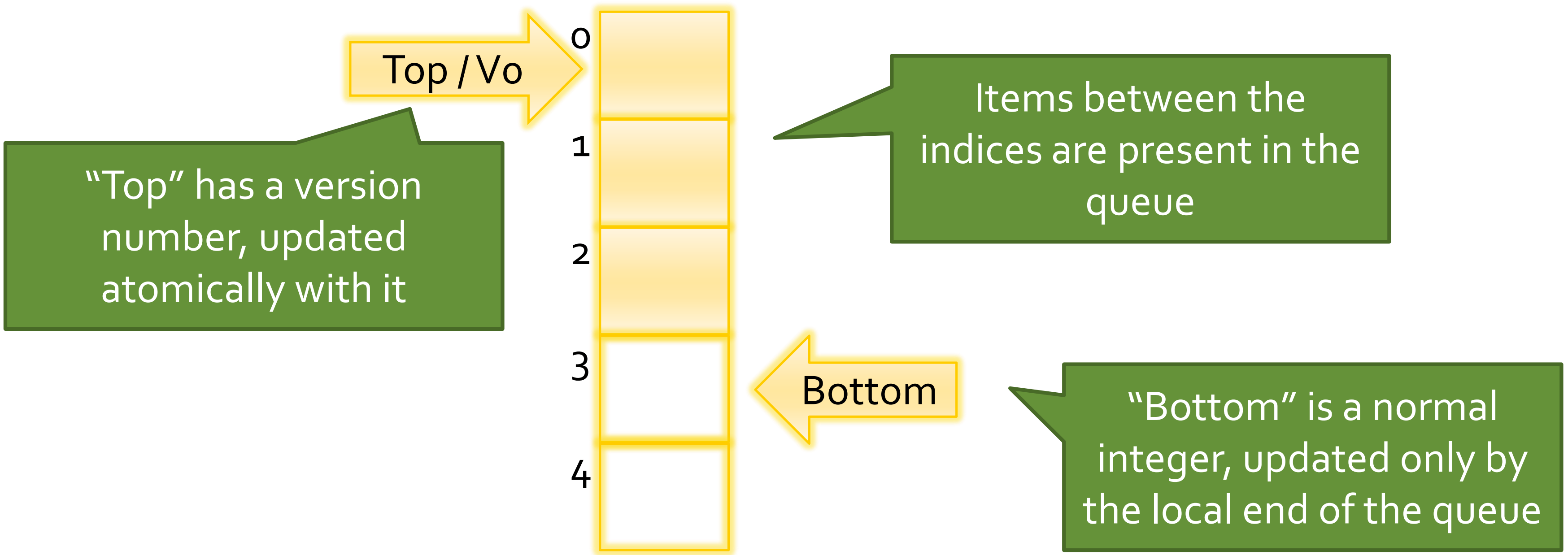


1. Semantics relaxed for “PopTop”

2. Restriction: only one thread ever calls “Push/PopBottom”

3. Implementation costs skewed toward “PopTop” complex

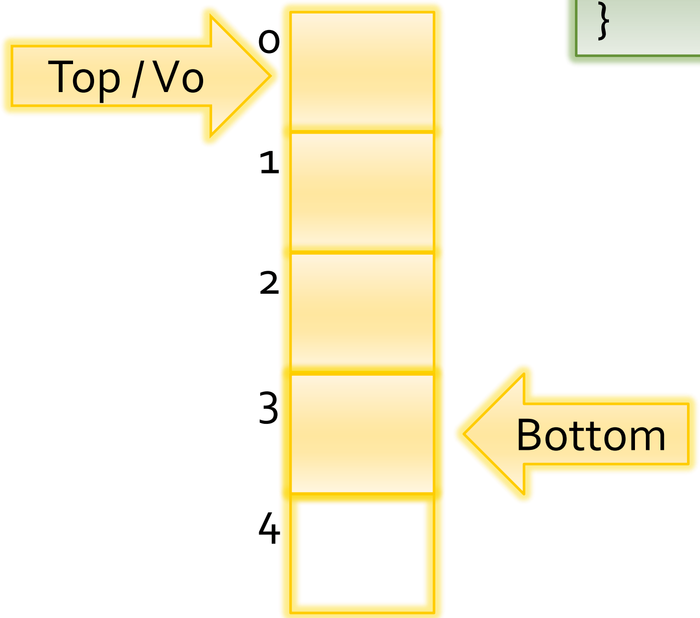
Bounded deque



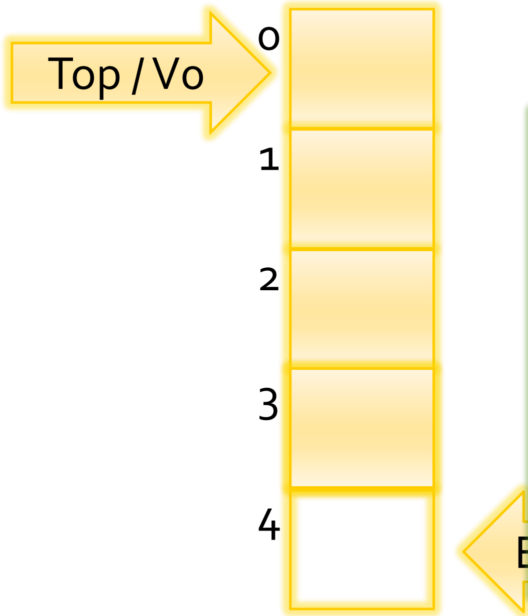
Arora, Blumofe, Plaxton

Bounded deque

```
void pushBottom(Item i){
   tasks[bottom] = i;
   bottom++;
}
```



Bounded deque

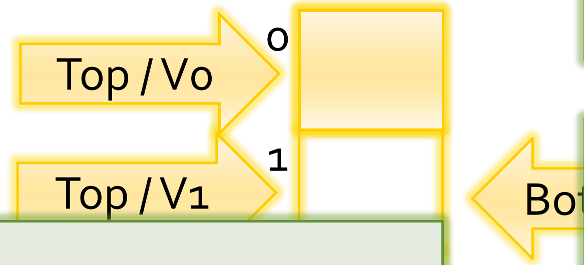



```
void pushBottom(Item i){
   tasks[bottom] = i;
   bottom++;
}
```

```
Item popBottom() {
  if (bottom ==0) return null;
  bottom--;
  result = tasks[bottom];
  <tmp_top,tmp_v> = <top,version>;
  if (bottom > tmp_top) return result;
  ….
  return null;
}
```

Bounded deque

```
void pushBottom(Item i){
   tasks[bottom] = i;
   bottom++;
}
```



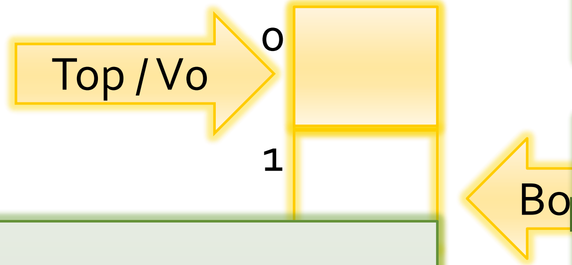

```
Item popTop() {
  if (bottom <= top) return null;
  <tmp_top,tmp_v> = <top, version>;
  result = tasks[tmp_top];
  if (CAS( &<top,version>,
              <tmp_top, tmp_v>,
              <tmp_top+1, tmp_v+1>)) {
    return result;
  }
  return null;
}
```

```
Item popBottom() {
  if (b
  bott
  resu
  <tm
  if (b
  ….
  retu
}
```

```
if (bottom==top) {
  bottom = 0;
  if (CAS( &<top,version>,
              <tmp_top,tmp_v>,
              <0,tmp_v+1>)) {
    return result;
  }
}
<top,version>=<0,v+1>
```

Bounded deque

```
void pushBottom(Item i){
   tasks[bottom] = i;
   bottom++;
}
```




```
Item popTop() {
  if (bottom <= top) return null;
  <tmp_top,tmp_v> = <top, version>;
  result = tasks[tmp_top];
  if (CAS( &<top,version>,
              <tmp_top, tmp_v>,
              <tmp_top+1, tmp_v+1>)) {
    return result;
  }
  return null;
}
```

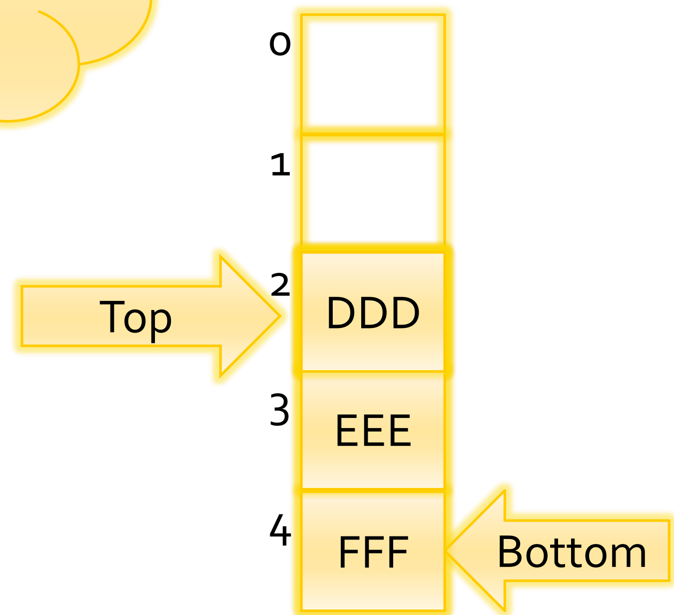
```
Item popBottom() {
  if (b
  bott
  resu
  <tm
  if (b
  ….
  retu
}
```

```
if (bottom==top) {
  bottom = 0;
  if (CAS( &<top,version>,
              <tmp_top,tmp_v>,
              <0,tmp_v+1>)) {
    return result;
  }
}
<top,version>=<0,v+1>
```

ABA problems

result = CCC

```
Item popTop() {
  if (bottom <= top) return nul
  tmp_top = top;
  result = tasks[tmp_top];
  if (CAS(&top, top, top+1)) {
      return result;
  }
  return null;
}
```



General techniques

- Local operations designed to avoid CAS
  - Traditionally slower, less so now
  - Costs of memory fences can be important ("Idempotent work stealing", Michael *et al*, and the "Laws of Order" paper)
- Local operations just use read and write
  - Only one accessor, check for interference
- Use CAS:
  - Resolve conflicts between stealers
  - Resolve local/stealer conflicts
  - Version number to ensure conflicts seen

Reducing contention

Reducing contention

- Suppose you're implementing a shared counter with the following sequential spec:

