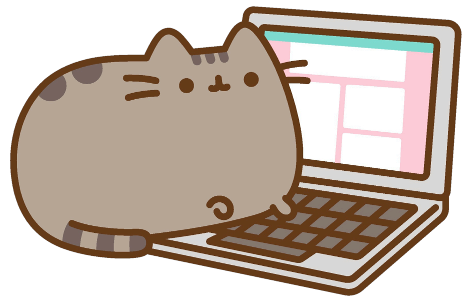
Announcements

- Office hours in announcement on Canvas
- Pull updates to HW2 with `git pull` (should not affect the part you are changing, since it is in the test driver)
- To zip up a directory, use `-r`, e.g., `zip -r hw2.zip hw2/`

CSE 6230:
HPC Tools and Applications


+


Lecture 7: Locking and Nondeterminism

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Determinism

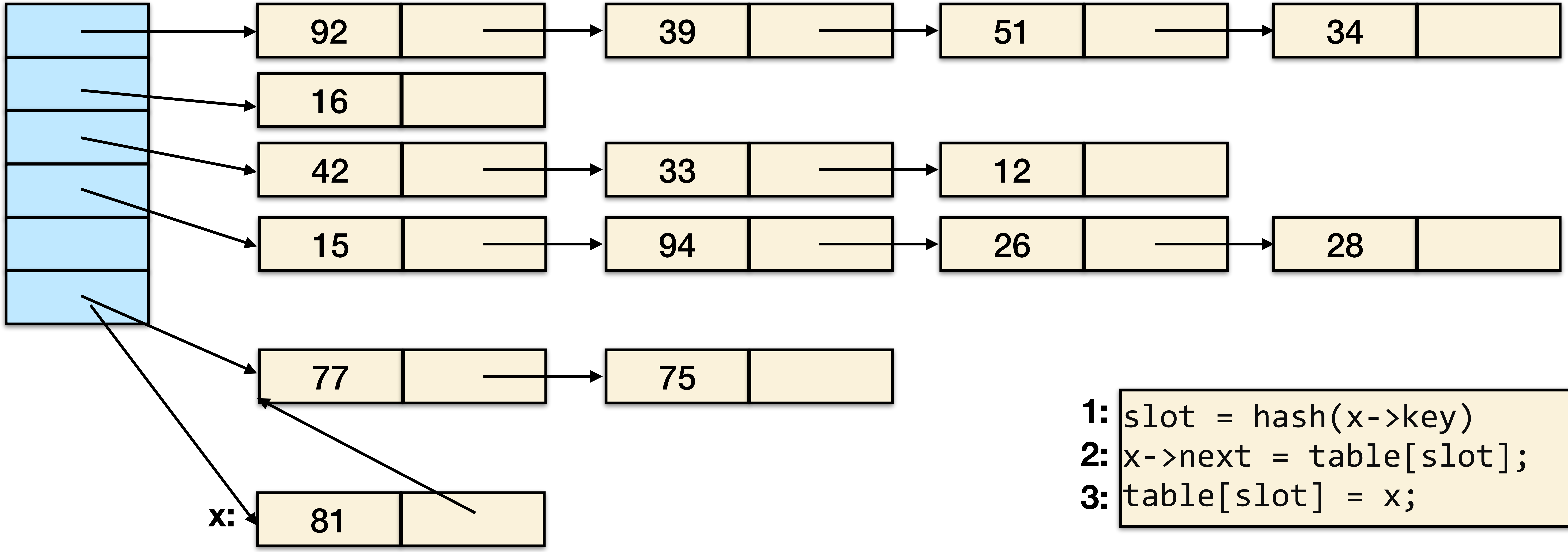
Definition. A program is **deterministic** on a given input if every memory location is updated with the same sequence of values in every execution.

- The program always behaves the same way.
- Two different memory locations may be updated in different orders, but each location always sees the same sequence of updates.

Advantage: Debugging!

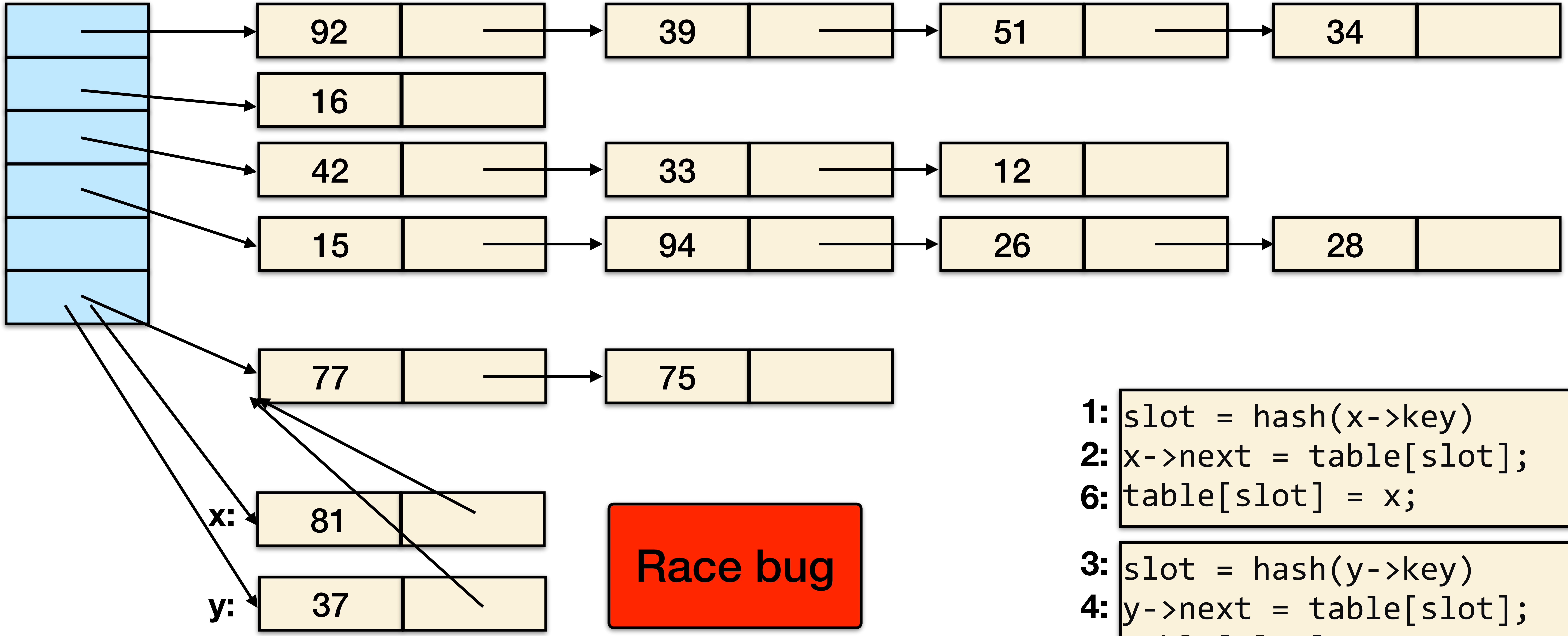
Mutual Exclusion & Atomicity

Hash Table Example




```
1: slot = hash(x->key)
2: x->next = table[slot];
3: table[slot] = x;
```

Cocurrent Hash Table Example




```
1: slot = hash(x->key)
2: x->next = table[slot];
6: table[slot] = x;

3: slot = hash(y->key)
4: y->next = table[slot];
5: table[slot] = y;
```

From MIT OCW 6.172

Atomicity

A sequence of instructions is **atomic** if the rest of the system cannot ever view them as partially executed. At any moment, either no instructions in the sequence have executed or all have executed.

A **critical section** is a piece of code that accesses a shared data structure that must not be accessed by two or more threads at the same time (**mutual exclusion**).

```
lock_t lock;

parallel_for ( i = 0; i < n; i++ ) {
  set_lock(&lock);

  // CRITICAL SECTION

  unset_lock(&lock);
}
```

Mutexes

A **mutex** is an object with `lock` and `unlock` member functions. An attempt by a thread to lock an already locked mutex causes that thread to **block** (i.e., wait) until the mutex is unlocked.

Modified code: Each slot is a `struct` with a mutex `L` and a pointer `head` to the slot contents.

```
slot = hash(x->key);
lock(&table[slot].L);
  x->next = table[slot].head;
  table[slot].head = x;
unlock(&table[slot].L);
```

Critical section

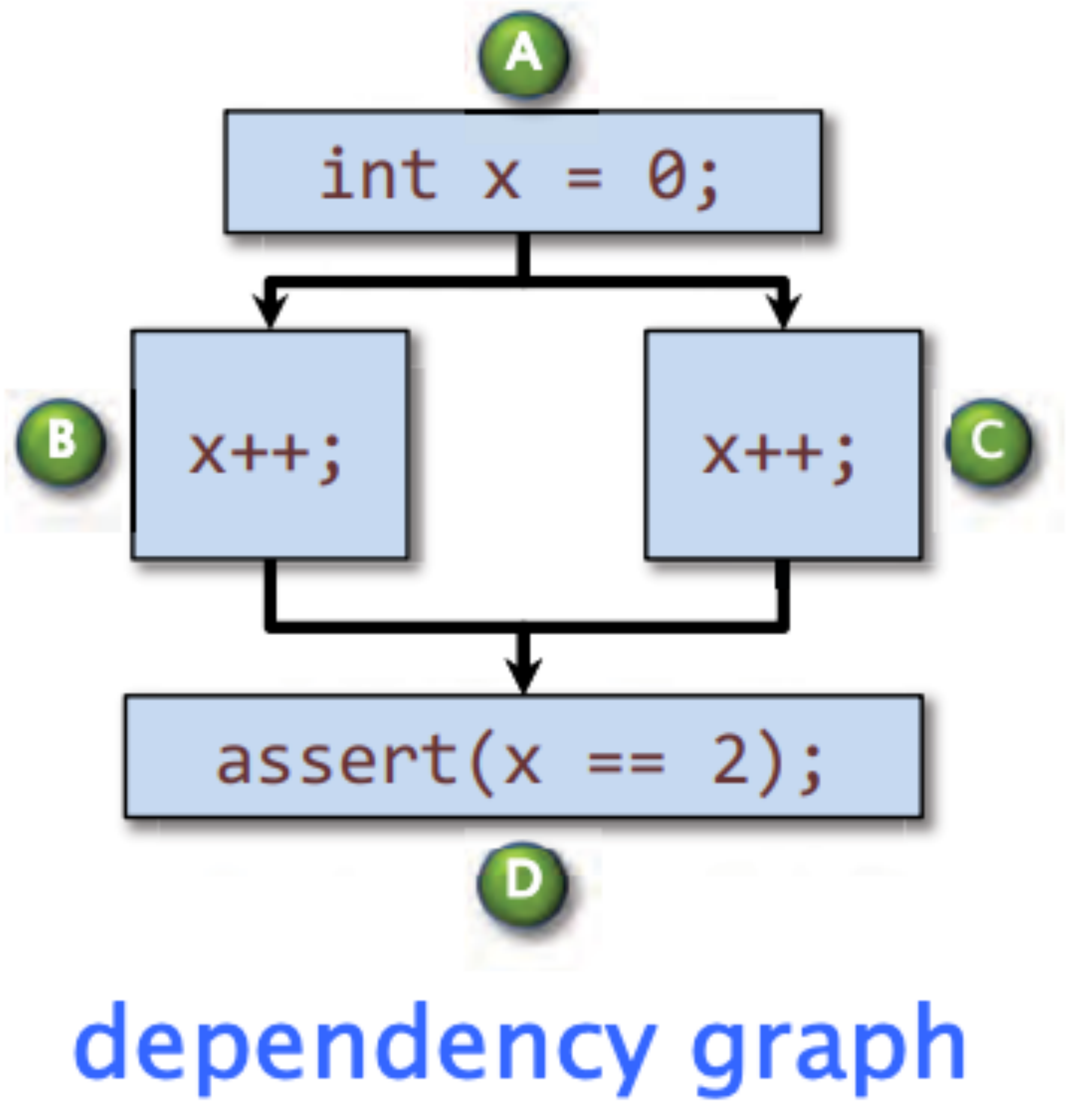
Mutexes can be used to implement atomicity.

Recall: Determinacy Races

A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

A program execution with no determinacy races means that the program always **behaves the same on a given input**, no matter how it is scheduled and executed.

A	`int x = 0;`
	`#pragma omp parallel for`
	`for (i = 0; i < 2; i++) {`
B, C	`  x++;`
	`}`
D	`assert(x == 2);`



Data Races

A **data race** occurs when two logically parallel instructions **holding no locks in common** access the same memory location and at least one of the instructions performs a write.

Although data-race-free programs obey atomicity constraints, they can still be nondeterministic, because acquiring a lock can cause a determinacy race with another lock acquisition.

No Data Races =/= No Bugs

Example:

```
slot = hash(x->key);
lock(&table[slot].L);
 x->next = table[slot].head;
unlock(&table[slot].L);

lock(&table[slot].L);
 table[slot].head = x;
unlock(&table[slot].L);
```

Nevertheless, the presence of mutexes and the absence of data races at least means that the programmer thought about the issue.