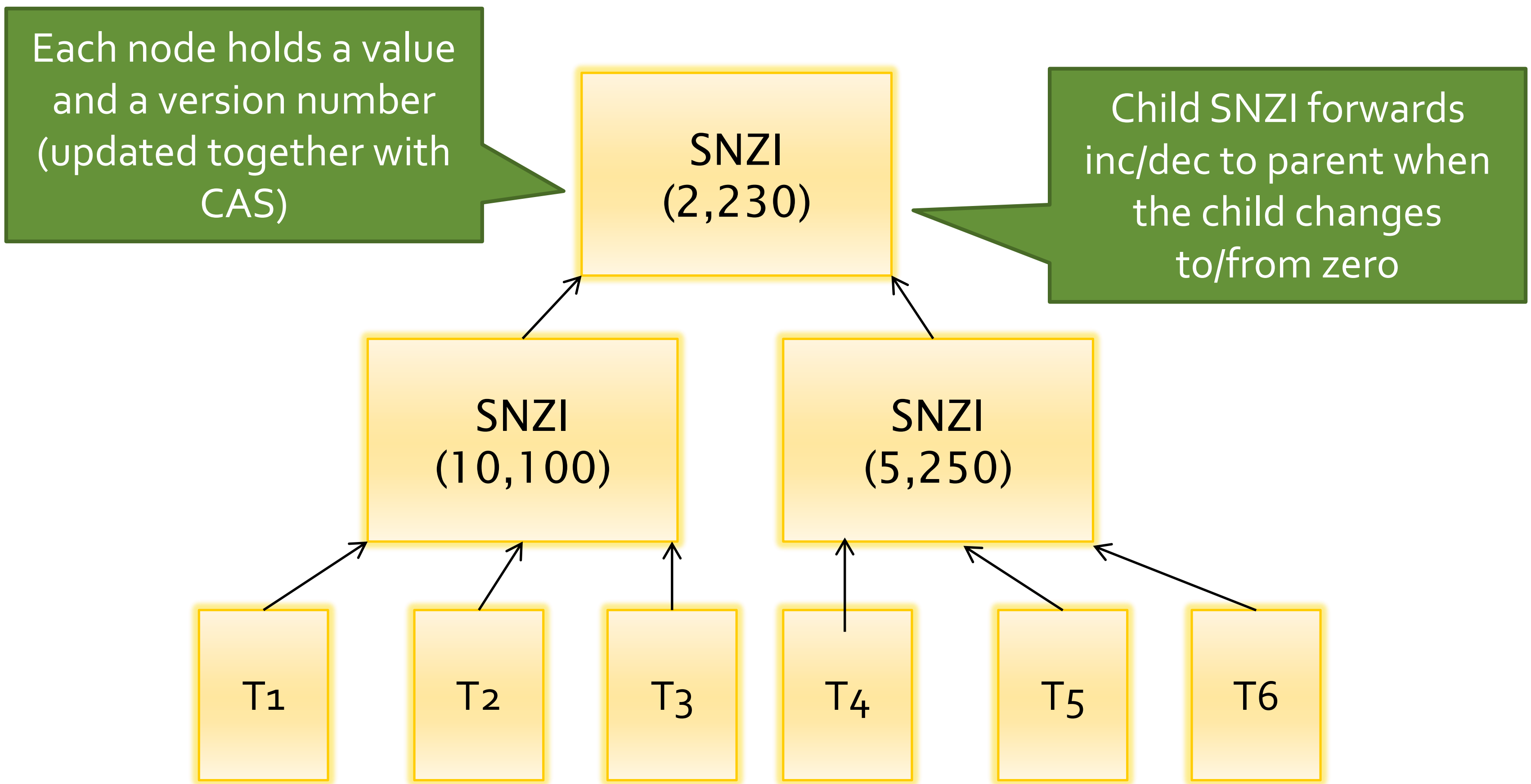
```
void increment(int *counter) {
  atomic {
     (*counter) ++;
  }
}
```

```
void decrement(int *counter) {
  atomic {
     (*counter) --;
  }
}
```

```
bool isZero(int *counter) {
  atomic {
     return (*counter) == 0;
  }
}
```

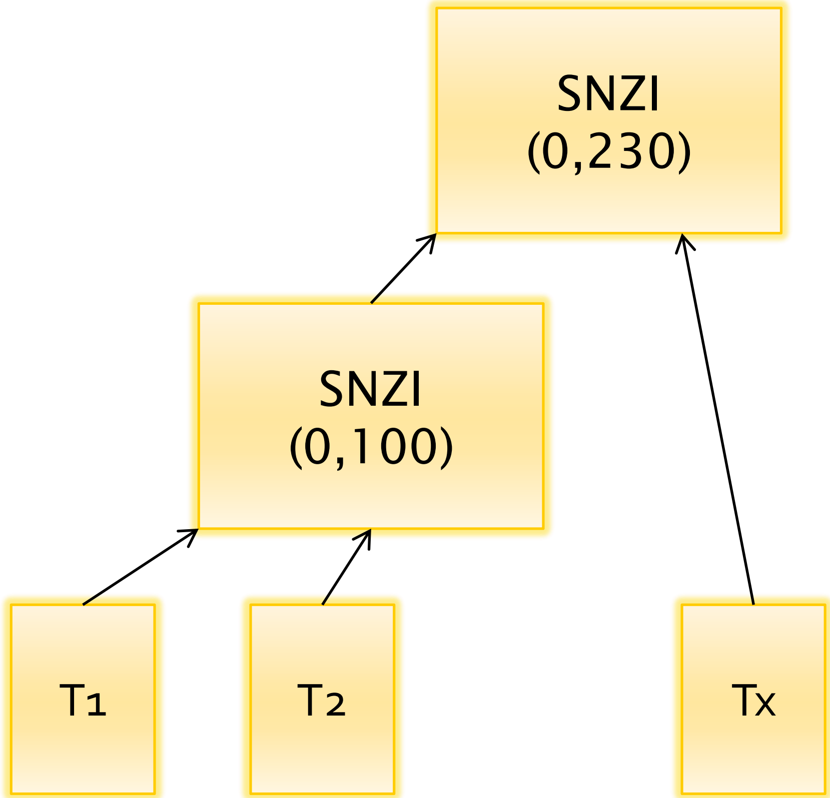
How well can this scale?

SNZI trees



SNZI: Scalable NonZero Indicators, Ellen et al

SNZI trees, linearizability on 0->1 change

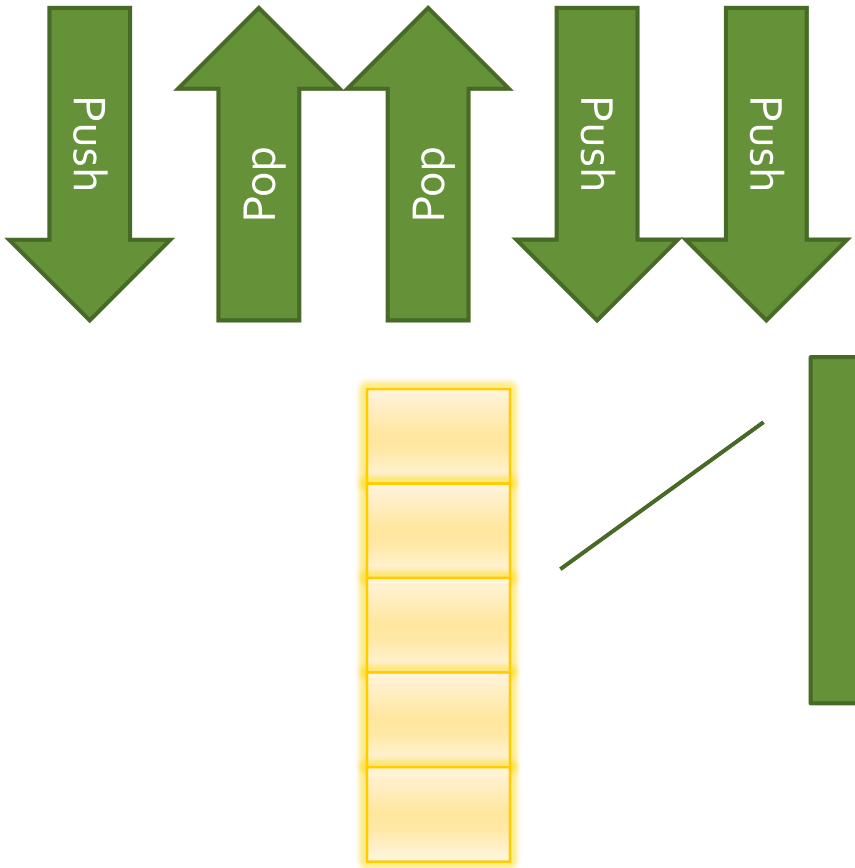


1. T1 calls increment
2. T1 increments child to 1
3. T2 calls increment
4. T2 increments child to 2
5. T2 completes
6. Tx calls isZero
7. Tx sees 0 at parent
8. T1 calls increment on parent
9. T1 completes

SNZI trees

```
void increment(snzi *s) {
   bool done=false;
   int undo=0;
   while(!done) {
      <val,ver> = read(s->state);
      if (val >= 1 && CAS(s->state, <val,ver>, <val+1,ver>)) { done = true; }
      if (val == 0 && CAS(s->state, <val,ver>, <½, ver+1>)) {
          done = true;  val=½; ver=ver+1
      }
      if (val == ½) {
          increment(s->parent);
          if (!CAS(s->state, <val, ver>, <1, ver>)) { undo ++; }
      }
   }
   while (undo > 0) {
      decrement(s->parent);
   }
}
```