"Benign" Races

Example: Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```
for (int i=0; i<10; ++i) {
  digits[i] = 0;
}

parallel_for (int i = 0; i < N; ++i) {
  digits[A[i]] = 1;
}
```

Benign race

digits:

1	1	0	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9

Caution: This code only works correctly if the hardware **writes the array elements atomically** — e.g., it races for byte values on some architectures.

Implementation of Mutexes

Properties of Mutexes

Yielding/spinning
A yielding mutex returns control to the operating system when it blocks. A spinning mutex consumes processor cycles while blocked.

Reentrant/nonreentrant
A reentrant mutex allows a thread that is already holding a lock to acquire it again. A non-reentrant mutex deadlocks if the thread attempts to reacquire a mutex it already holds.

Fair/unfair
A fair mutex puts blocked threads on a FIFO queue, and the unlock operation unblocks the thread that has been waiting the longest. An unfair mutex lets any blocked thread go next.

Simple Spinning Mutex

```
Spin_Mutex:
  cmp 0, mutex ; Check if mutex is free
  je Get_Mutex
  pause ; x86 hack to unconfuse pipeline
  jmp Spin_Mutex
Get_Mutex:
  mov 1, %eax
  xchg mutex, %eax ; Try to get mutex
  cmp 0, %eax ; Test if successful
  jne Spin_Mutex
Critical_Section:
  <critical-section code>
  mov 0, mutex ; Release mutex
```

Key property: `xchg` is an atomic exchange.

Simple Yielding Mutex

```
Spin_Mutex:
 cmp 0, mutex ; Check if mutex is free
 je Get_Mutex
 call thread_yield ; Yield quantum
 jmp Spin_Mutex
Get_Mutex:
 mov 1, %eax
 xchg mutex, %eax ; Try to get mutex
 cmp 0, %eax ; Test if successful
 jne Spin_Mutex
Critical_Section:
 <critical-section code>
 mov 0, mutex ; Release mutex
```

Yield

Competitive Mutex

Competing goals:

- To claim mutex soon after it is released.
- To behave nicely and waste few cycles.

IDEA: Spin for a while, and then yield.

How long to spin?

As long as a context switch takes. Then, you never wait longer than twice the optimal time.

- If the mutex is released while spinning, optimal.
- If the mutex is released after yield, $\leq 2 \times$ optimal.

Randomized algorithm [Karlin, Manasse, McGeoch, Owicki 94]

A clever randomized algorithm can achieve a competitive ratio of $e/(e-1) \approx 1.58$.

Locking Anomaly - Deadlock

Deadlock

Holding more than one lock at a time can be dangerous:

Thread 1:

1
```
lock(&A)
lock(&B)
  // critical section
unlock(&B)
unlock(&A)
```

Thread 2:

2
```
lock(&B)
lock(&A)
  // critical section
unlock(&A)
unlock(&B)
```

Neither thread can continue - the ultimate loss of performance!

Conditions for Deadlock

1. Mutual exclusion — Each thread claims exclusive control over the resources it holds.

2. Nonpreemption — Each thread does not release the resources it holds until it completes its use of them.

3. Circular waiting — A **cycle of threads** exists in which each thread is blocked waiting for resources held by the next thread in the cycle.

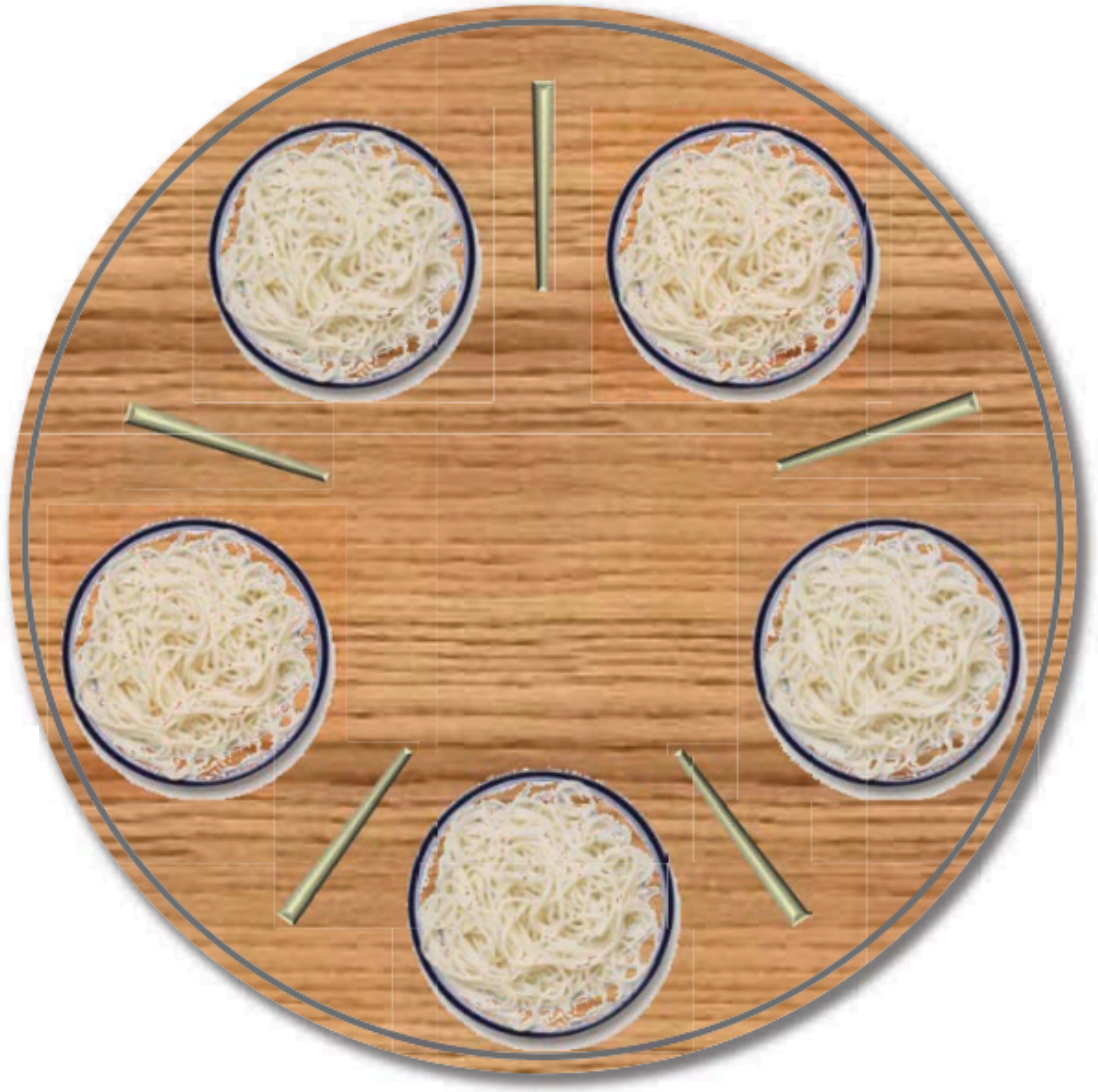
In the previous example

Dining Philosophers - A Story of Deadlock

Each of n philosophers needs the two chopsticks on either side of his/her plate to eat his/her noodles.

Philosopher `i`**:**

```
while (1) {
  think();
  lock(&chopstick[i].L);
  lock(&chopstick[(i+1)%n].L);
  eat();
  unlock(&chopstick[i].L);
  unlock(&chopstick[(i+1)%n].L);
}
```



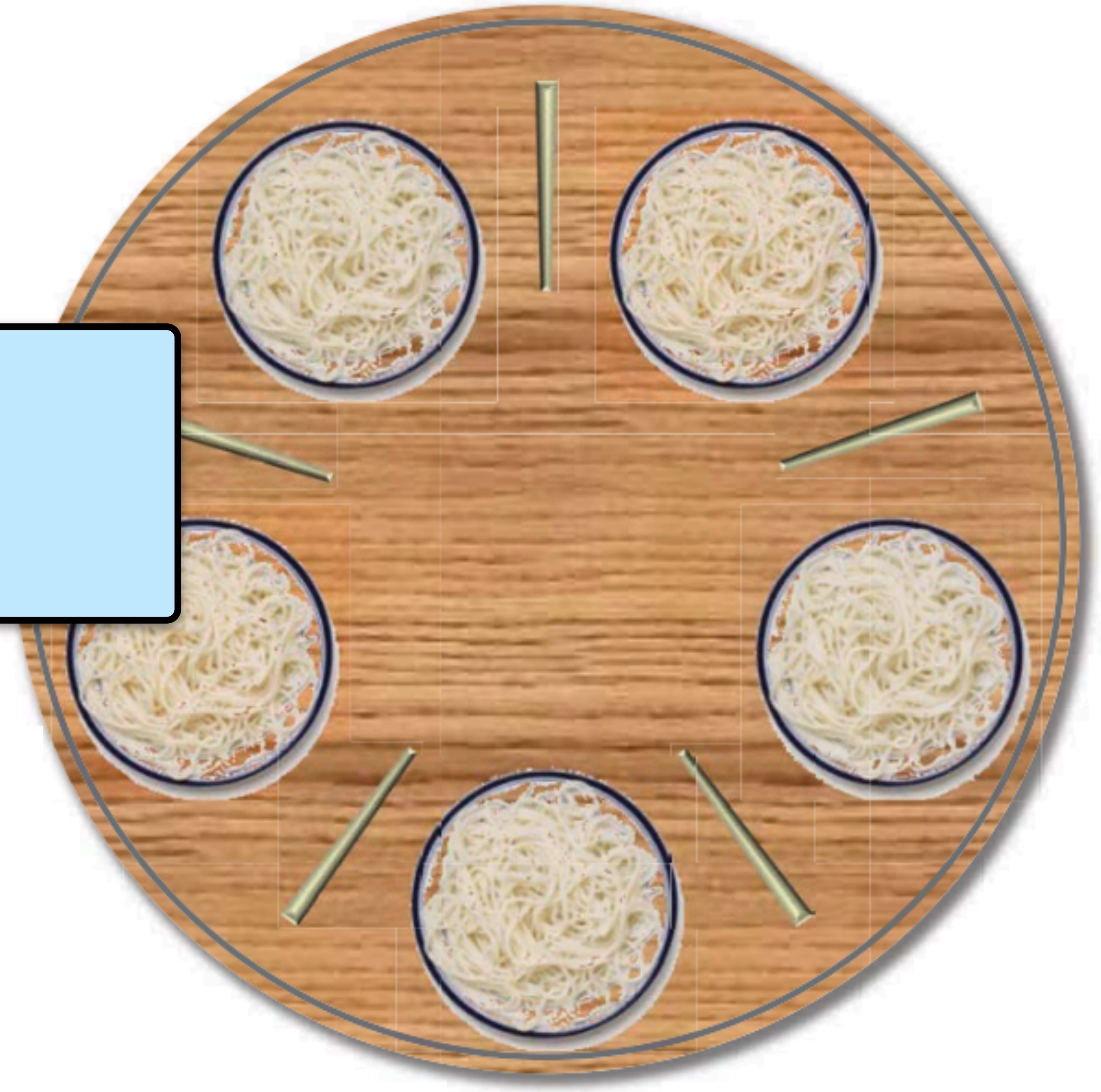
Dining Philosophers - A Story of Deadlock

Each of n philosophers needs the two chopsticks on either side of his/her plate to eat his/her noodles.

Philosopher i:

```
while (1) {
  think();
  lock(&chopstick[i].L);
  lock(&chopstick[(i+1)%n].L);
  eat();
  unlock(&chopstick[i].L);
  unlock(&chopstick[(i+1)%n].L);
}
```

One day they all pick up the left chopstick at the same time



Preventing Deadlock

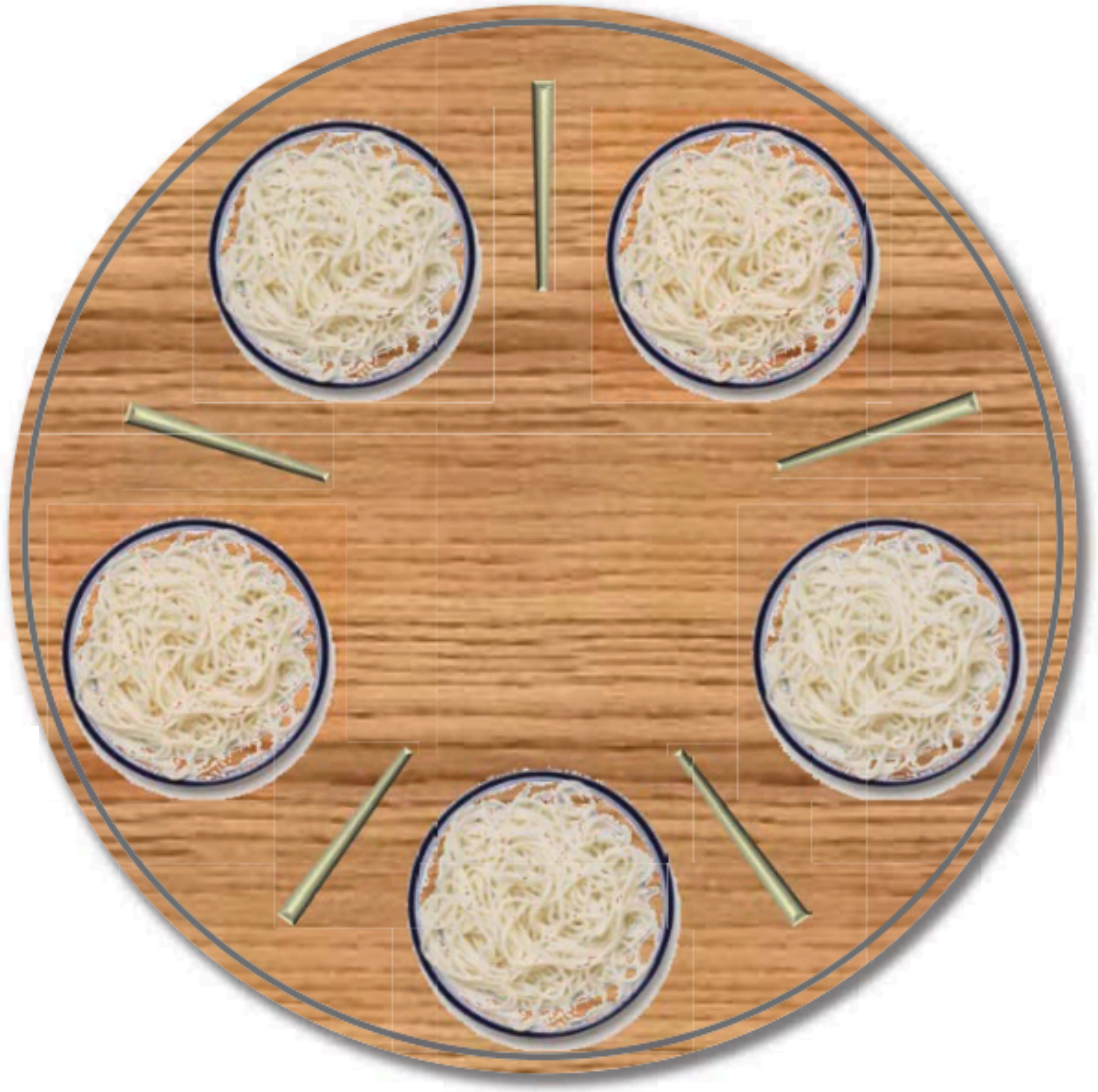
Theorem. Assume that we can linearly order the mutexes $L_1 \lessdot L_2 \lessdot \cdots \lessdot L_n$ so that whenever a thread holds a mutex L_i and attempts to lock another mutex L_j, we have $L_i \lessdot L_j$. Then, no deadlock can occur.

Proof. Suppose that a cycle of waiting exists. Consider the thread in the cycle that holds the “largest” mutex $L_{\max}$ in the ordering, and suppose that it is waiting on a mutex L held by the next thread in the cycle. Then, we must have $L_{\max} \lessdot L$. Contradiction.

Dining Philosophers

Philosopher i:

```
while (1) {
  think();
  lock(&chopstick[min(i,(i+1)%n)].L);
  lock(&chopstick[max(i,(i+1)%n)].L);
  eat();
  unlock(&chopstick[min(i,(i+1)%n)].L);
  unlock(&chopstick[max(i,(i+1)%n)].L);
}
```



Avoid deadlock with a total ordering of locks!

Locking Anomaly - Contention

Summing Example

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  // …

  int result = 0;

  for (size_t i = 0; i < n; ++i) {
    result += compute(myArray[i]);
  }
  printf("The result is: %d\n”, result);
  return 0;
}
```

Something O(1)

Summing Example

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  // …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    result += compute(myArray[i]);
  }
  printf("The result is: %d\n", result);
  return 0;
}
```

Something O(1)

Race

Mutex Solution

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  // …
  omp_lock_t lock;
  omp_init_lock(&lock);
  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n
    omp_set_lock(&lock);
    result += compute(myArray[i]);
    omp_unset_lock(&lock);
  }
  omp_destroy_lock(&lock);
  printf("The result is: %d\n”, result);
  return 0;
}
```

Init lock

Every thread has to get this lock

Lock contention => no parallelism

Performance Anomaly - False Sharing

Example: Serial Sum

```c
#define N 10000
int main()
{
  int i;
  float a[N];
  double result = 0.0;

  /* Some initializations */
  for (i = 0; i < N; i++) { a[i] = i * 1.0; }

  // do sum
  for (i = 0; i < N; i++)
    result += a[i];
  }

  printf("Final result= %f\n", result); return 0;
}
```

Example: Parallel Sum

```
#define NUM_THREADS 4
#define N 10000
int main()
{
  int i;
  float a[N];
  double partials[NUM_THREADS];
  double result = 0.0;

  omp_set_num_threads(NUM_THREADS);
  /* Some initializations */
  for (i = 0; i < NUM_THREADS; i++) { partials[i]  = 0.0; }
  for (i = 0; i < N; i++) { a[i] = i * 1.0; }

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    for (i = id; i < N; i += NUM_THREADS)
      partials[id] += a[i];
  }

  for(i = 0; i < NUM_THREADS; i++) { result += partials[i]; }
  printf("Final result= %f\n", result); return 0;
}
```

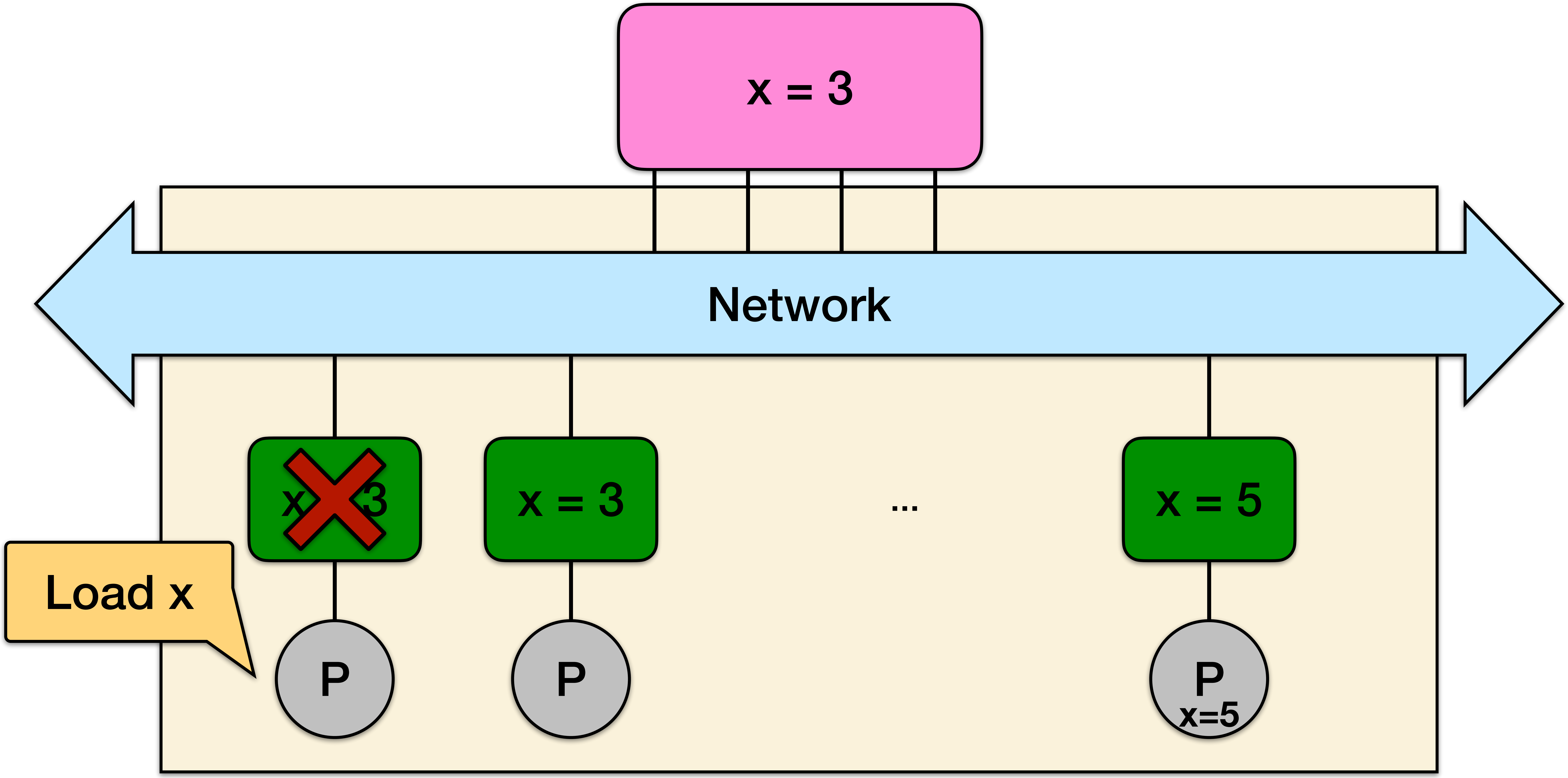
Array for partial sums - one entry per thread

Parallel region - every thread executes it

Add every id-th element into partials

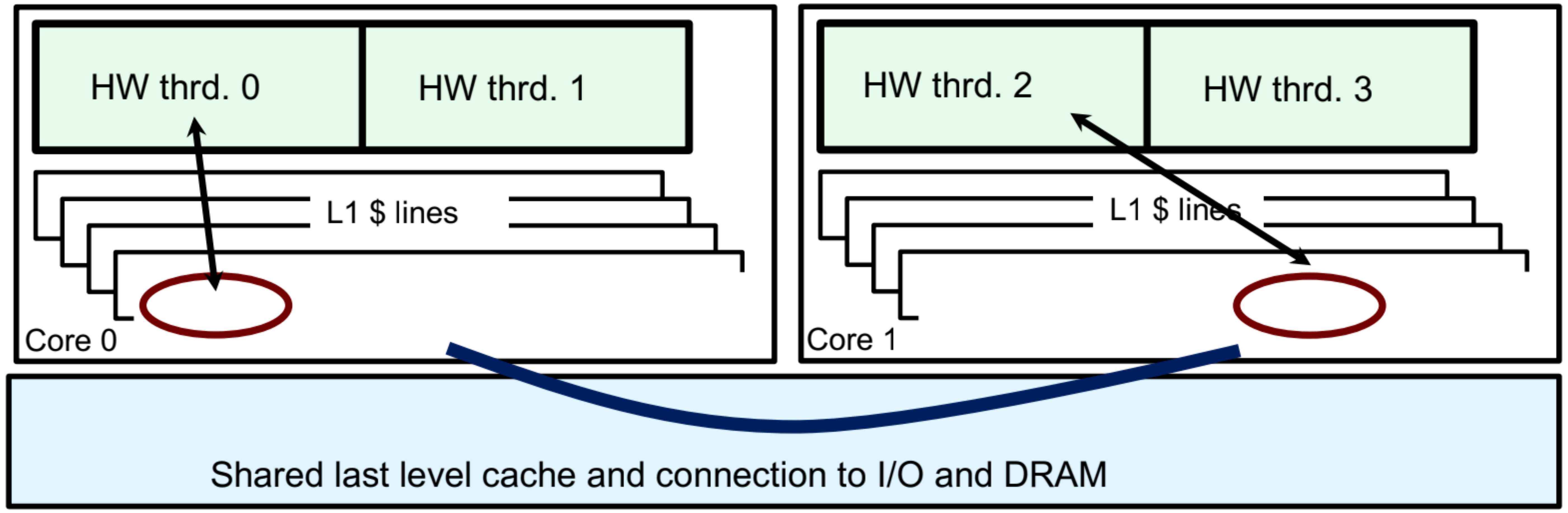
Combine

Recall: Cache coherence



False Sharing Limits Scalability

False sharing is a performance bug in which **independent data elements happen to sit on the same cache line**, so each update will cause the cache lines to "slosh back and forth" between threads.



If you promote scalars to an array to support creation of a parallel program, the **array elements are contiguous in memory** and hence share cache lines -> results in poor scalability

Solution: **Pad arrays** so elements you use are on distinct cache lines.

Example: Eliminate false sharing by padding

```
#define NUM_THREADS 4
#define N 10000
#define PAD 8
int main()
{
  int i;
  float a[N];
  double partials[NUM_THREADS][PAD];
  double result = 0.0;

  omp_set_num_threads(NUM_THREADS);
  /* Some initializations */
  for (i = 0; i < NUM_THREADS; i++) { partials[i][0]  = 0.0; }
  for (i = 0; i < N; i++) { a[i] = i * 1.0; }

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    for (i = id; i < N; i += NUM_THREADS)
      partials[id][0] += a[i];
  }

  for(i = 0; i < NUM_THREADS; i++) { result += partials[i][0]; }
  printf("Final result= %f\n", result); return 0;
```

Assuming 64-byte cache line size

Pad so only one real value per cache line

First element in cache line is the partial

Combine

Fine-Grained Synchronization

Fine-Grained Synchronization

- Instead of using a single lock for the entire data structure, split the concurrent object into **independently-synchronized components**.
- Threads conflict when they access the **same component at the same time**.
- As opposed to coarse-grained synchronization, which locks the entire object.

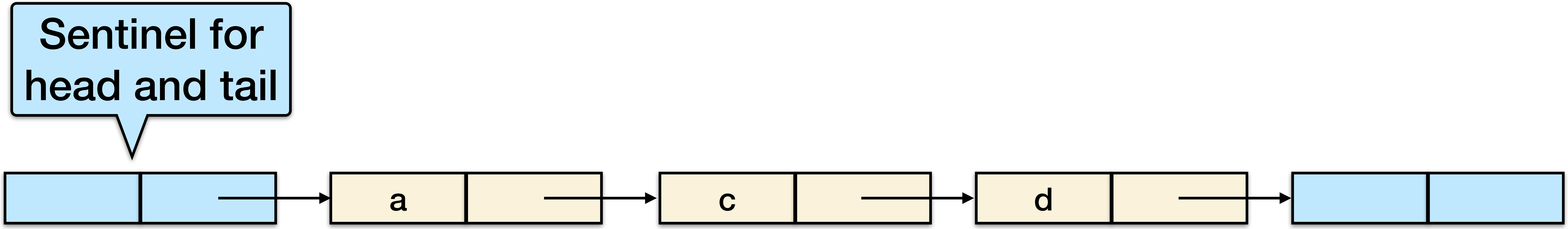
Example:

Lock per node in list

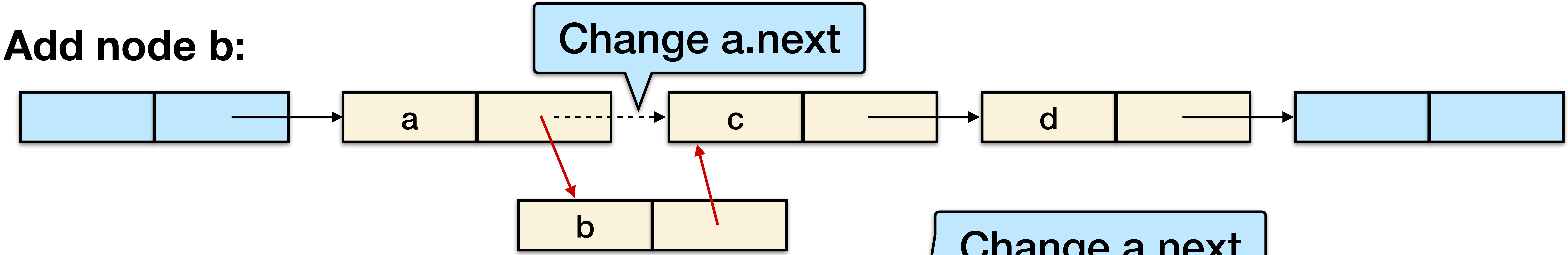
```
typedef struct node {
 lock_t l;
 int data;
 struct node *next; } node_t;
```

https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

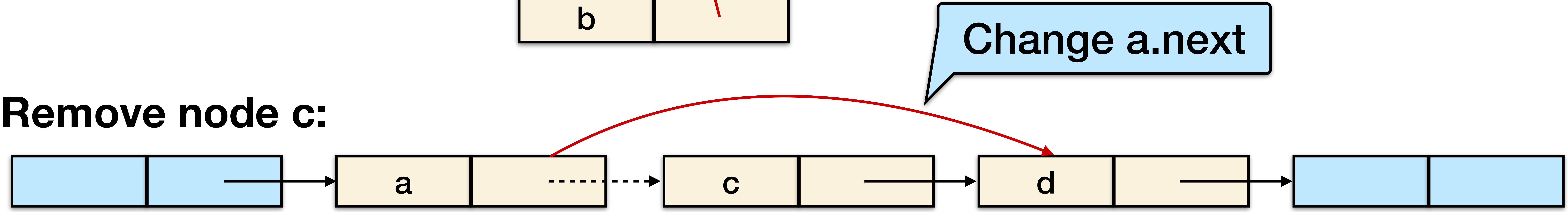
Example: List-Based Set



Add node b:



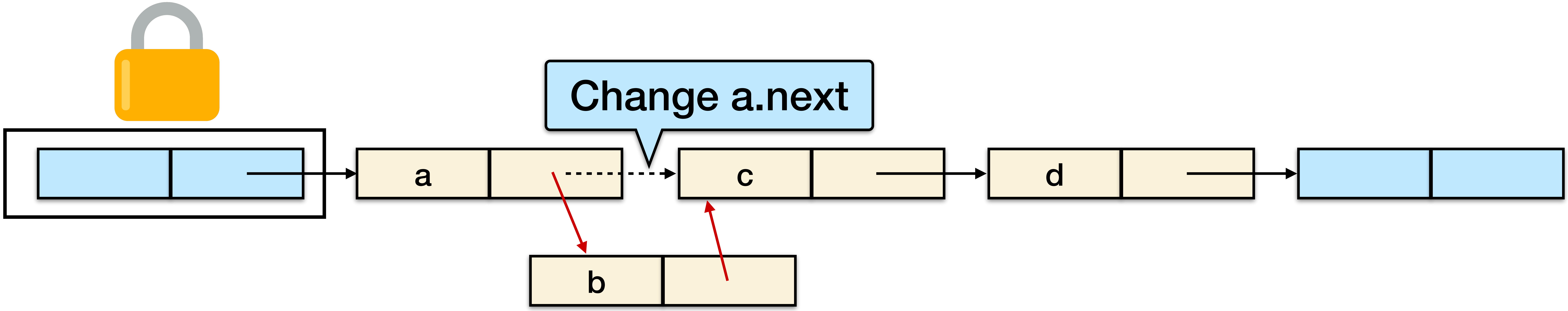
Remove node c:



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Example: Coarse-Grained Locking

A simple solution is to lock the entire list for each operation e.g., by locking the head