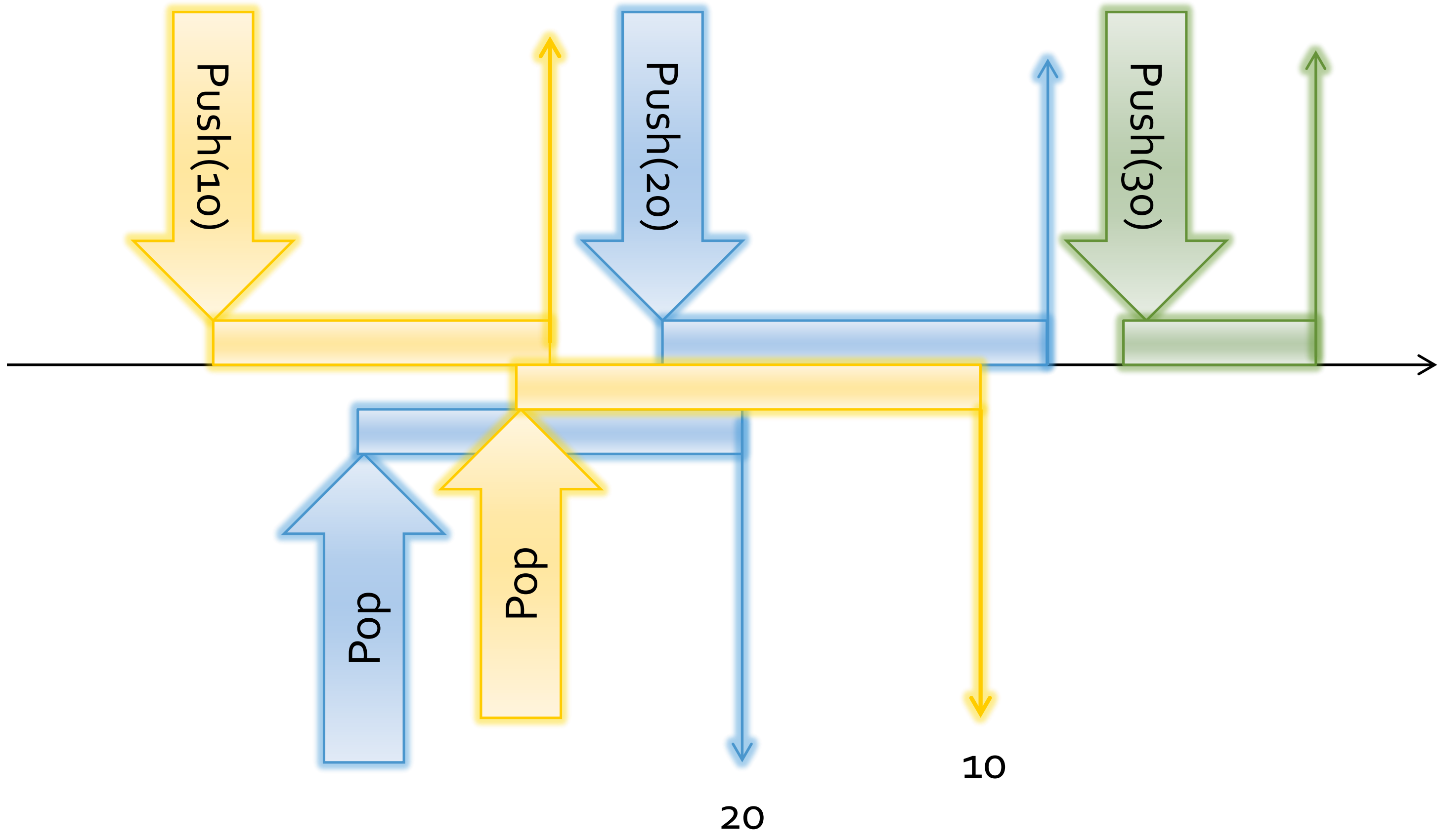
Reducing contention: stack



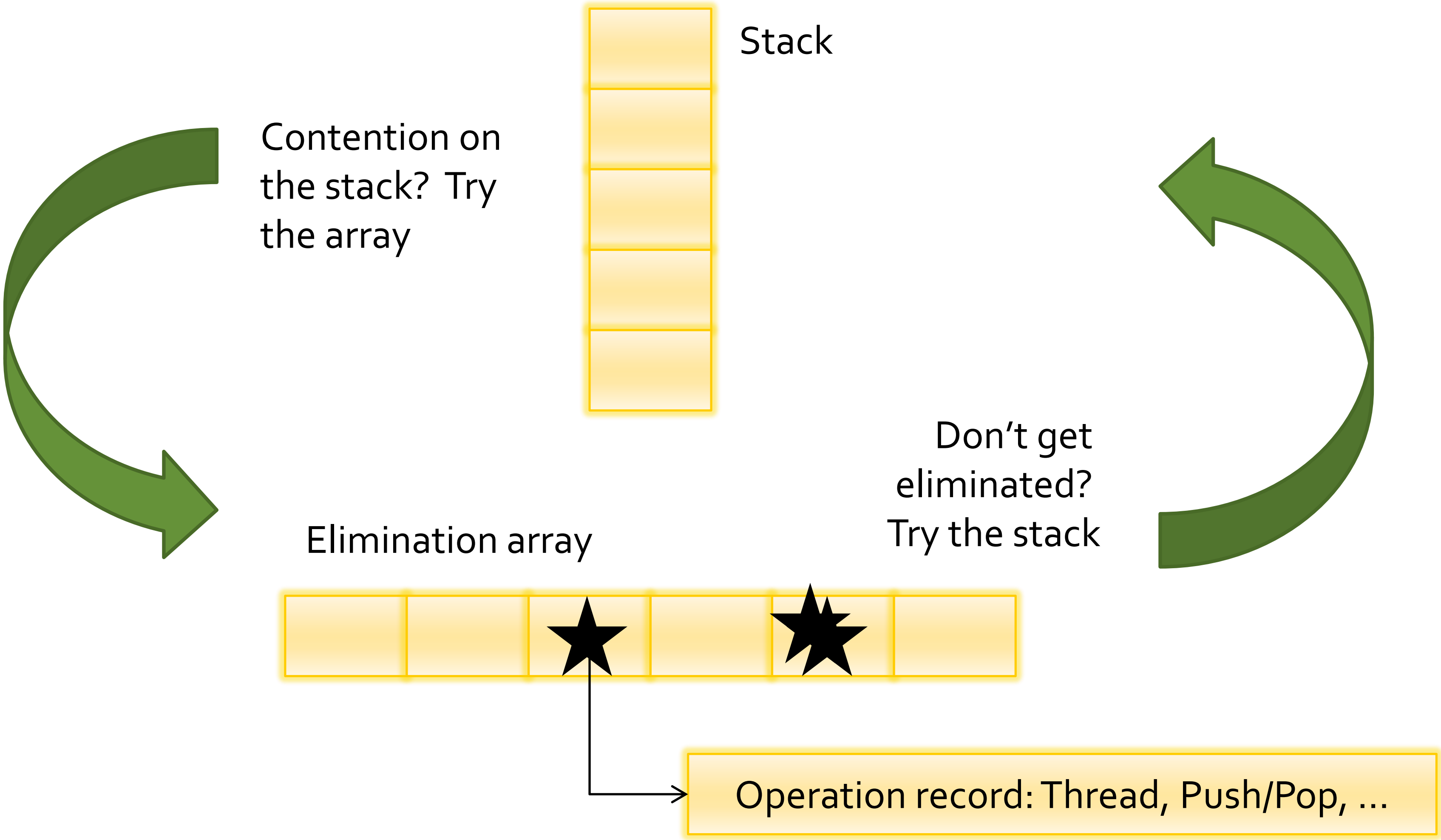
Existing lock-free stack (e.g., Treiber's): good performance under low contention, poor scalability