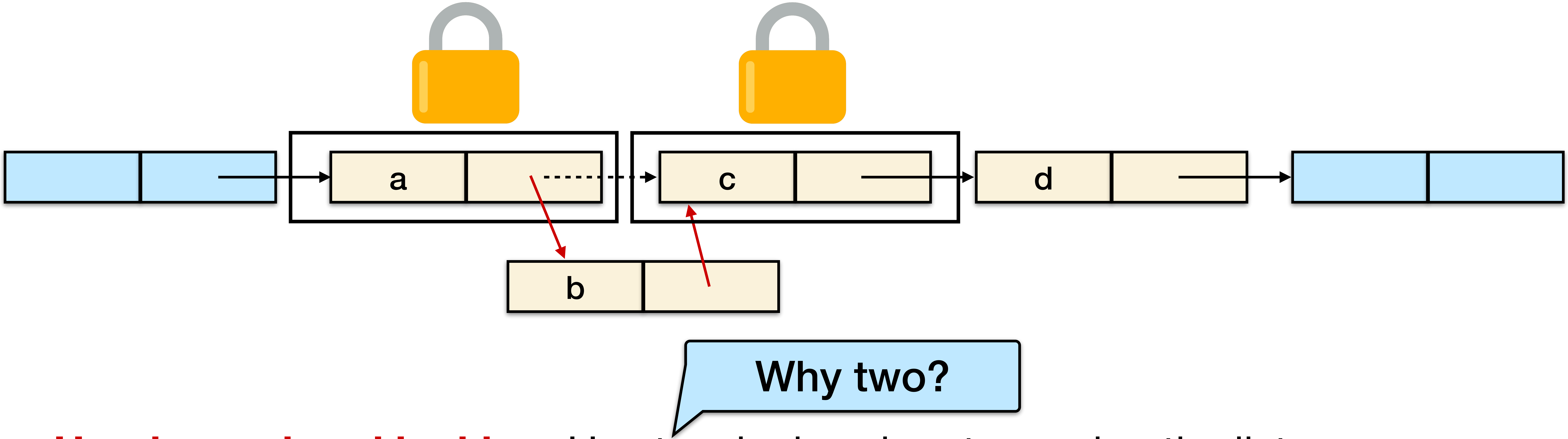
Simple and clearly correct, but works poorly with contention…

https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Example: Fine-Grained Locking

Split object (list) into pieces (nodes)

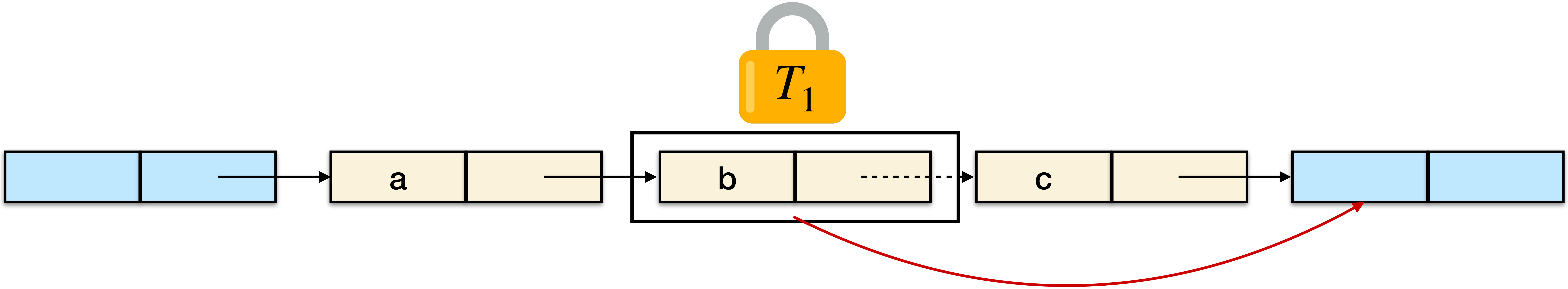
- Each piece (each node in the list) has its own lock
- Methods that work on disjoint pieces need not exclude each other



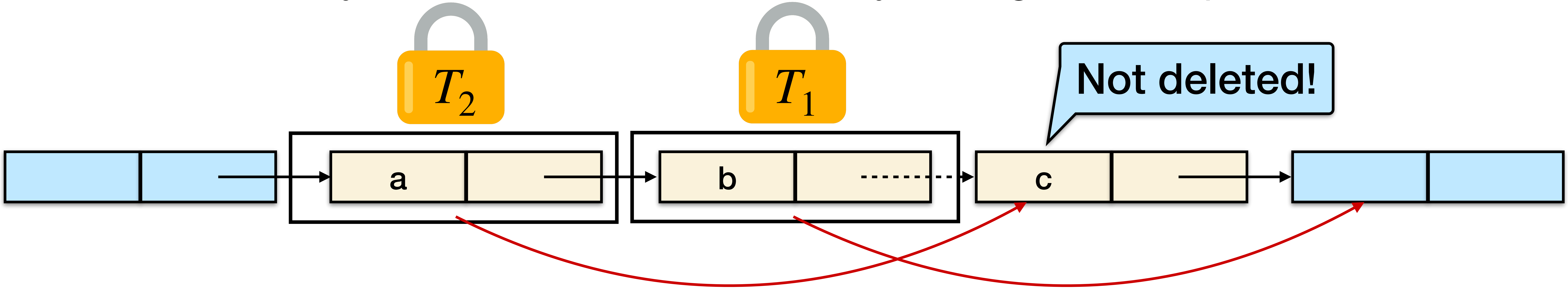
Hand-over-hand locking: Use two locks when traversing the list

Problem with one lock

Assume that we want to delete node c.
We lock node b and set its next pointer to the node after c.



Another thread may concurrent delete node b by setting the next pointer from a to c.



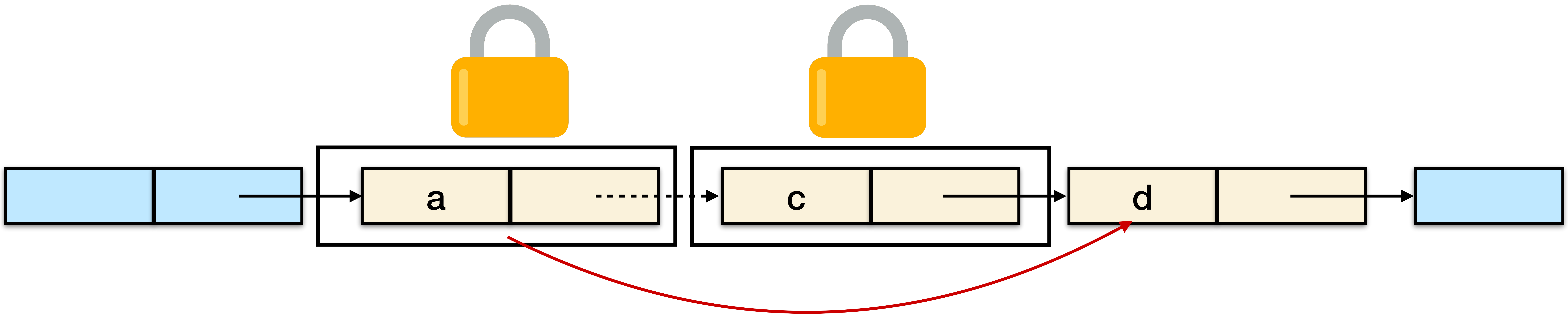
https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking Insight

If a node is locked, no one can delete the node's **successor**.

If a thread locks **the node to be deleted and also its predecessor**, then it works!

That's why we (have to) use **two locks**.

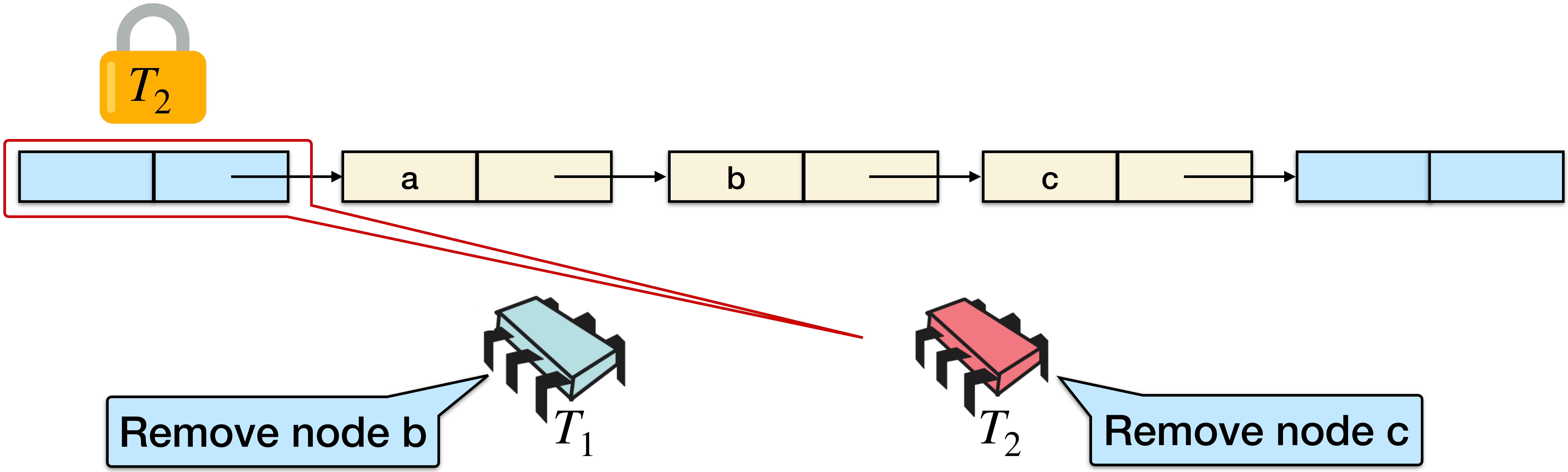


https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

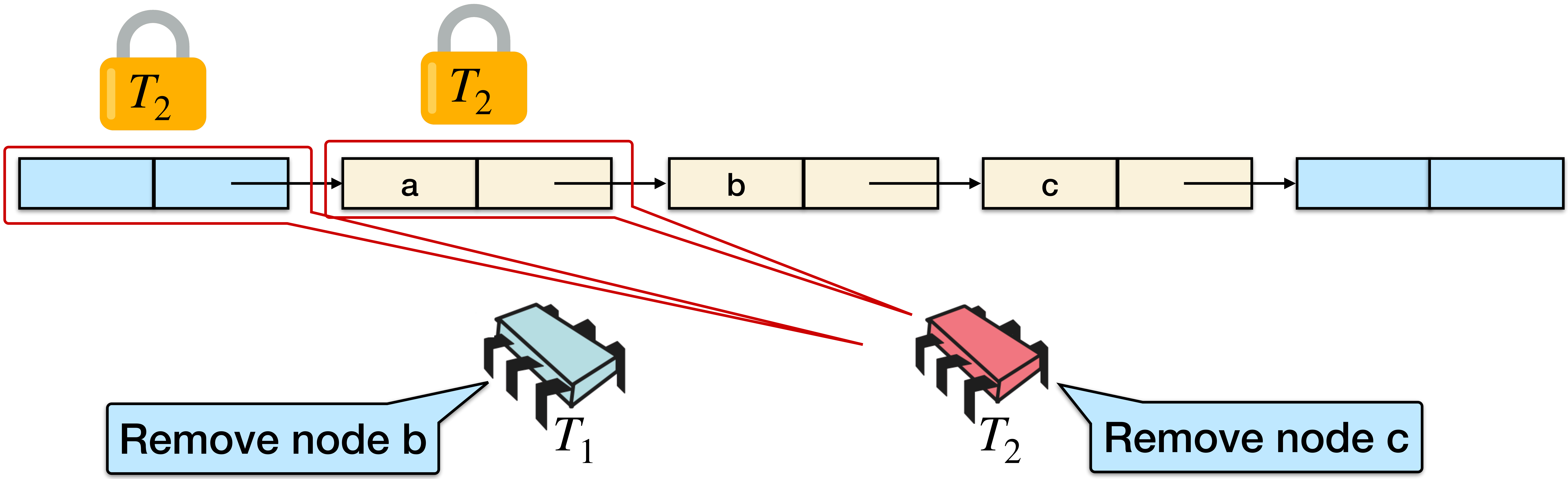
Suppose that two threads want to remove the nodes b and c.

One thread acquires the lock on the sentinel, the other has to wait



Hand-Over-Hand Locking: Removing Nodes

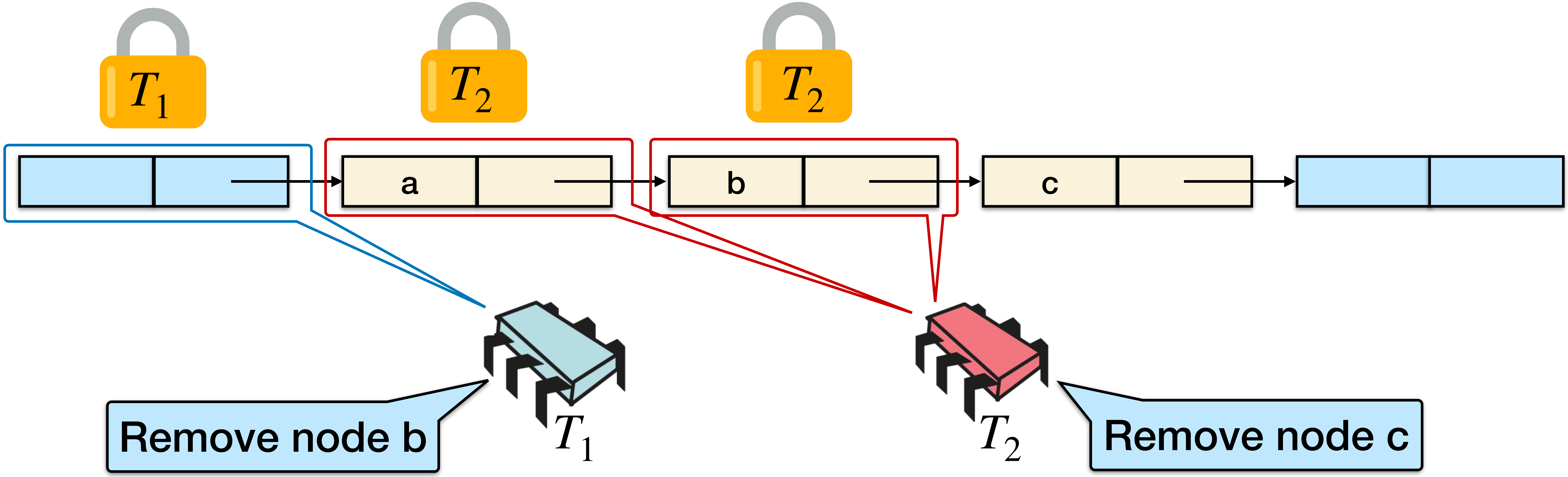
The same thread that acquired the sentinel lock can then lock the next node.



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

Before locking node b, the sentinel lock is released, so the other thread can now acquire the sentinel lock.