A scalable lock-free stack algorithm, Hendler et al

Pairing up operations

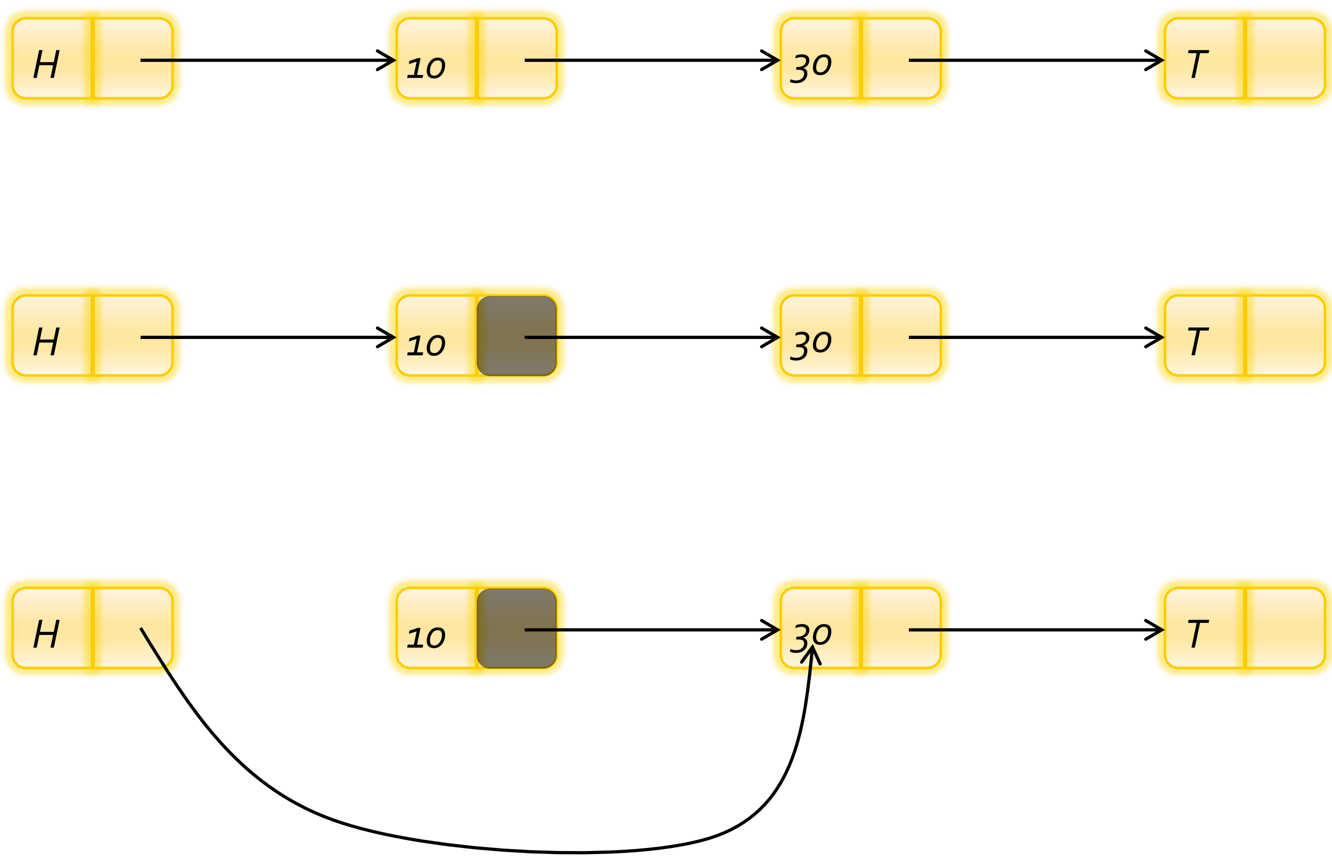


Back-off elimination array

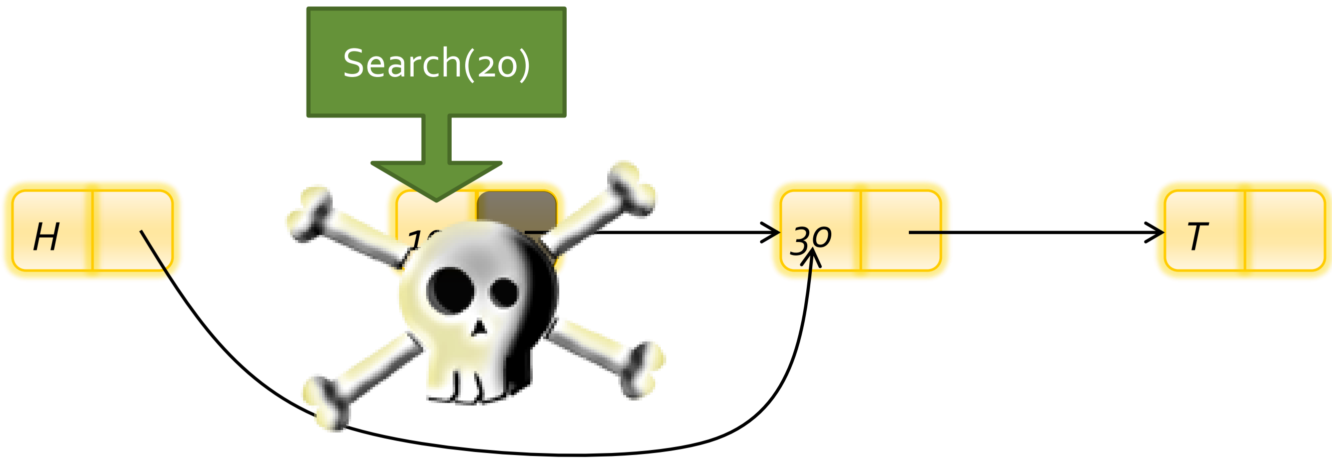


Explicit memory management

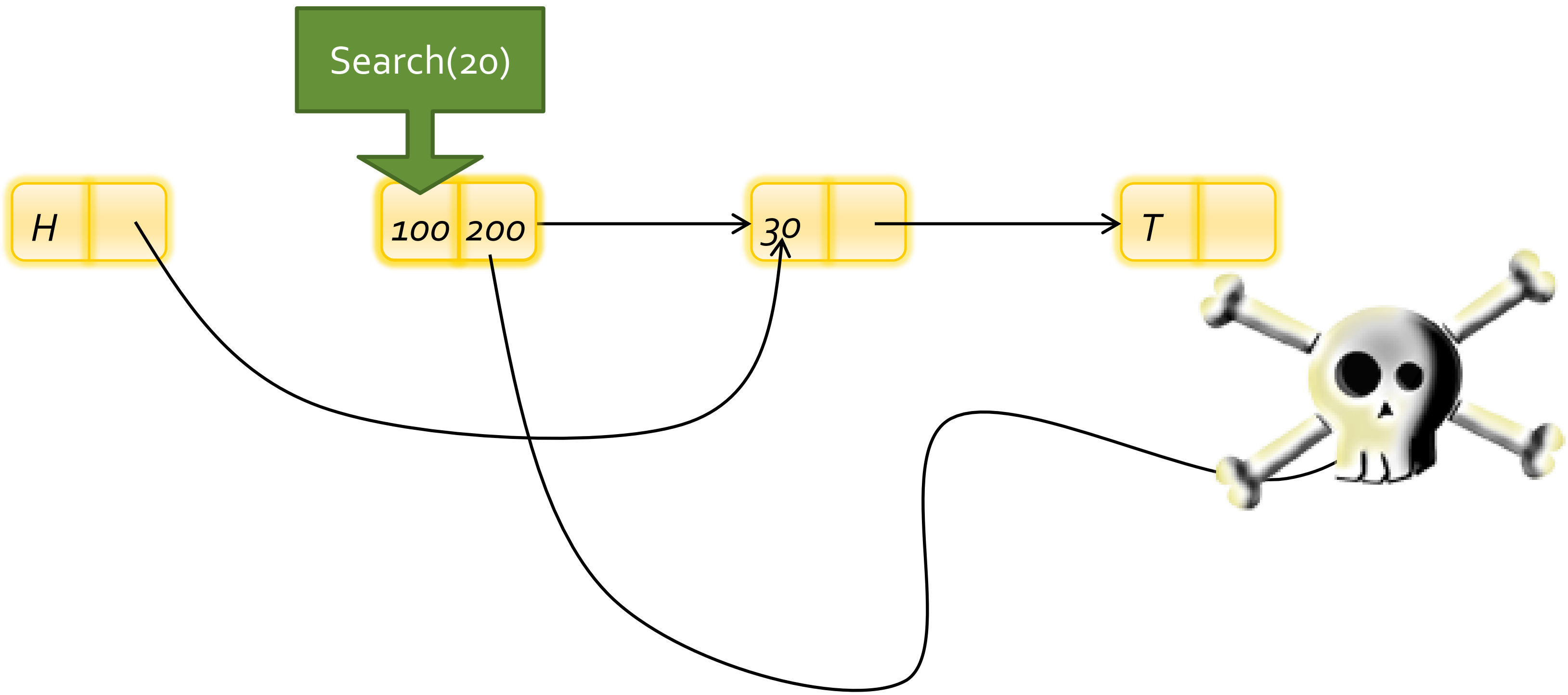
Deletion revisited: Delete(10)



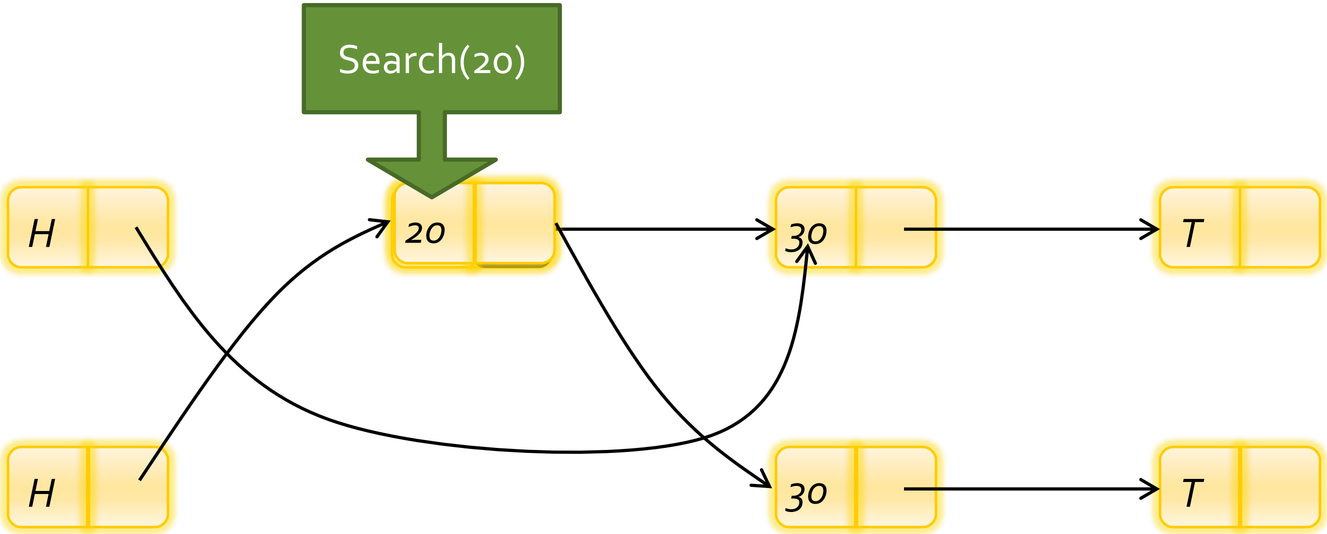
De-allocate to the OS?



Re-use as something else?

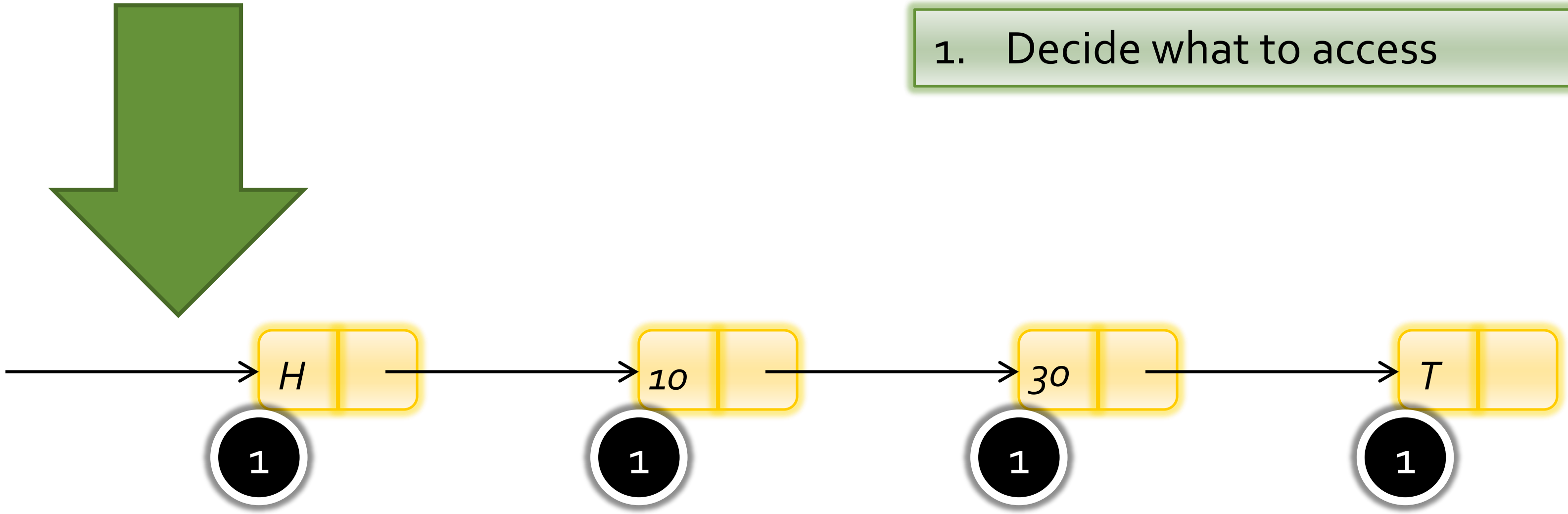


Re-use as a list node?