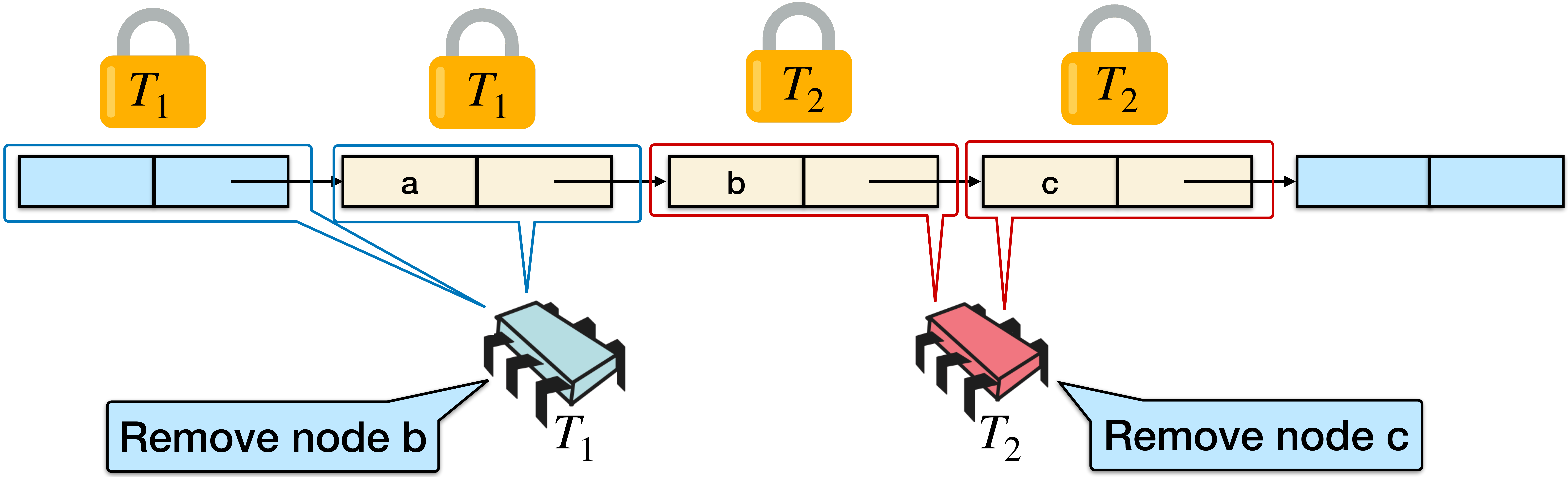


https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

Before locking node c, the lock of node a is released.

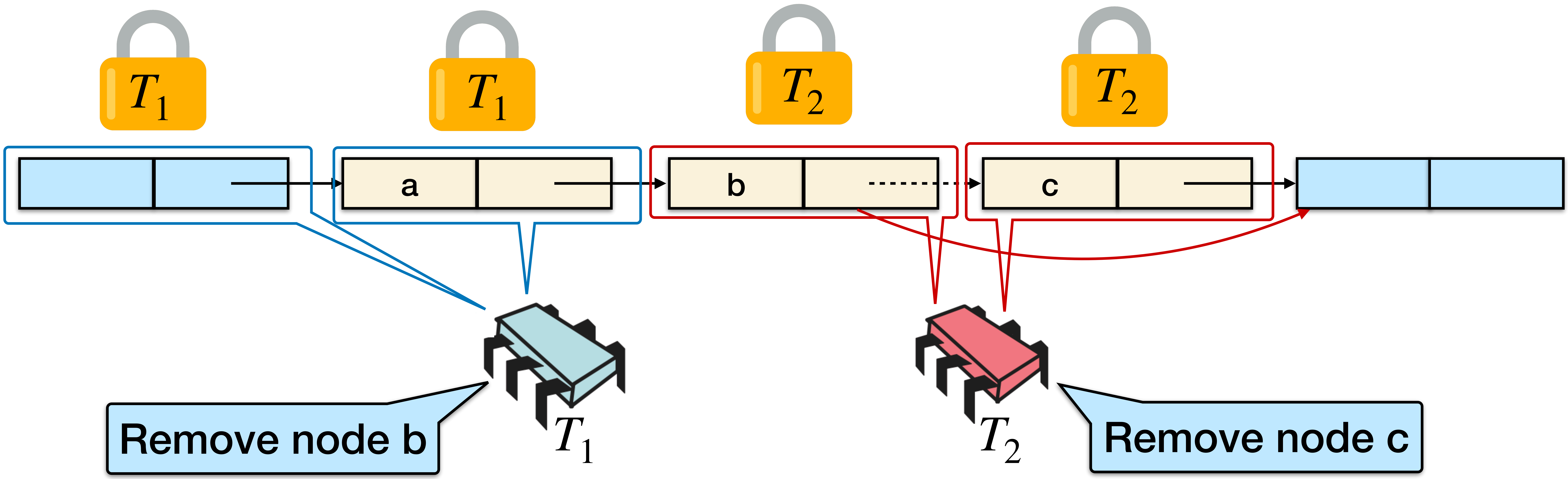
The other thread can now lock node a.



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

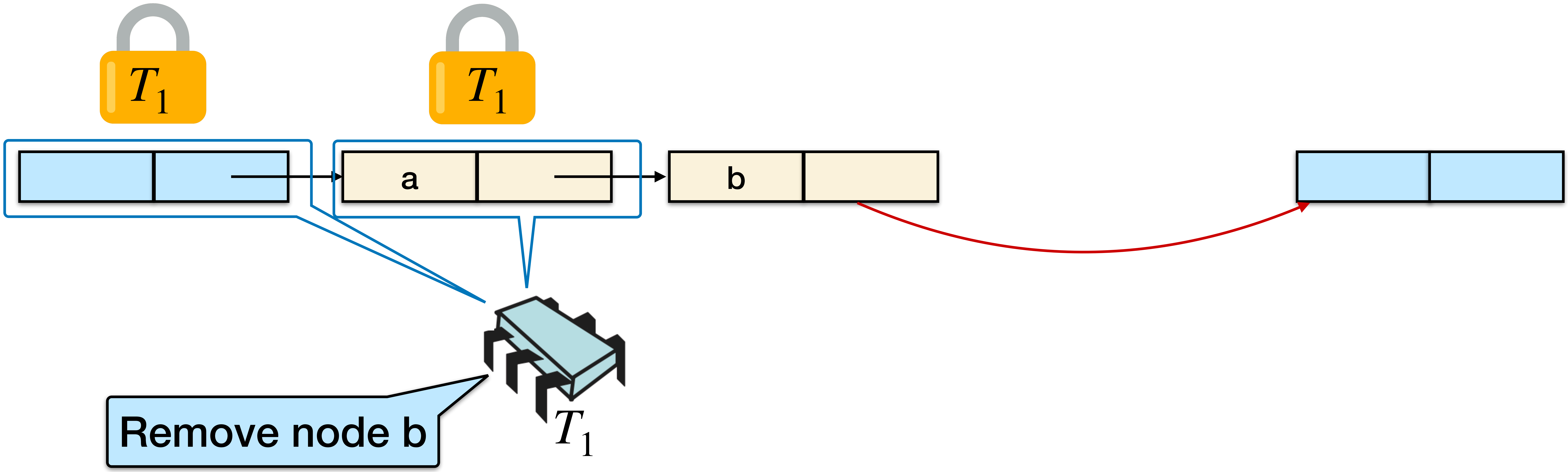
Node c can now be removed, and the two locks can be released.



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

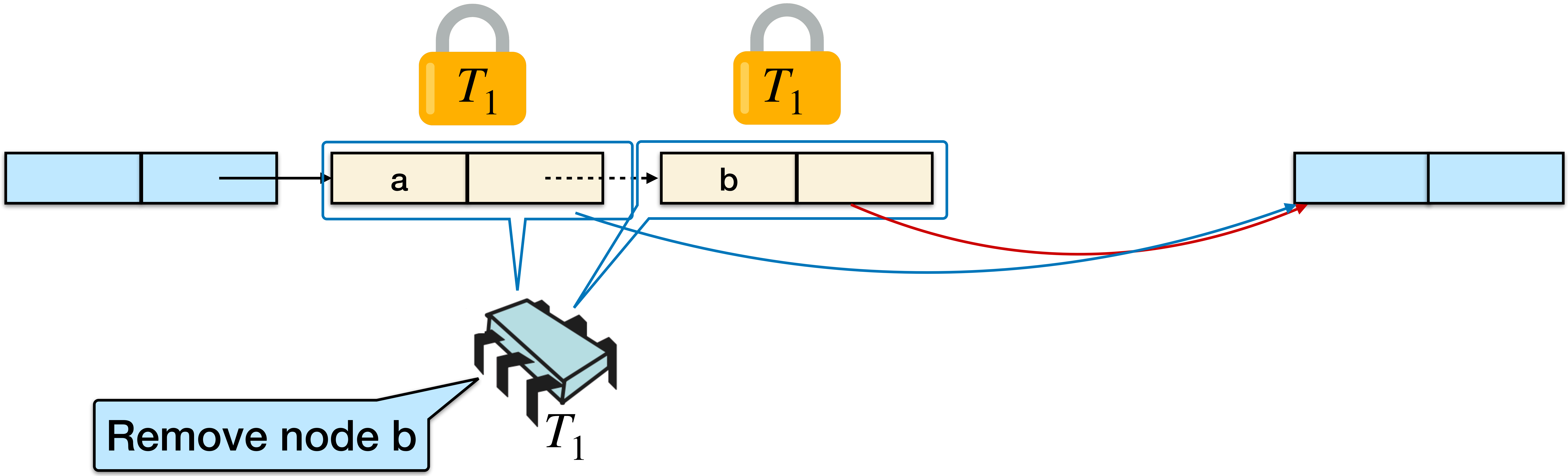
Node c can now be removed, and the two locks can be released.



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Hand-Over-Hand Locking: Removing Nodes

The other thread can now lock node b and remove it.



Why does this work?

To remove a node
- The node must be locked
- Its predecessor must be locked

Therefore, if you lock a node – It can't be removed – And neither can its successor

Drawbacks

Hand-over-hand locking is sometimes better than coarse-grained locking

- Threads can traverse in parallel
- Sometimes, it's worse!

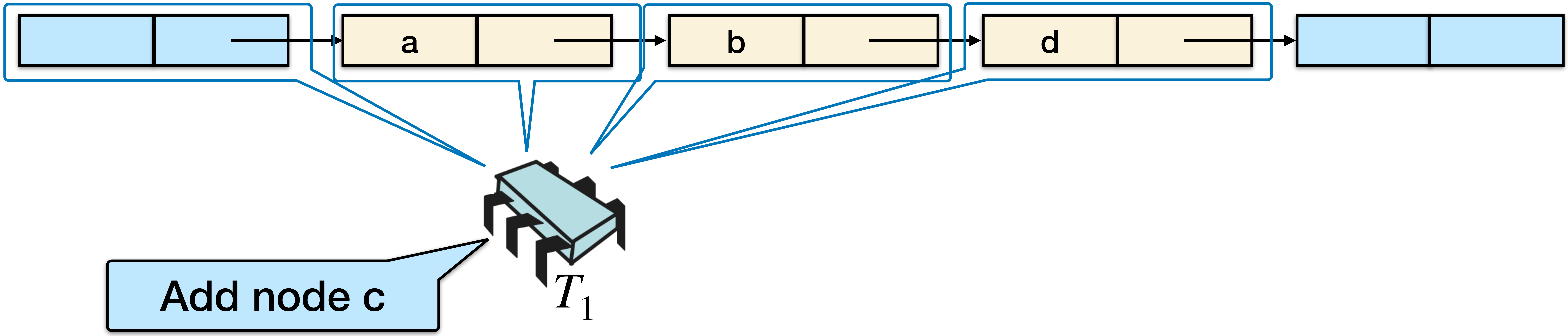
However, it's certainly not ideal - inefficient because **many locks must be acquired and released**

How can we do better?

https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Optimistic Synchronization

Traverse the list without locking!

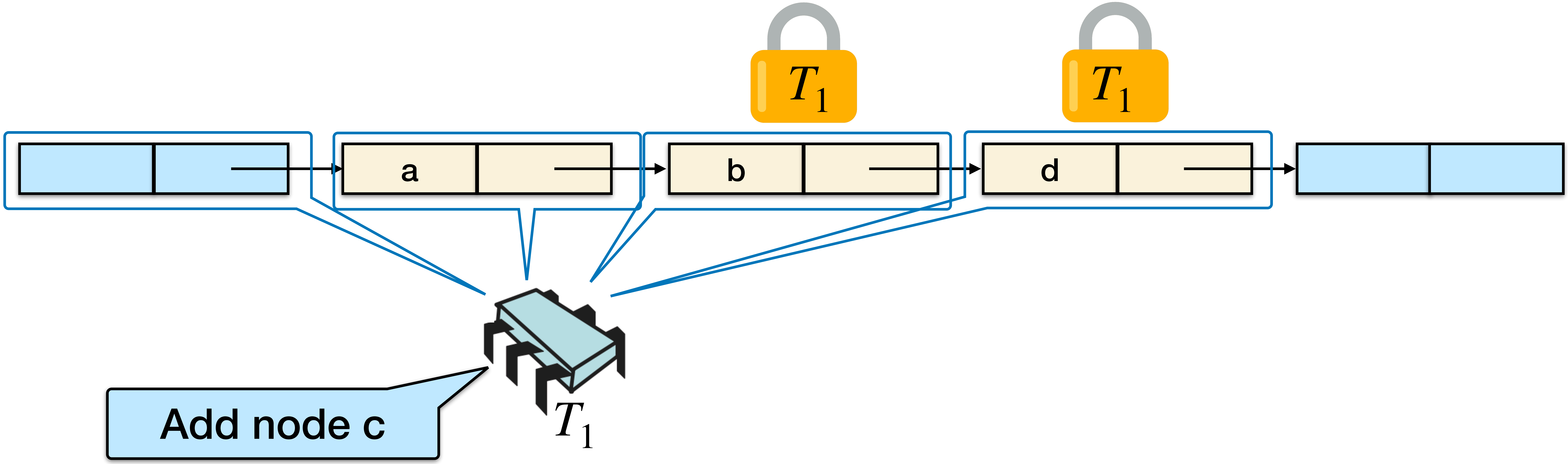


Optimistic Synchronization

Once the nodes are found, try to lock them.

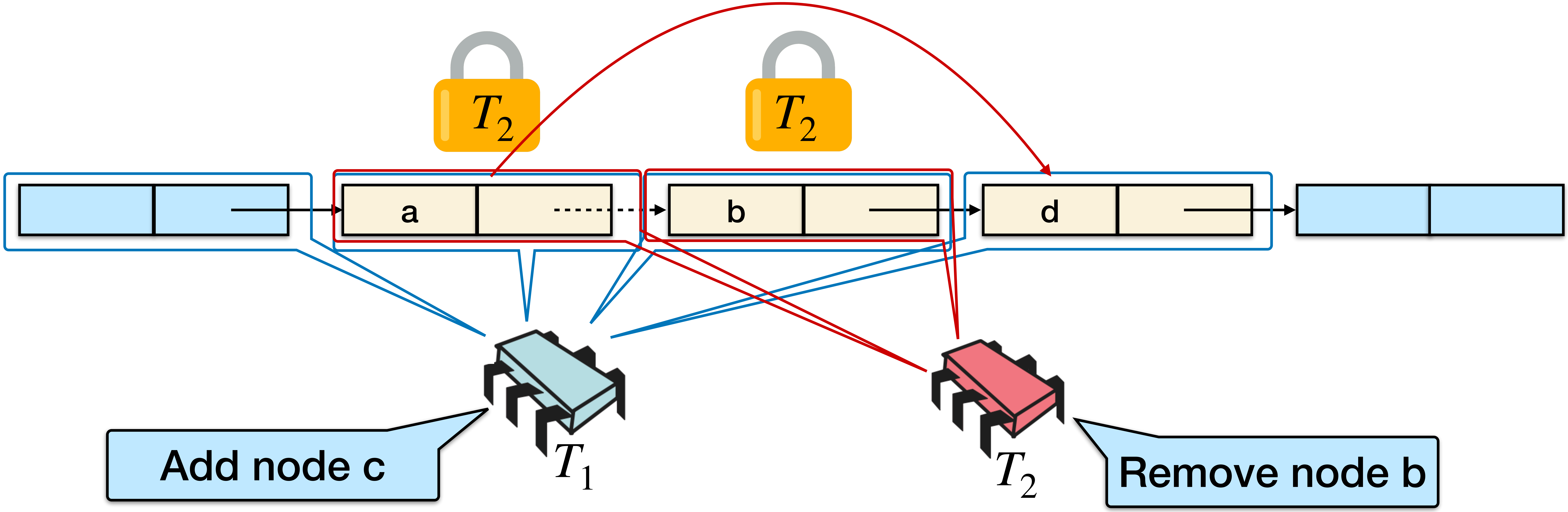
Check that everything is ok!

What can go wrong?



Issue: Concurrent removal of the predecessor of node you want to add

Another thread may lock nodes a and b and remove b before node c is added.

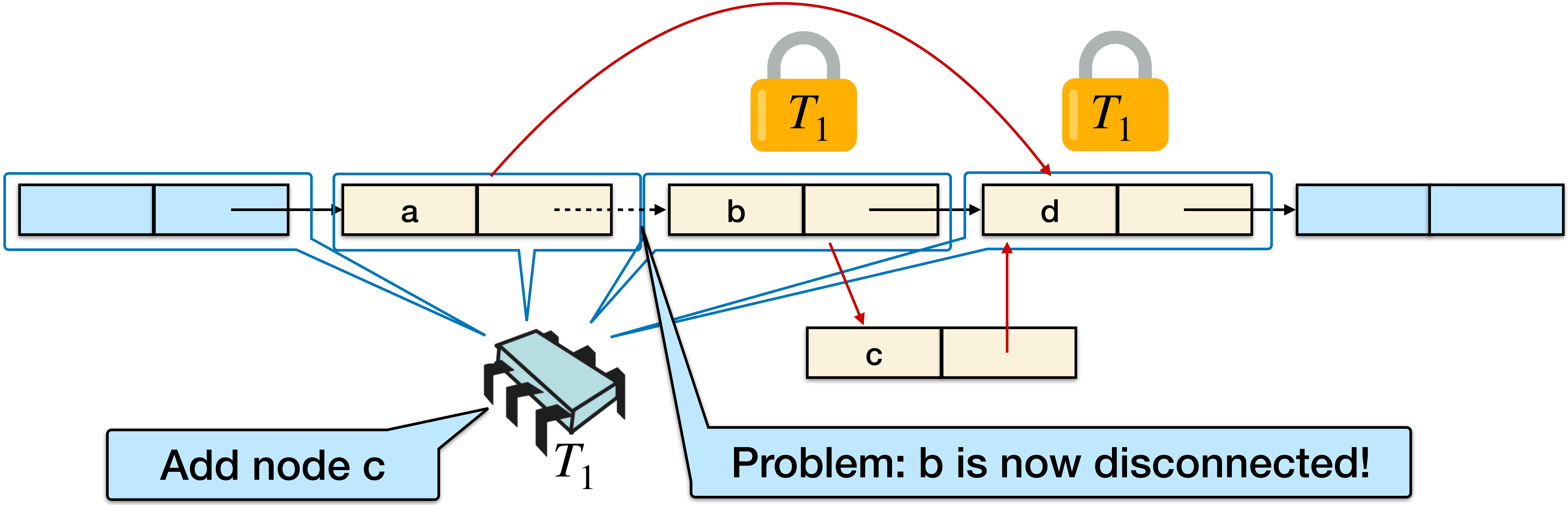


https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Issue: Concurrent removal of the predecessor of node you want to add

Another thread may lock nodes a and b and remove b before node c is added.

If the pointer from node b is set to node c, then node c is not added to the list!

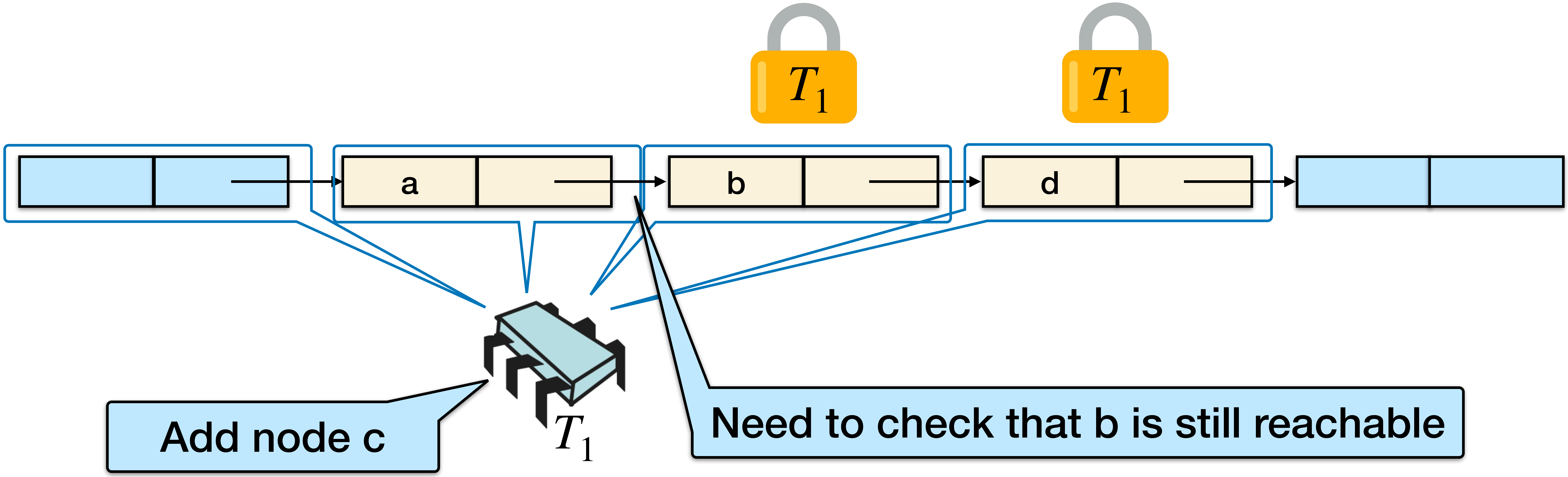


https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

Solution: Validation

After locking node b and node d, traverse the list again to **verify that b is still reachable**.

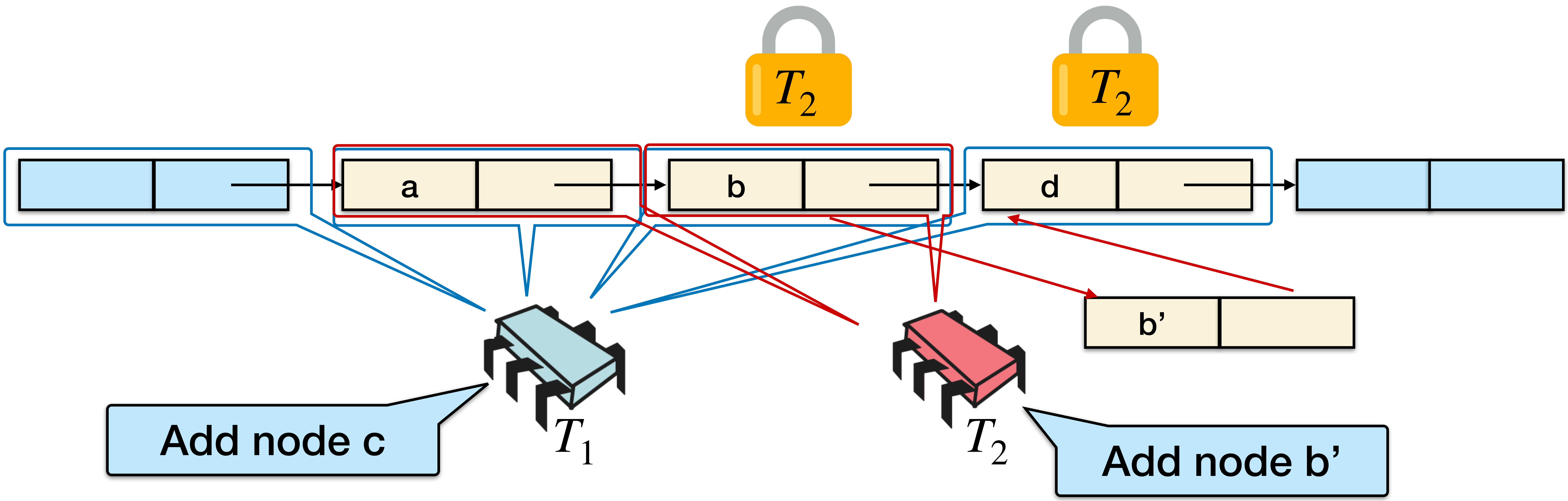
If it is not, **start over**.



https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

What else can go wrong?

Another thread may lock nodes b and d and add a node b’ before c is added.

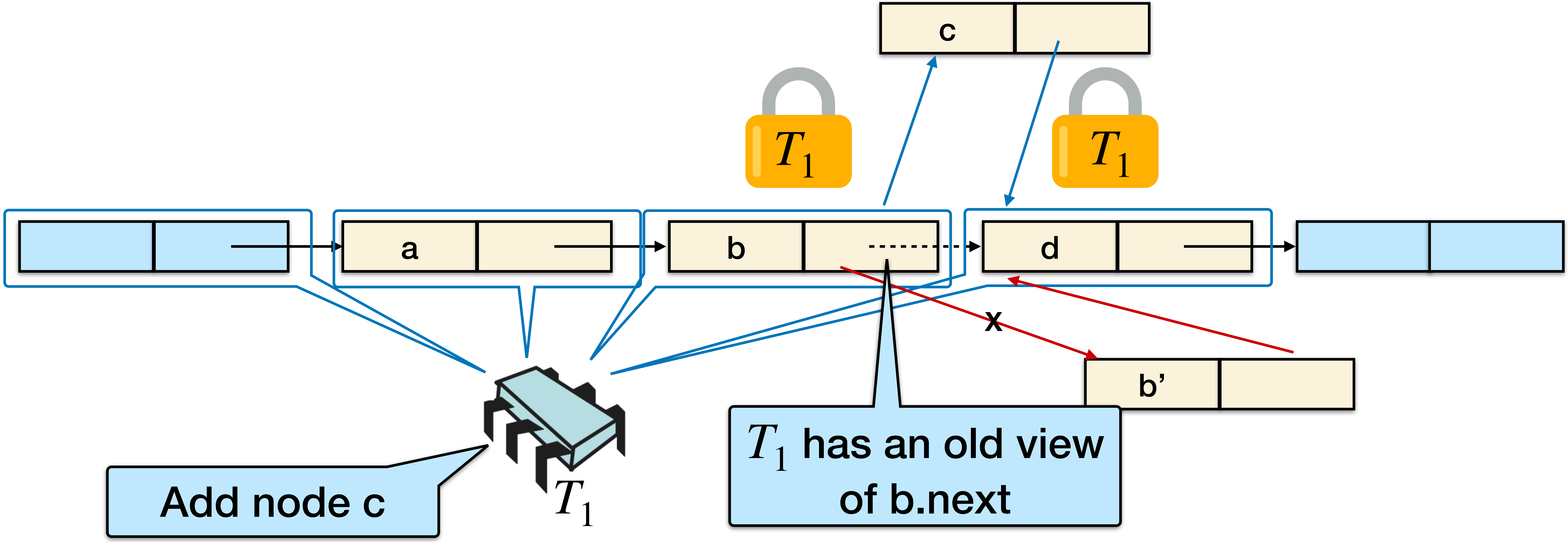


https://disco.ethz.ch/courses/fs16/ti2/lecture/chapter08_2on1.pdf

What else can go wrong?

Another thread may lock nodes b and d and add a node b’ before c is added.

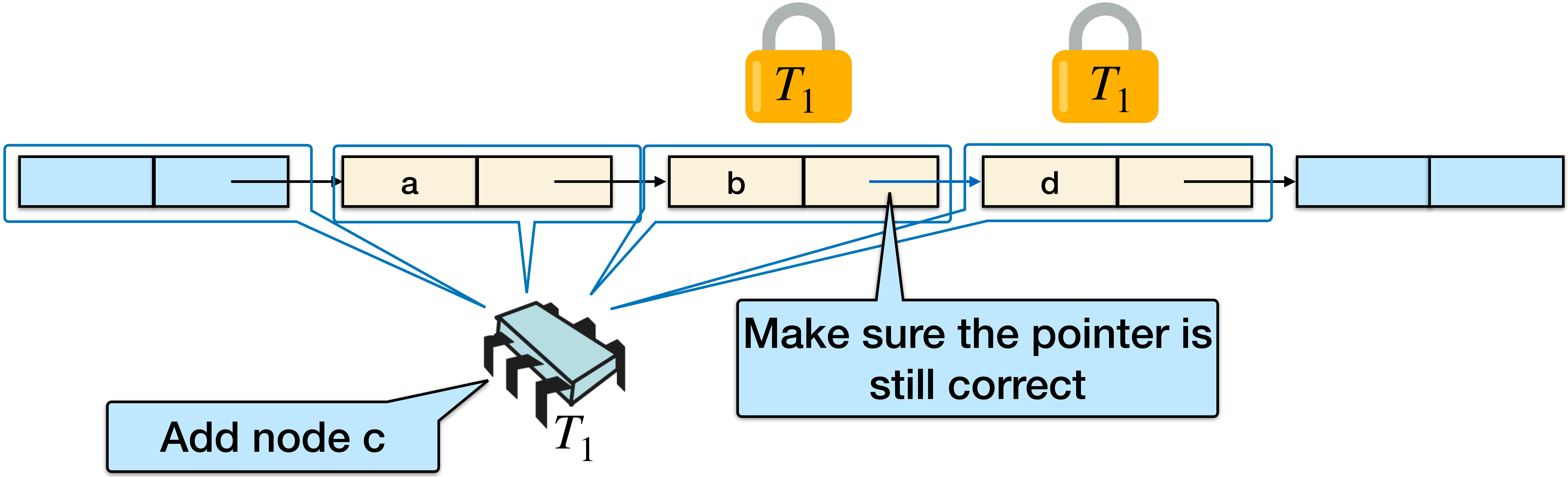
By adding node c, the addition of b’ is undone!



Solution: More validation

After locking nodes b and d, also check that node b still points to node d!

If not, **start over**.



Optimistic Synchronization Summary

Why is this correct?

- If nodes b and c are both locked, node b still accessible, and node c still the successor of node b, then neither b nor c will be deleted by another thread
- This means that it's ok to delete node c!

Why is it good to use optimistic synchronization?

- **Limited hot-spots**: no contention on traversals
- Fewer lock acquisitions and releases

When is it good to use optimistic synchronization?