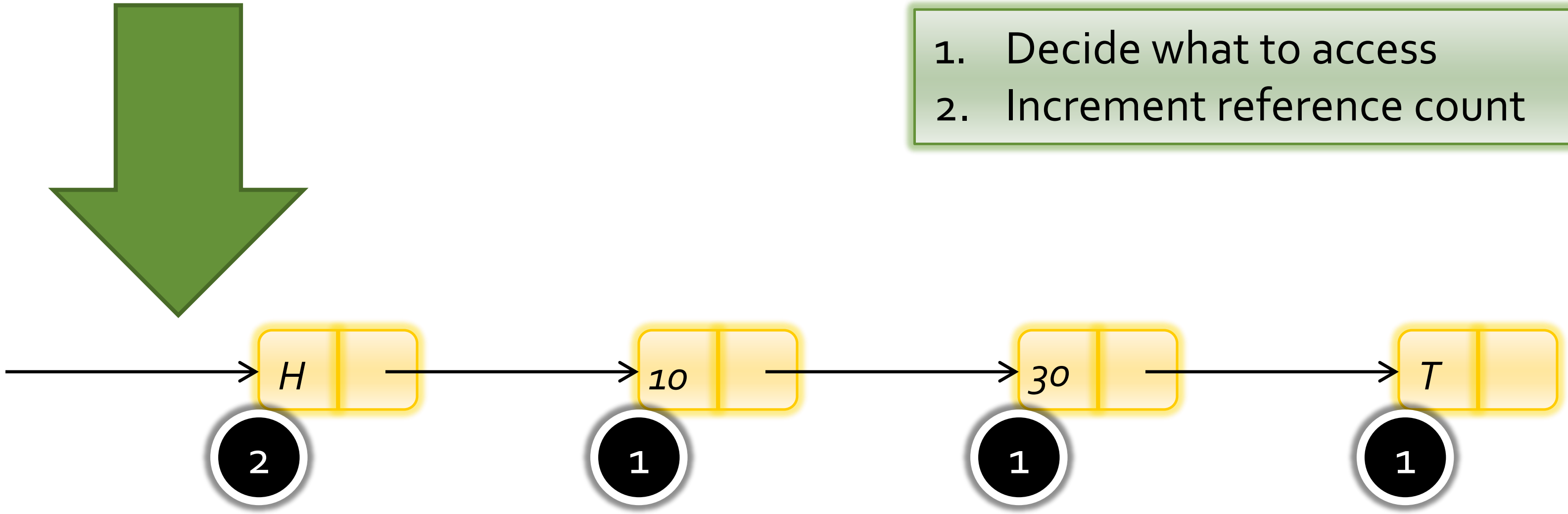
Reference counting

1. Decide what to access



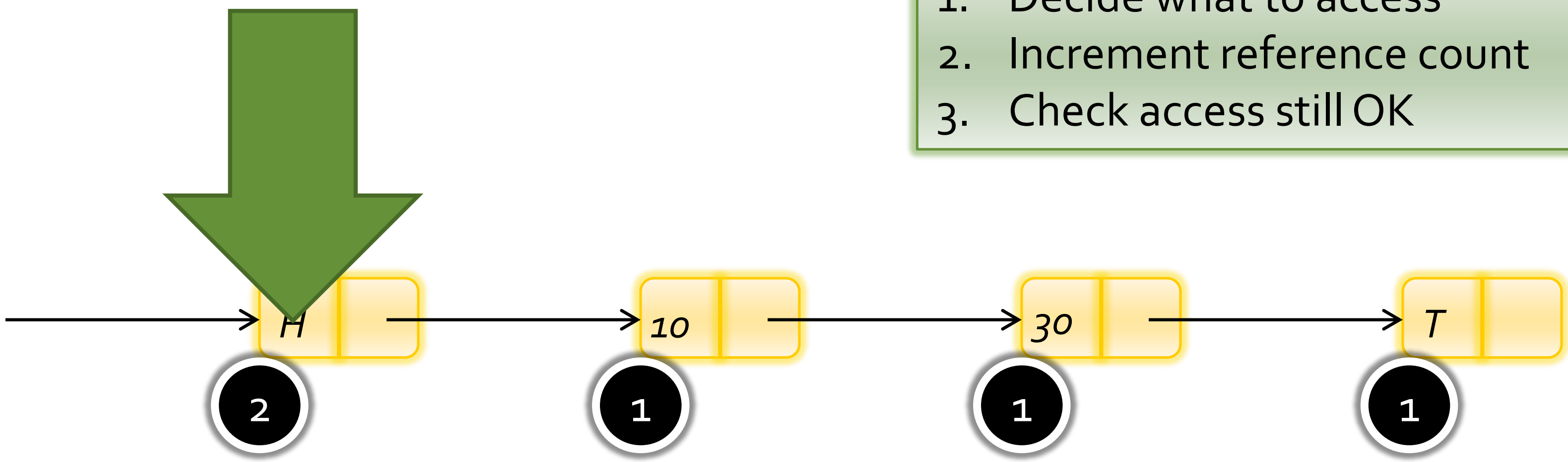
Reference counting

1. Decide what to access
2. Increment reference count



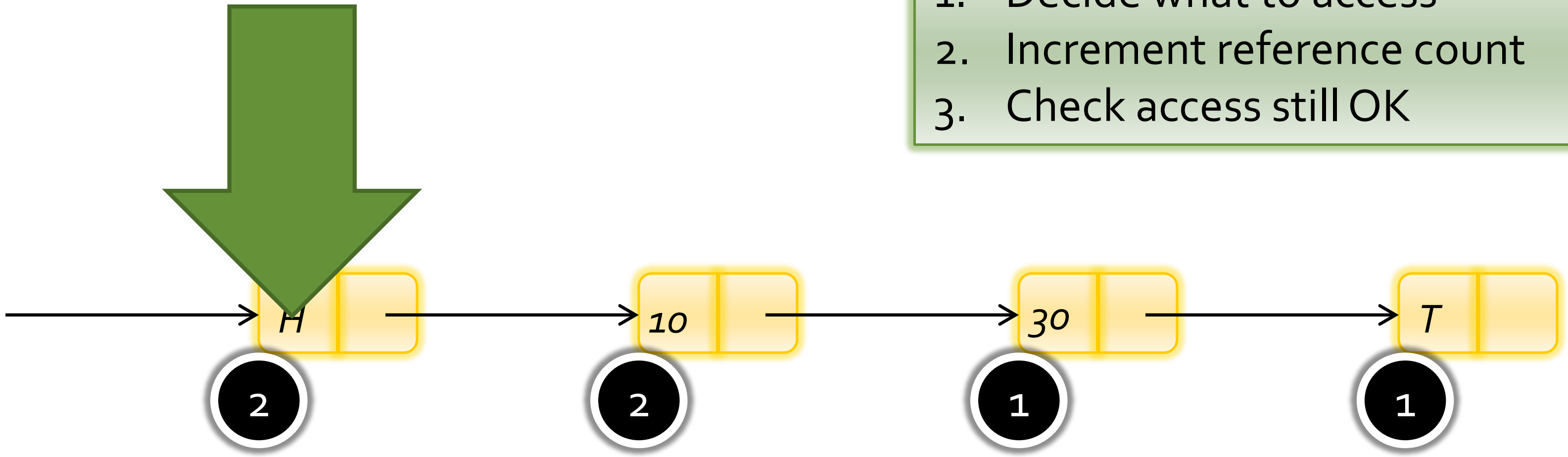
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK



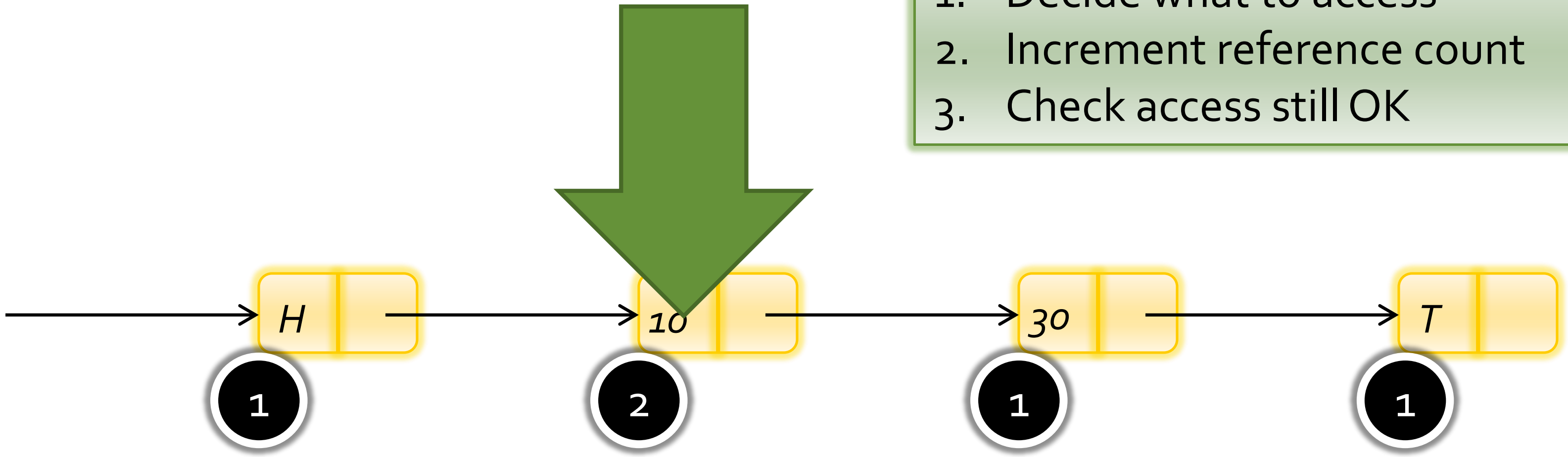
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK



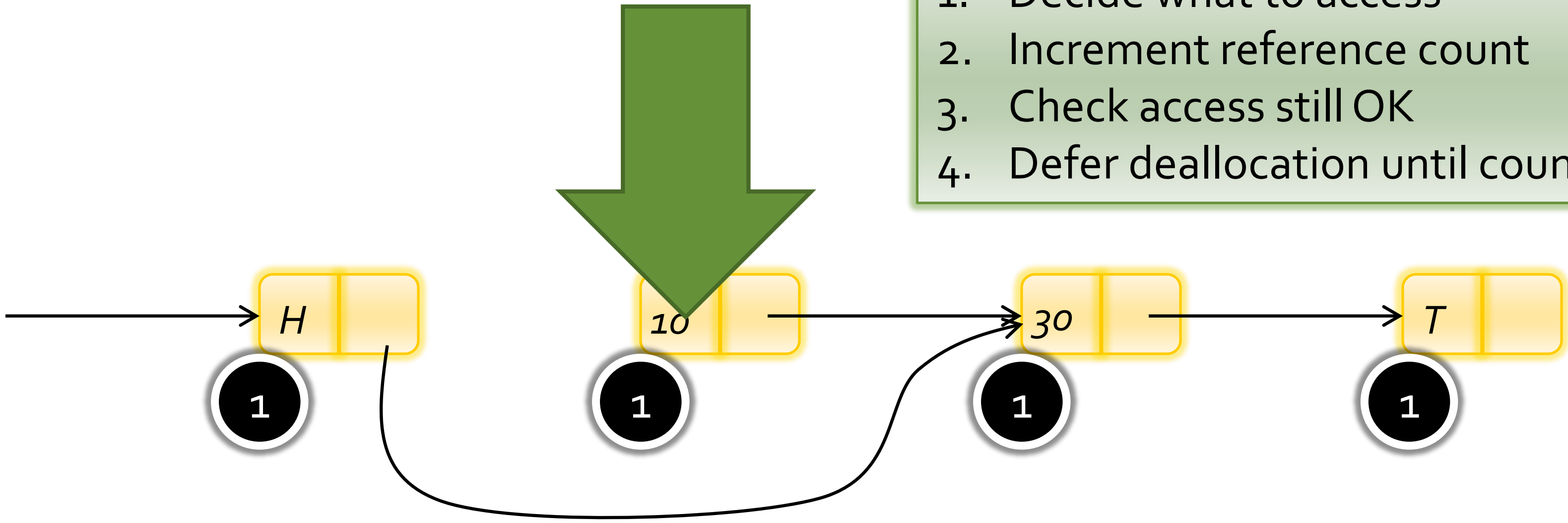
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK



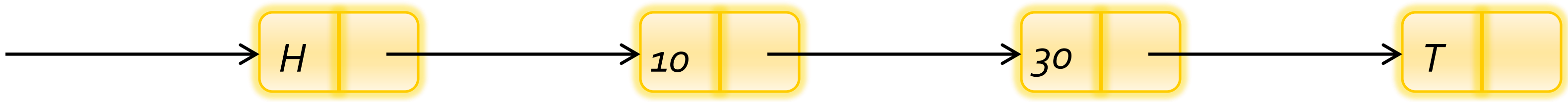
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK
4. Defer deallocation until count 0



Epoch mechanisms

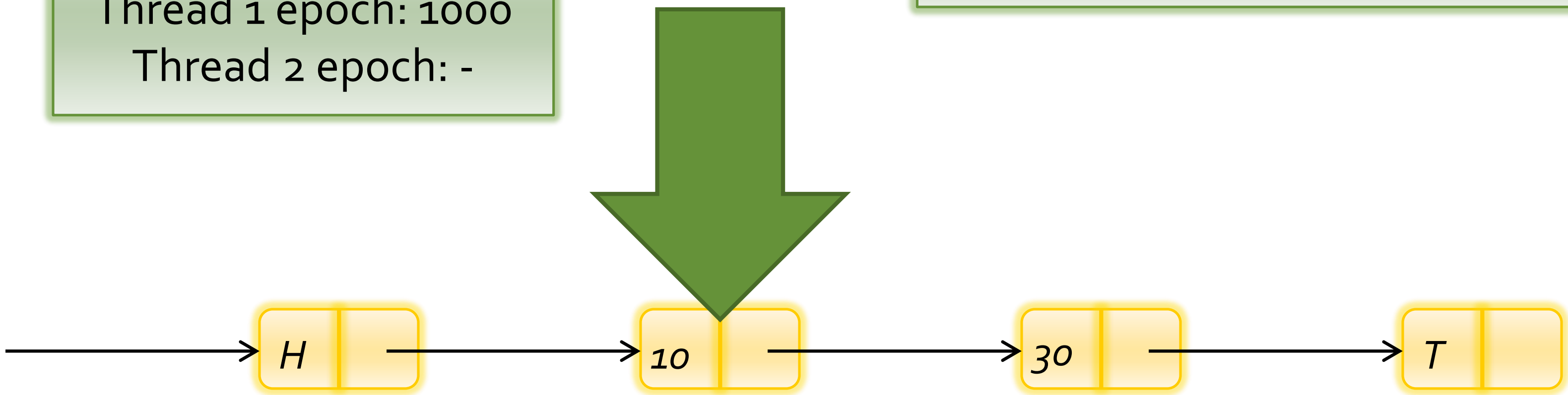
Global epoch: 1000
Thread 1 epoch: -
Thread 2 epoch: -