- When the cost of scanning twice without locks is less than the cost of scanning once with locks

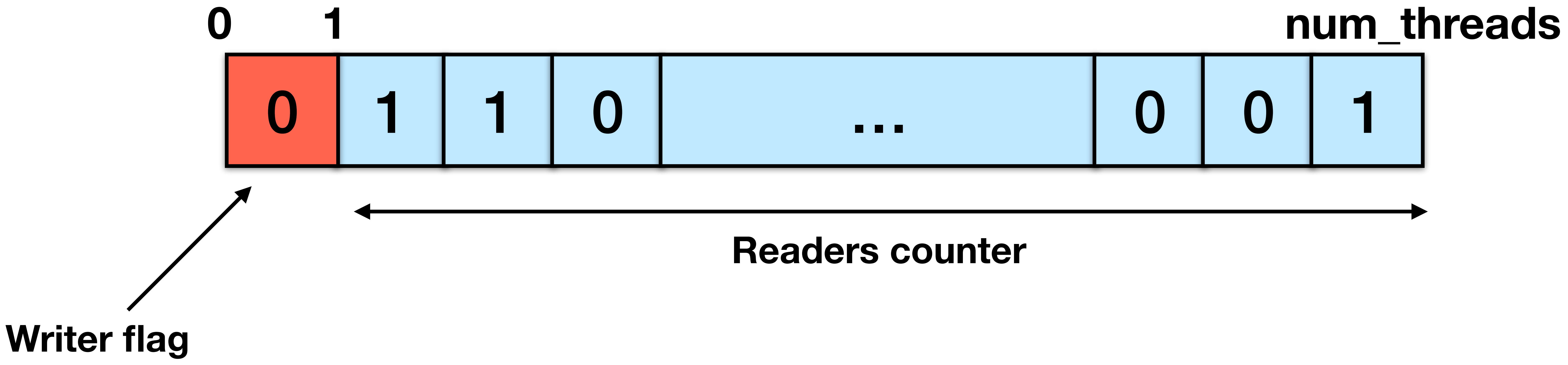
B+-tree Concurrency Control

Reader-Writer Locks

- A reader-writer lock allows **concurrent access for read-only operations**, whereas write operations require exclusive access.

- That is, multiple threads can read the data in parallel, but **an exclusive lock is needed for writing/modifying data**.

- All other threads (both writers and readers) are blocked when the lock is taken in write mode.



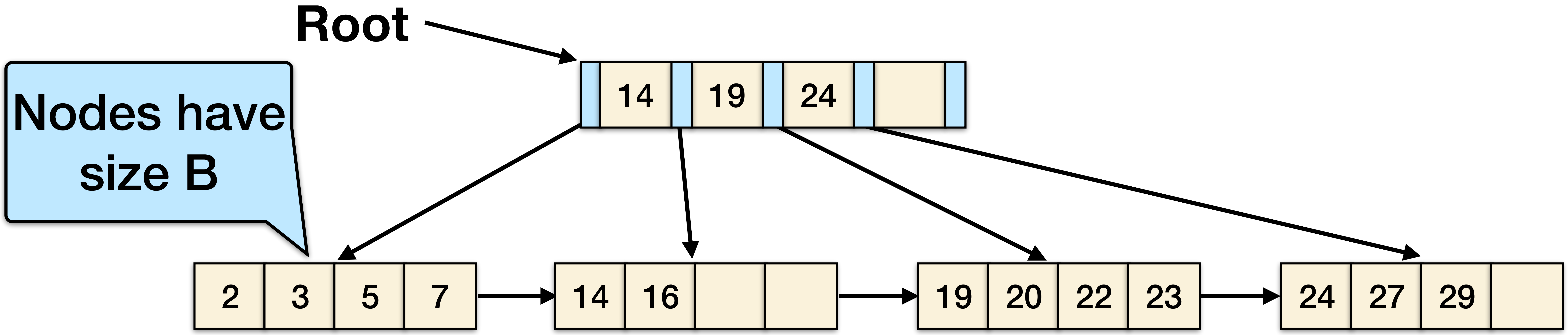
High-Level Strategy

- Goal: allow multiple threads to read and update a B+-tree at the same time.

- We need to protect from two kinds of problems:

  - Threads trying to modify the **contents of a node** at the same time.

  - One thread traversing the tree while another thread **merges/splits** nodes.



Latch Crabbing / Coupling

In database indexing, the term "lock" is for transactions, while "latches" are for operations.

Protocol to enable multiple threads to access/modify a B+-tree at the same time.

Basic idea:

- Get latch for parent
- Get latch for child.
- Release latch for parent if it is deemed **safe**.

A safe node is one that **will not split or merge** when updated.

- Not full (upon insertion)
- More than half-full (upon deletion)

Latch Crabbing / Coupling

Find: Start at root and traverse down to the correct leaf.

- Acquire **R**(eader) latch on child.
- Then unlatch parent.

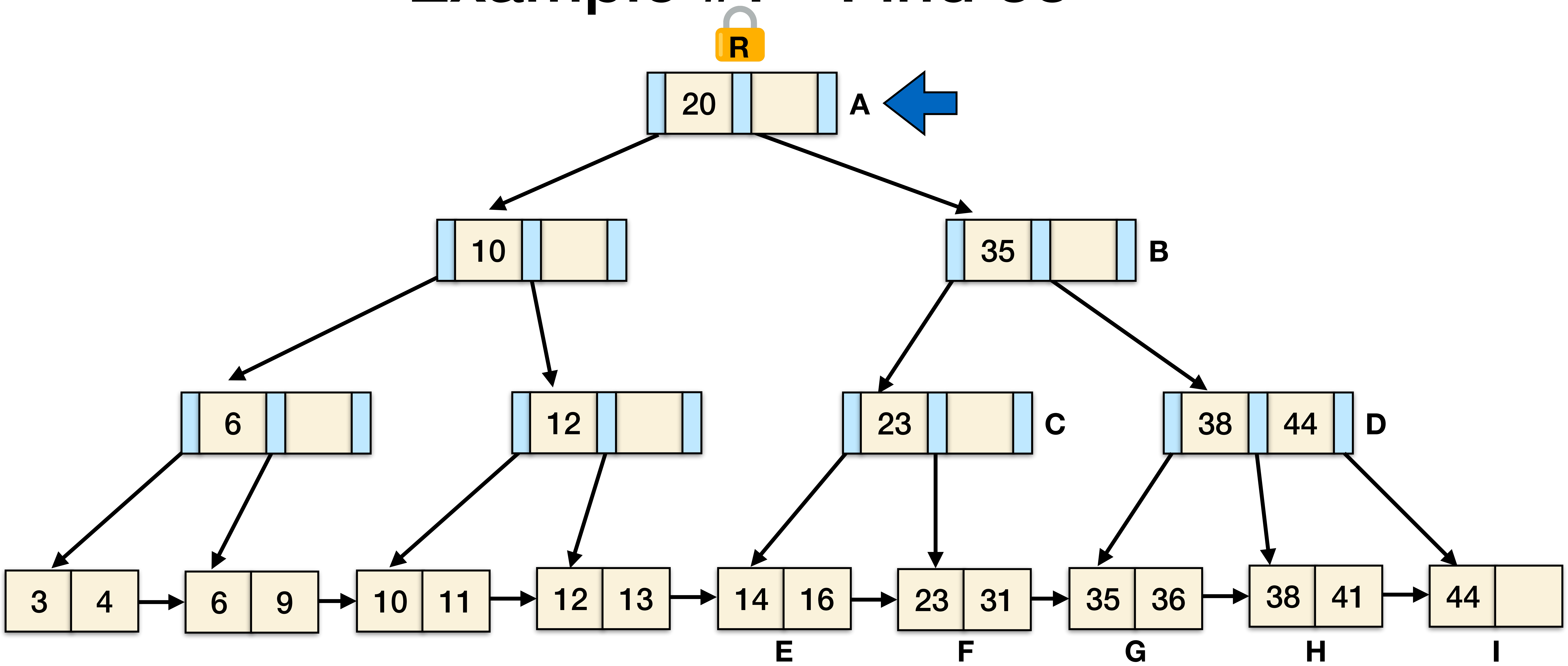
Similar to hand-over-hand

Insert/Delete: Start at root, and go down, obtaining **W**(riter) latches as needed.

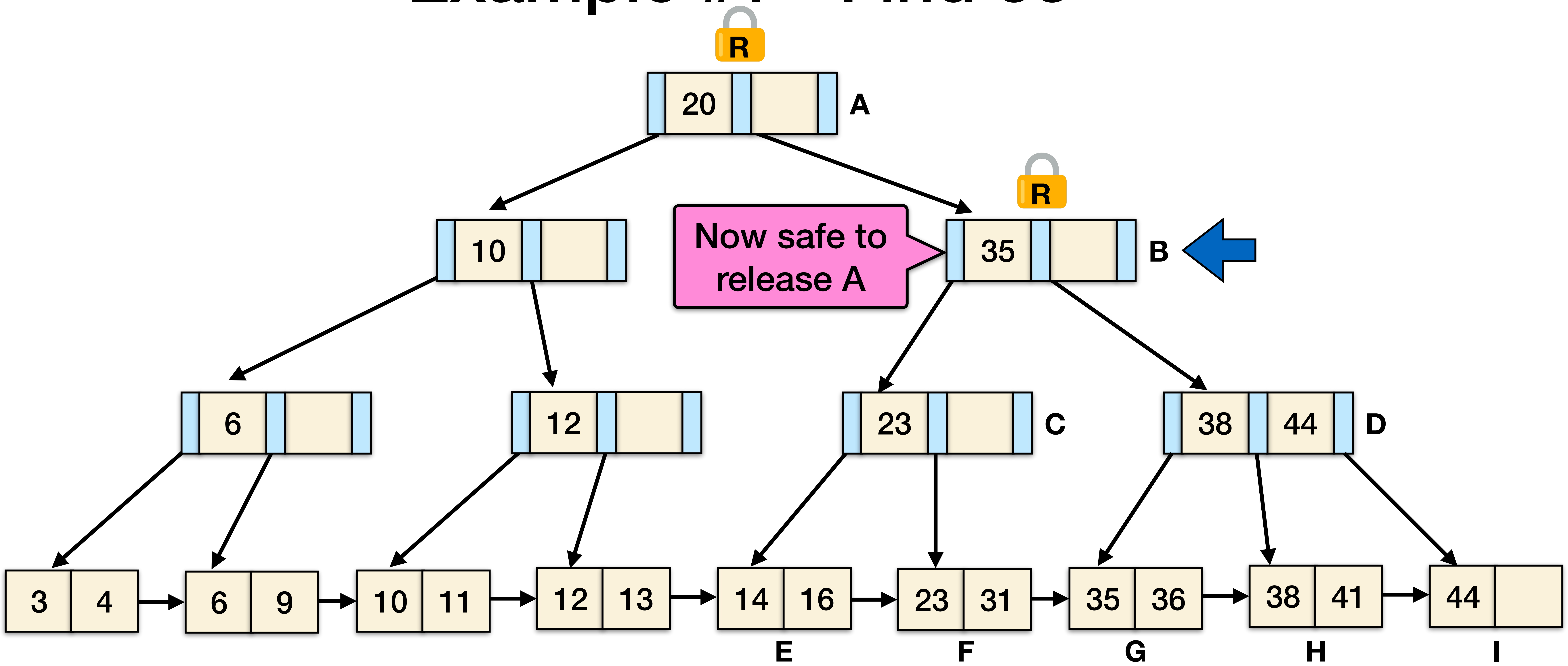
- Once the child is latched, check that it is safe.
- If it is safe, release all latches on ancestors.