Epoch mechanisms

Global epoch: 1000
Thread 1 epoch: 1000
Thread 2 epoch: -

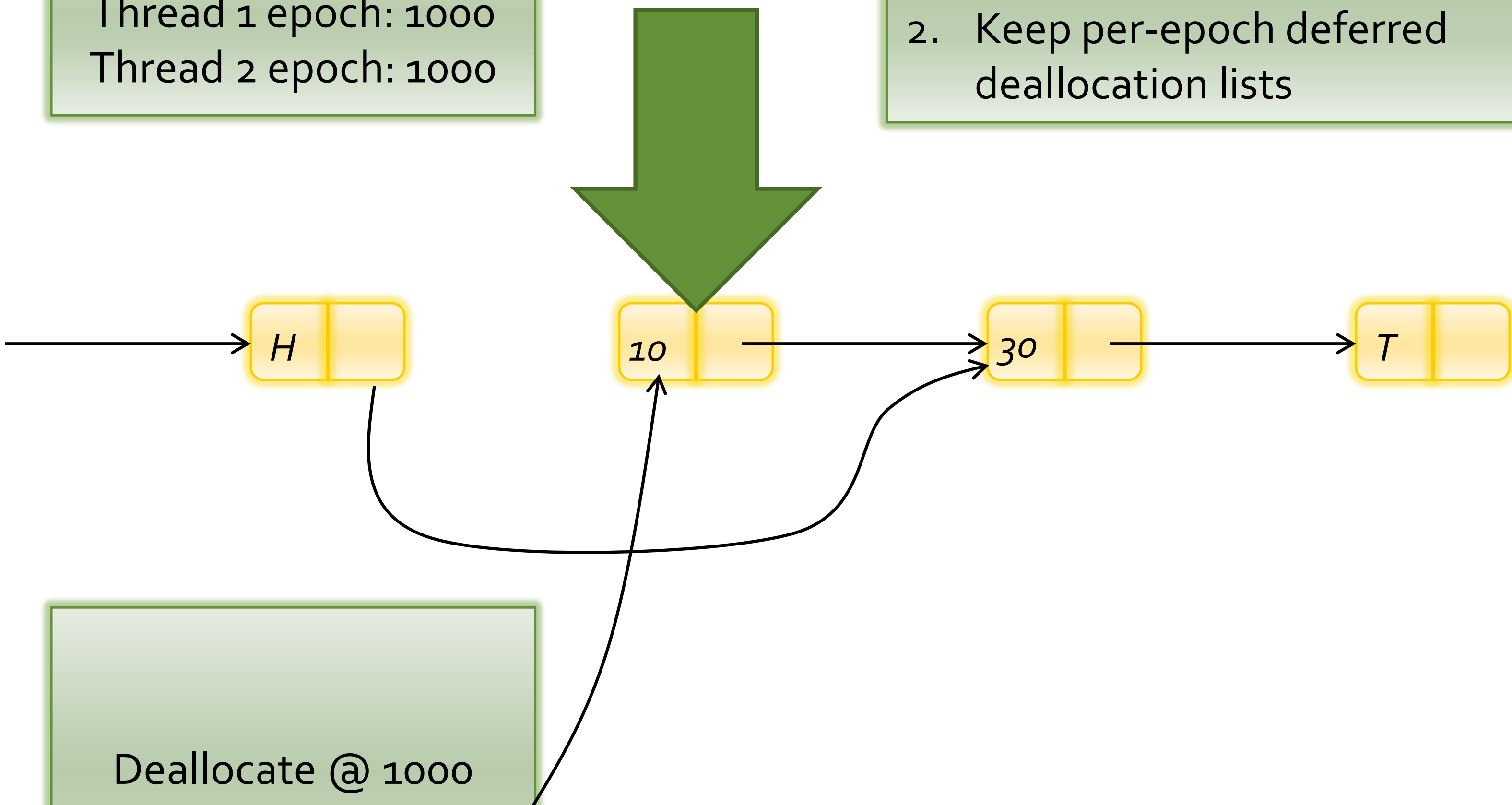
1. Record global epoch at start of operation



Epoch mechanisms

Global epoch: 1000
Thread 1 epoch: 1000
Thread 2 epoch: 1000

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists

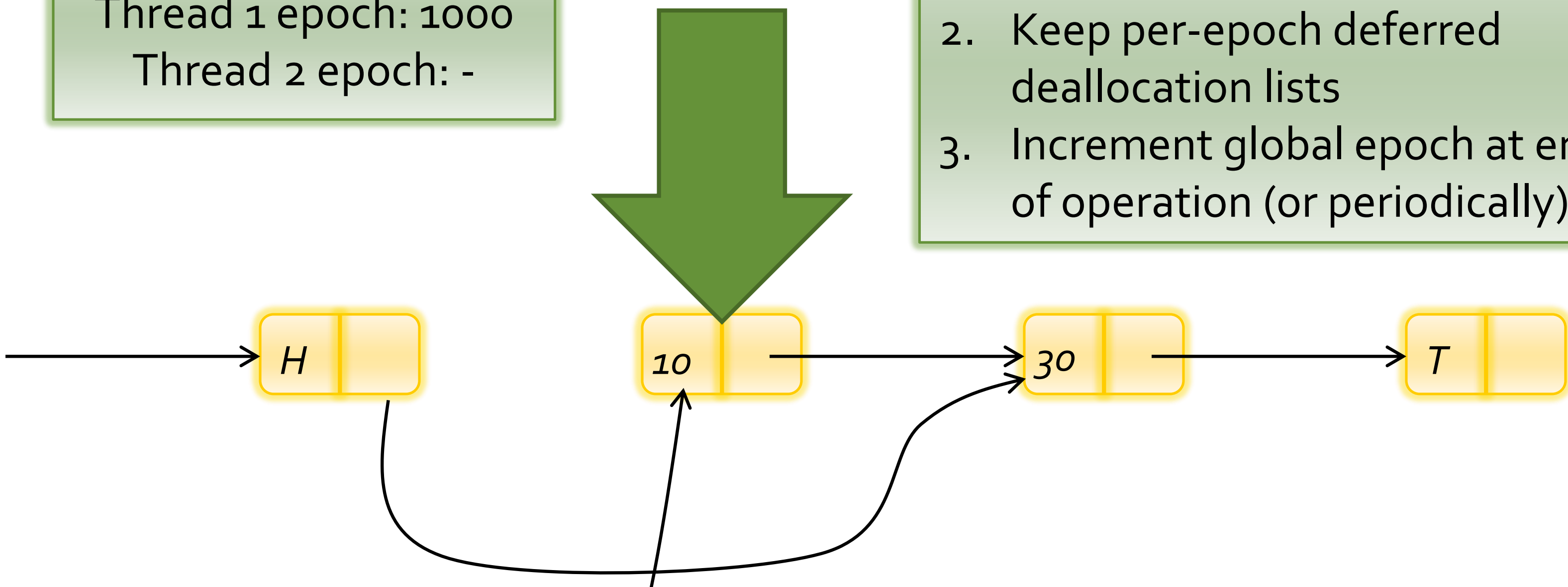


Deallocate @ 1000

Epoch mechanisms

Global epoch: 1001
Thread 1 epoch: 1000
Thread 2 epoch: -

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)

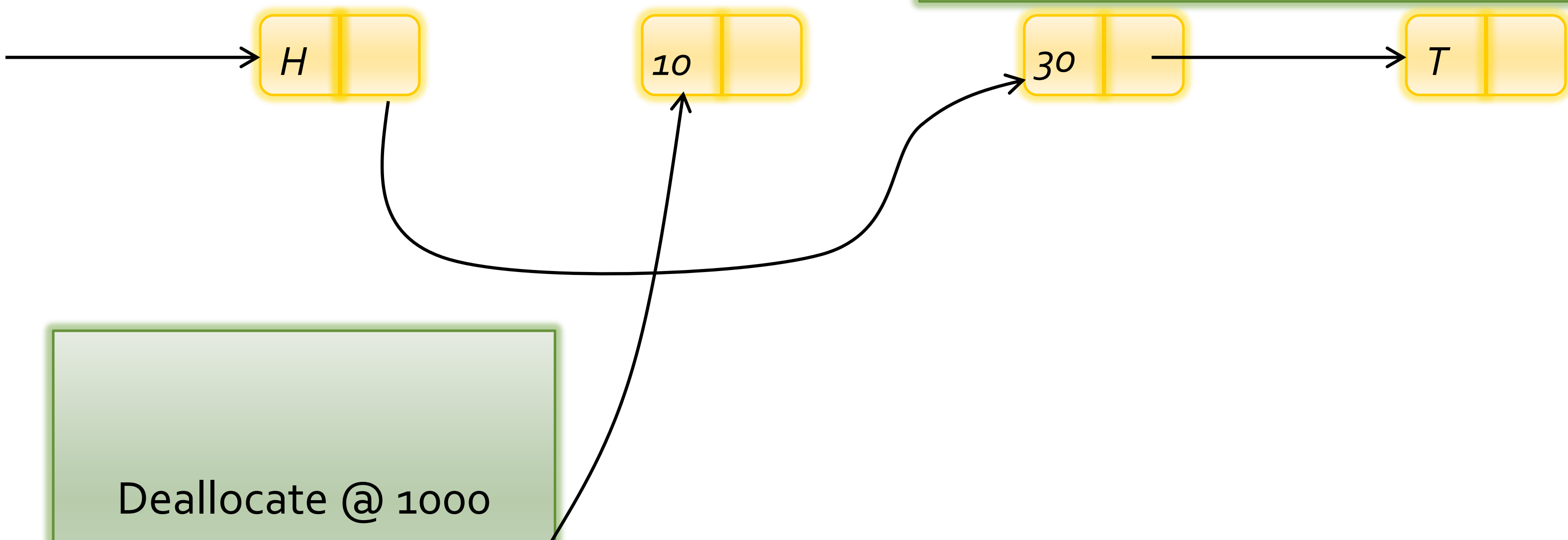


Deallocate @ 1000

Epoch mechanisms

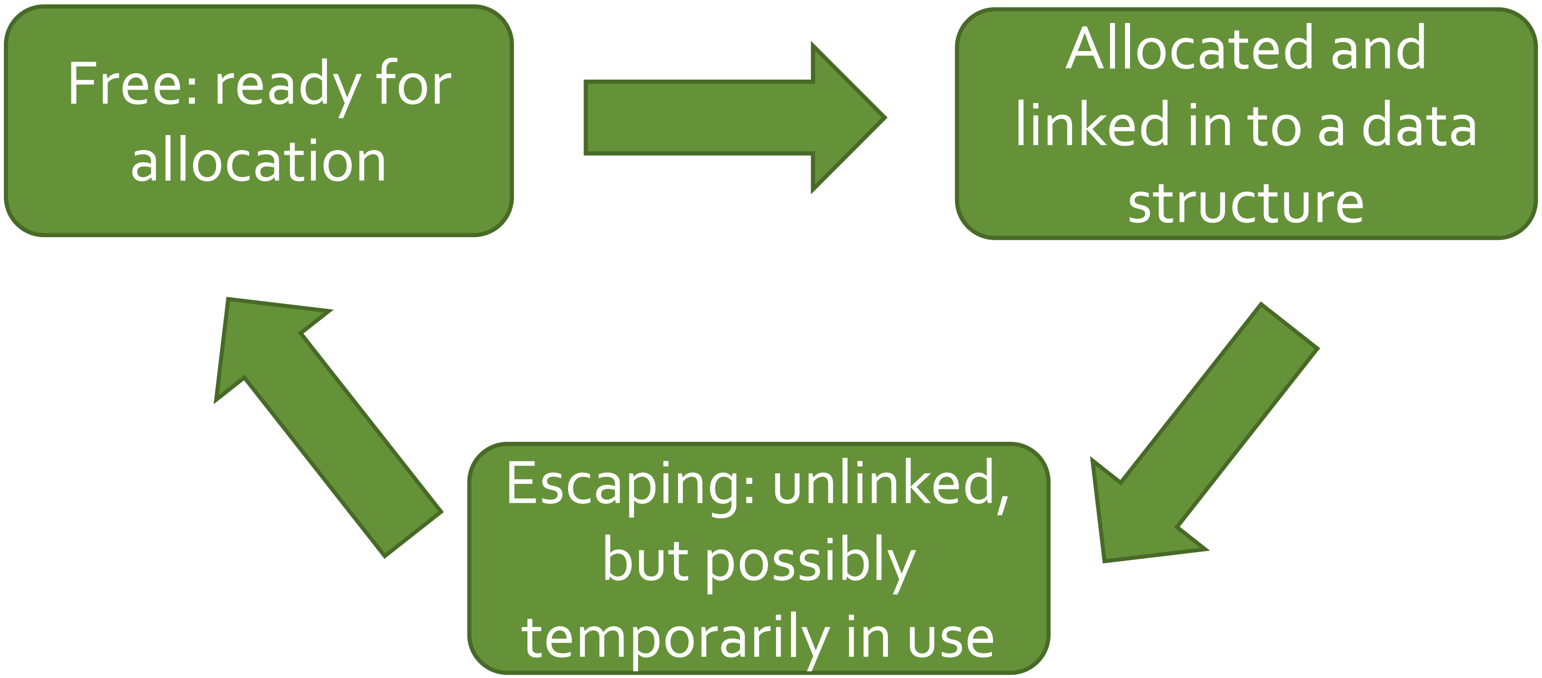
Global epoch: 1002
Thread 1 epoch: -
Thread 2 epoch: -

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)
4. Free when everyone past epoch



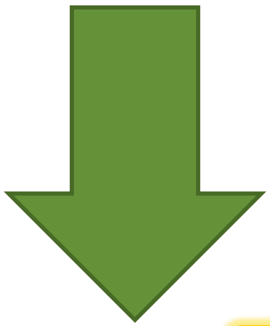
Deallocate @ 1000

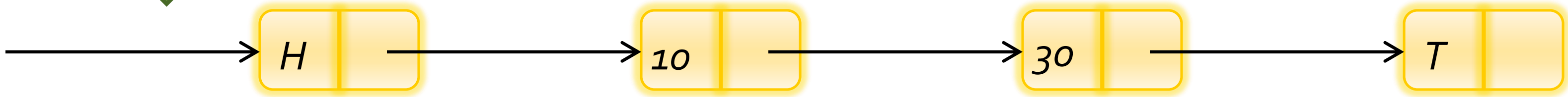
The "repeat offender problem"



Re-use via ROP

1. Decide what to access
2. Set guard
3. Check access still OK

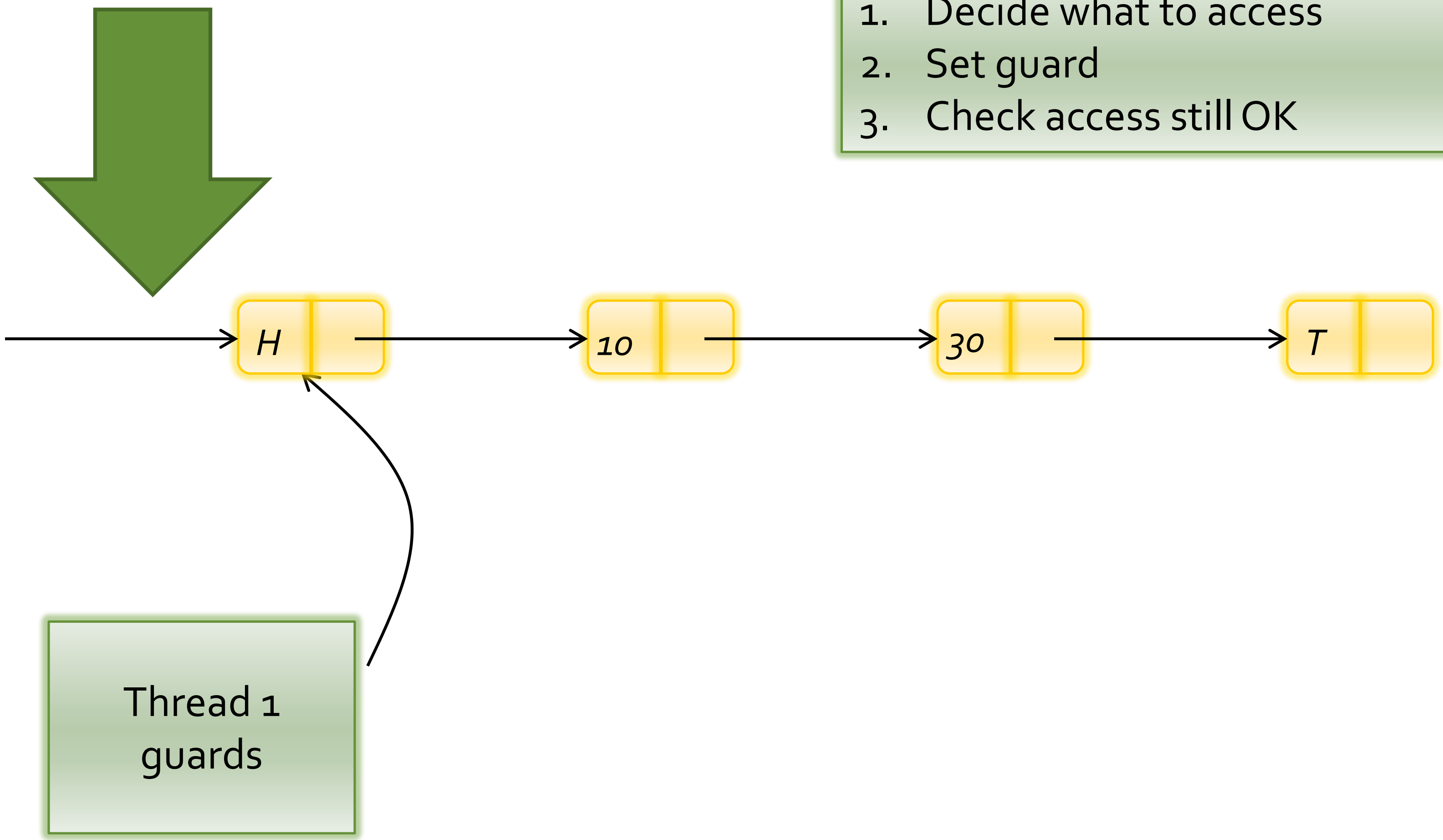




Thread 1 guards

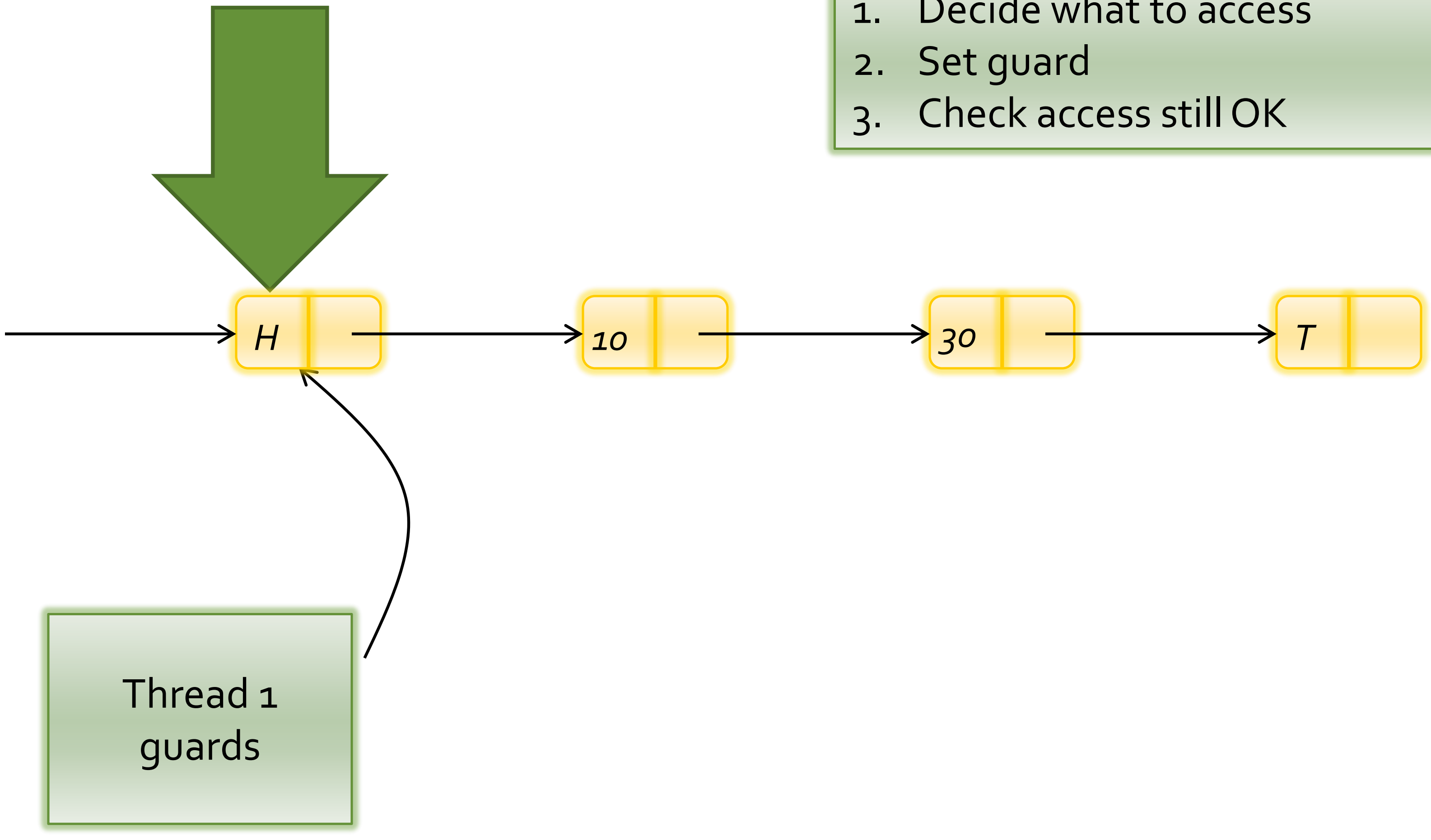
Re-use via ROP

1. Decide what to access
2. Set guard
3. Check access still OK



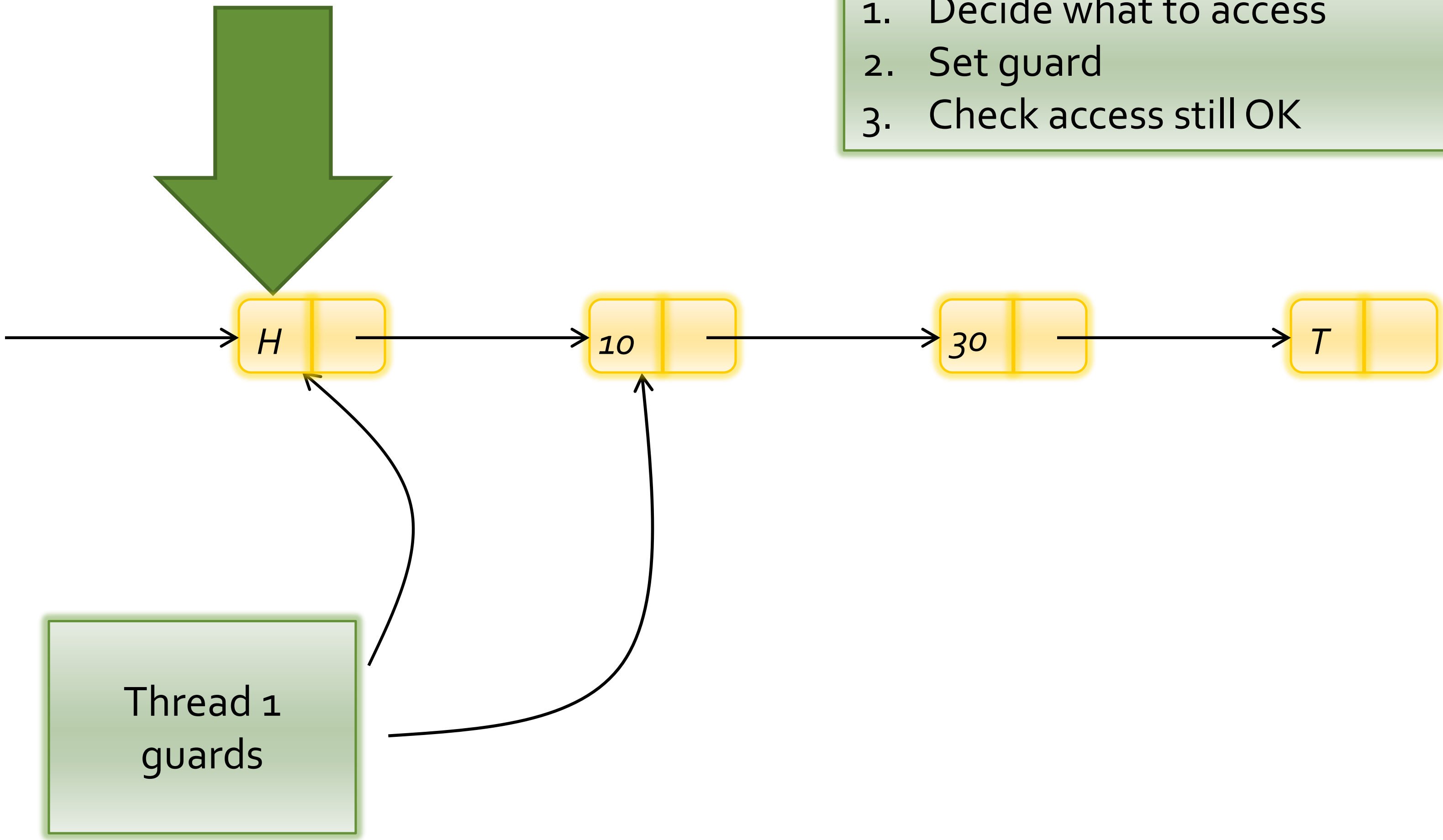
Re-use via ROP

1. Decide what to access
2. Set guard
3. Check access still OK

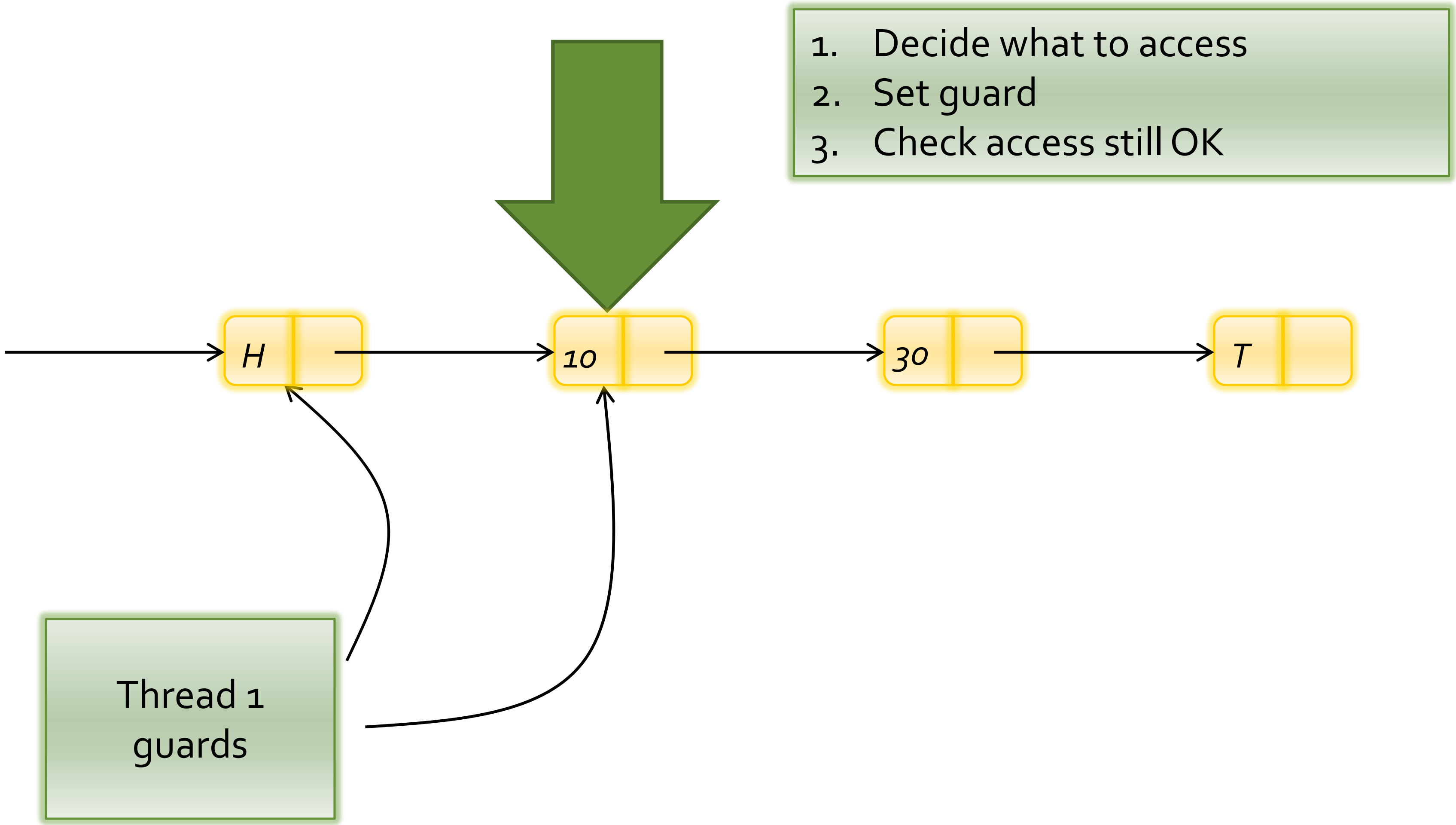


Re-use via ROP

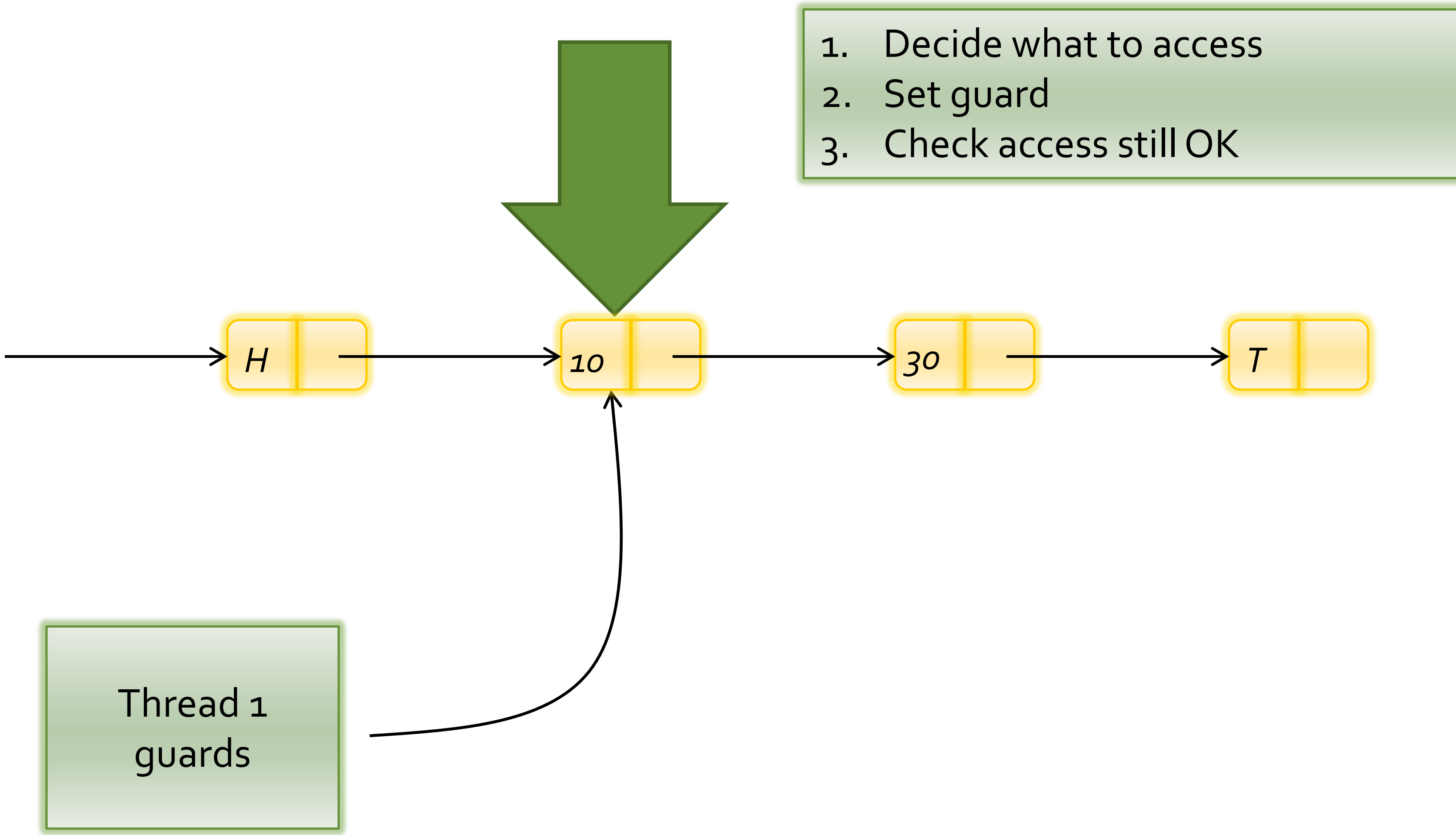
1. Decide what to access
2. Set guard
3. Check access still OK



Re-use via ROP



Re-use via ROP



Re-use via ROP

See also: "Safe memory reclamation" & hazard pointers, Maged Michael

3. Chec… s still OK
4. Batch deallocations and defer on objects while guards are present