Modified hand-over-hand: can keep hold on ancestors, if necessary

Example #1 - Find 38

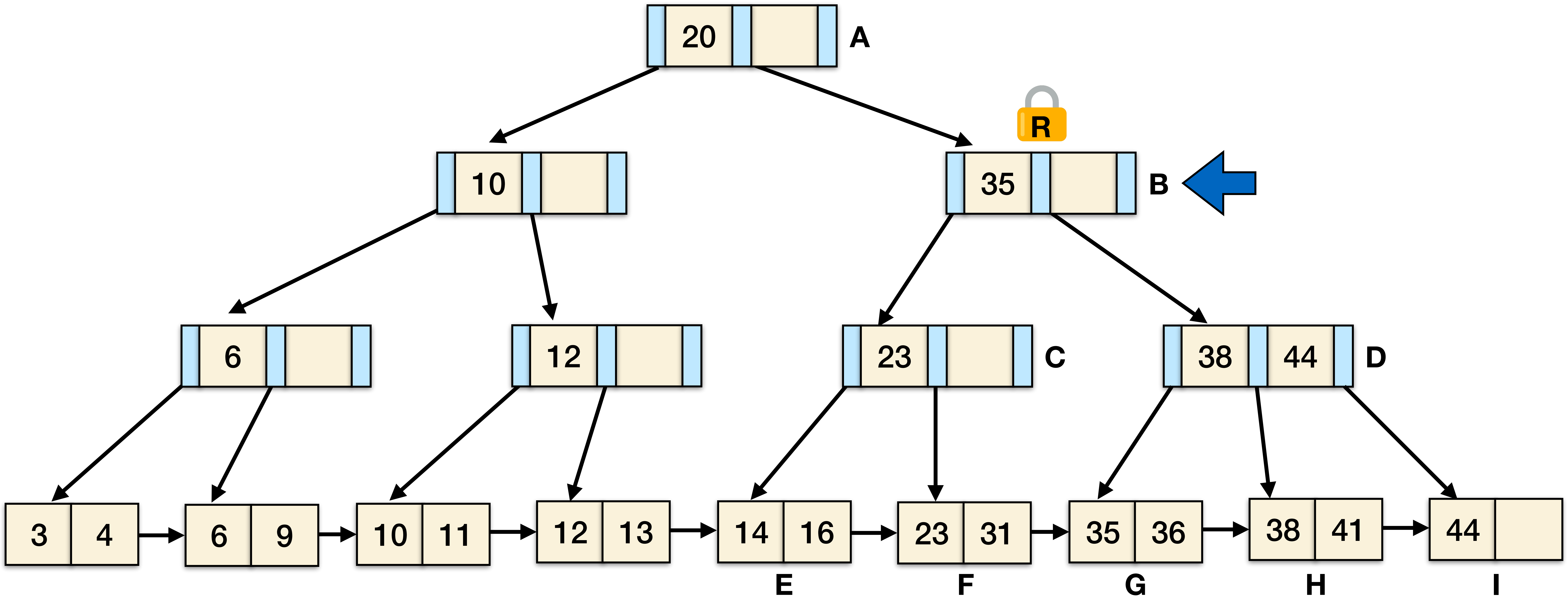


Example #1 - Find 38

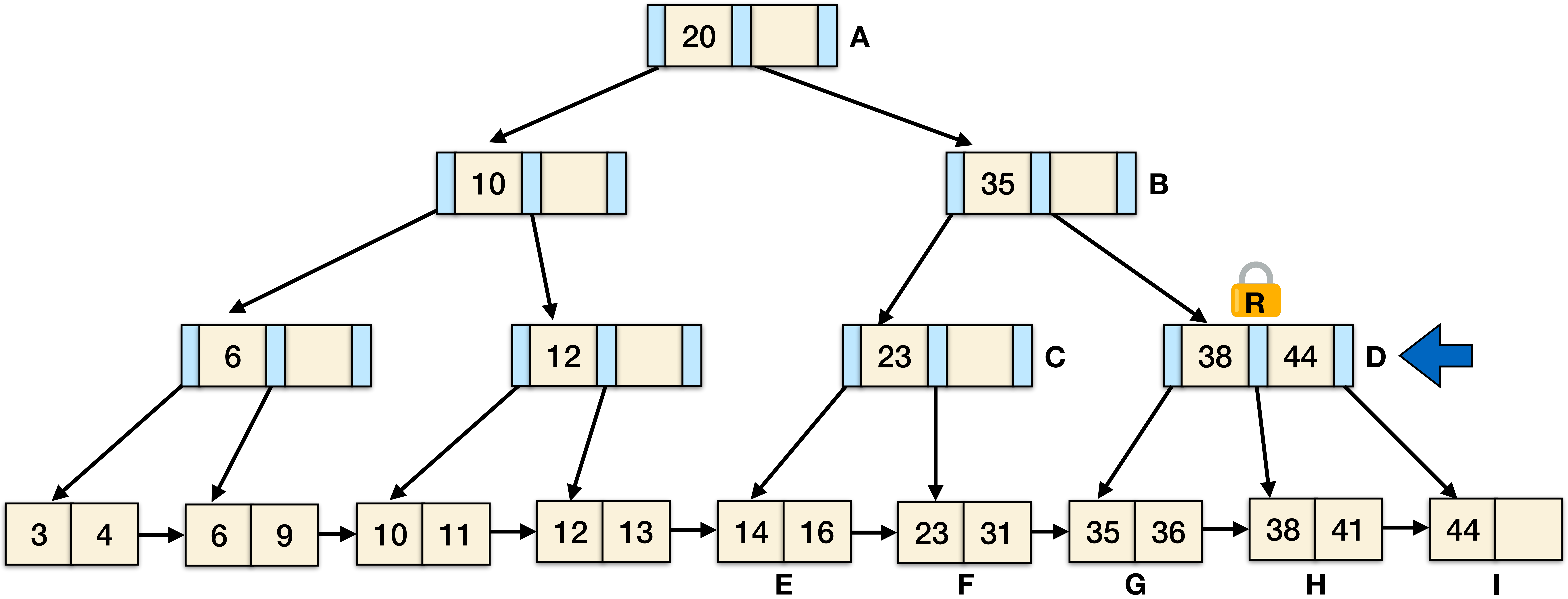


From Utah CS6530

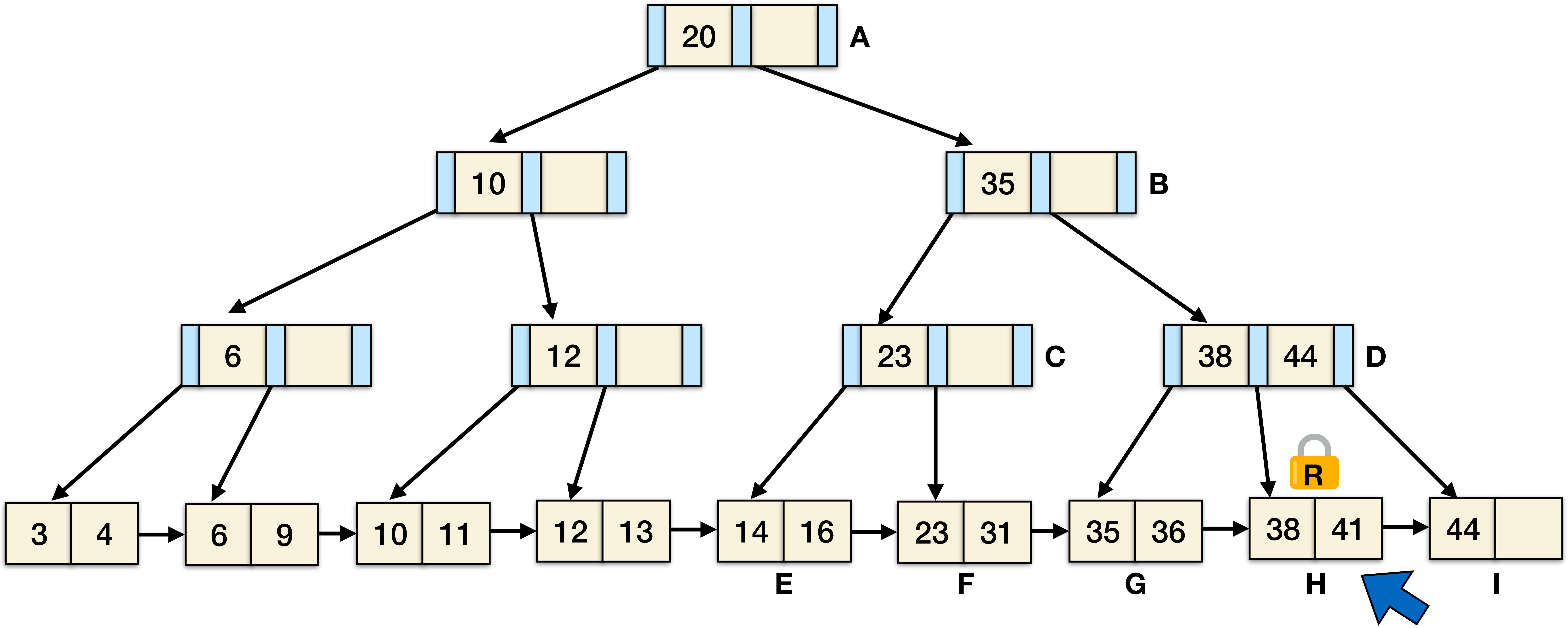
Example #1 - Find 38



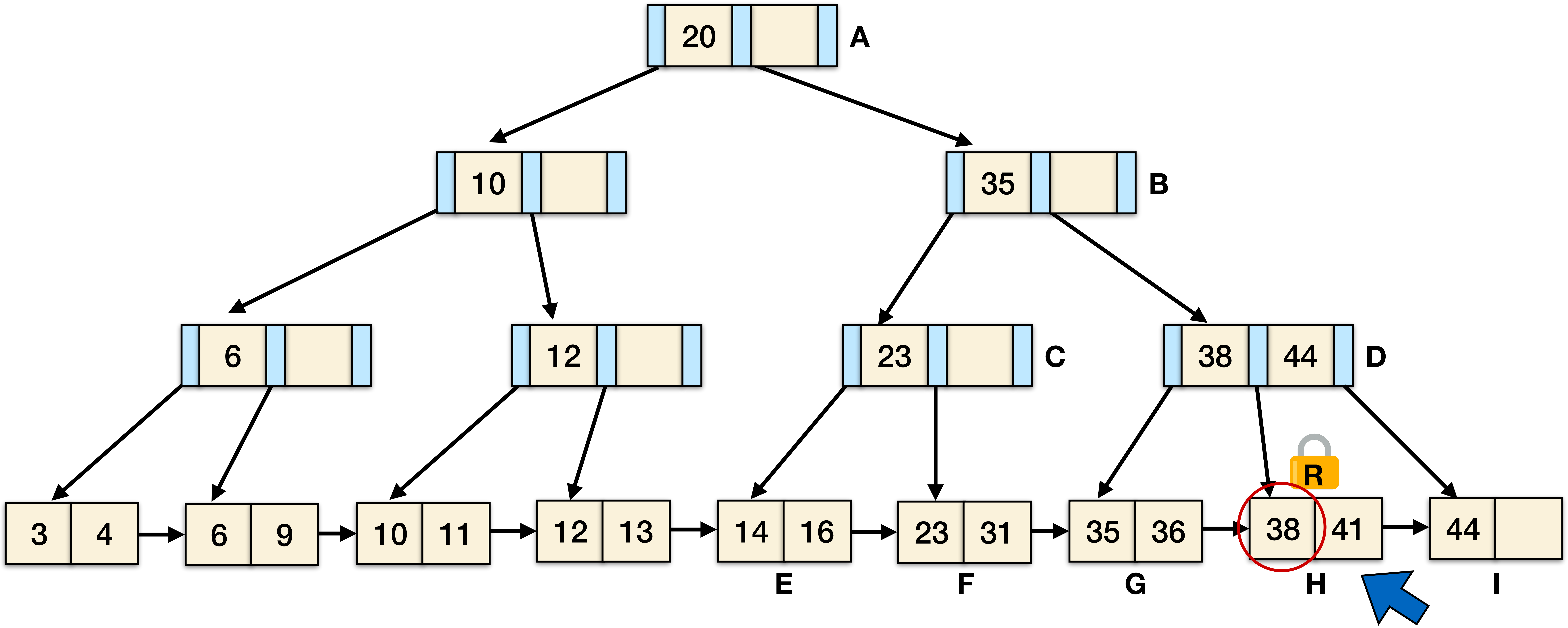
Example #1 - Find 38



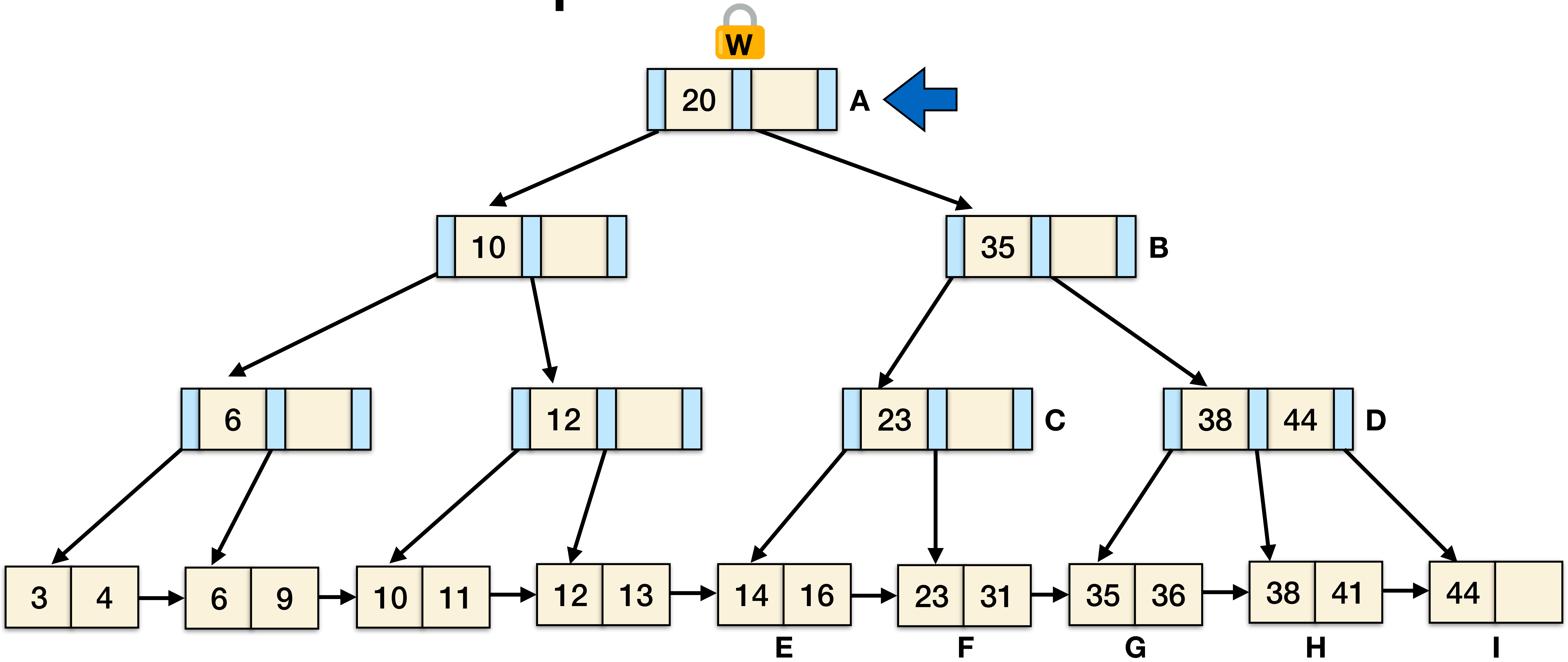
Example #1 - Find 38



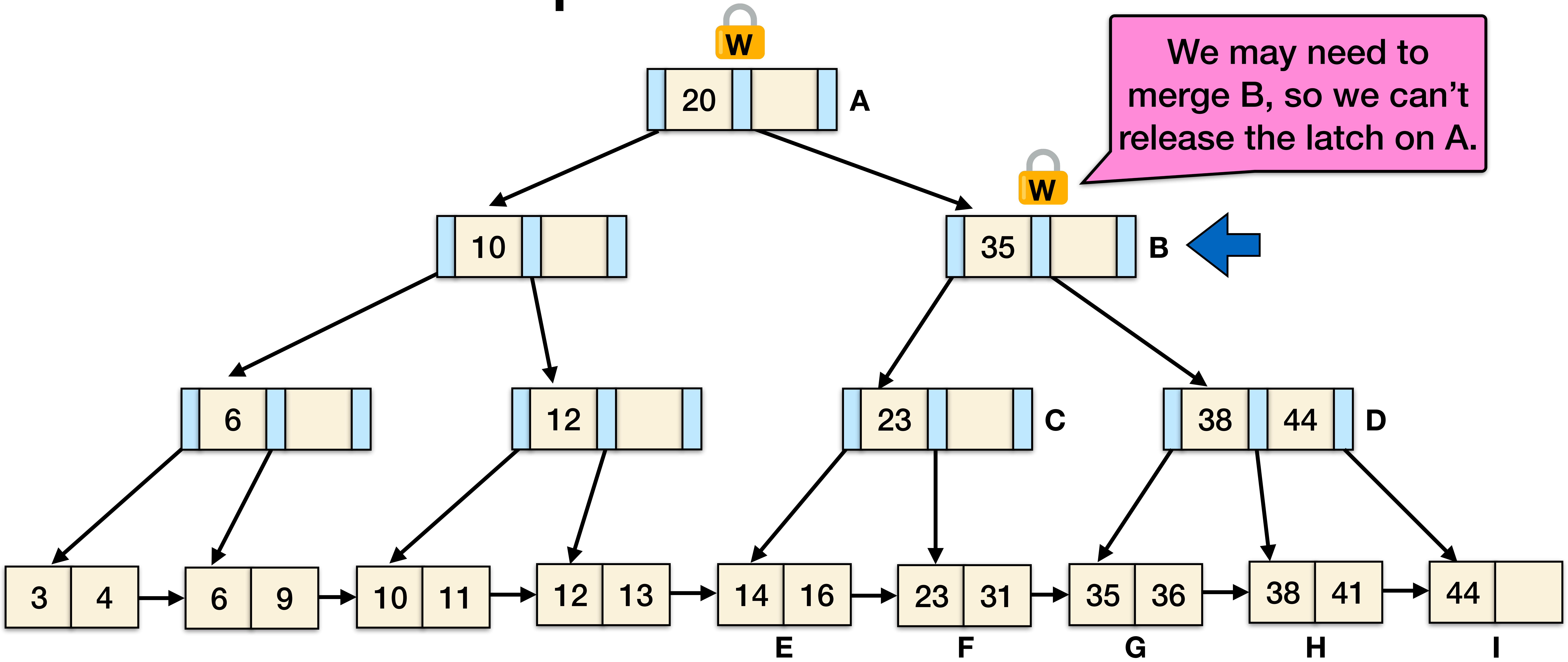
Example #1 - Find 38



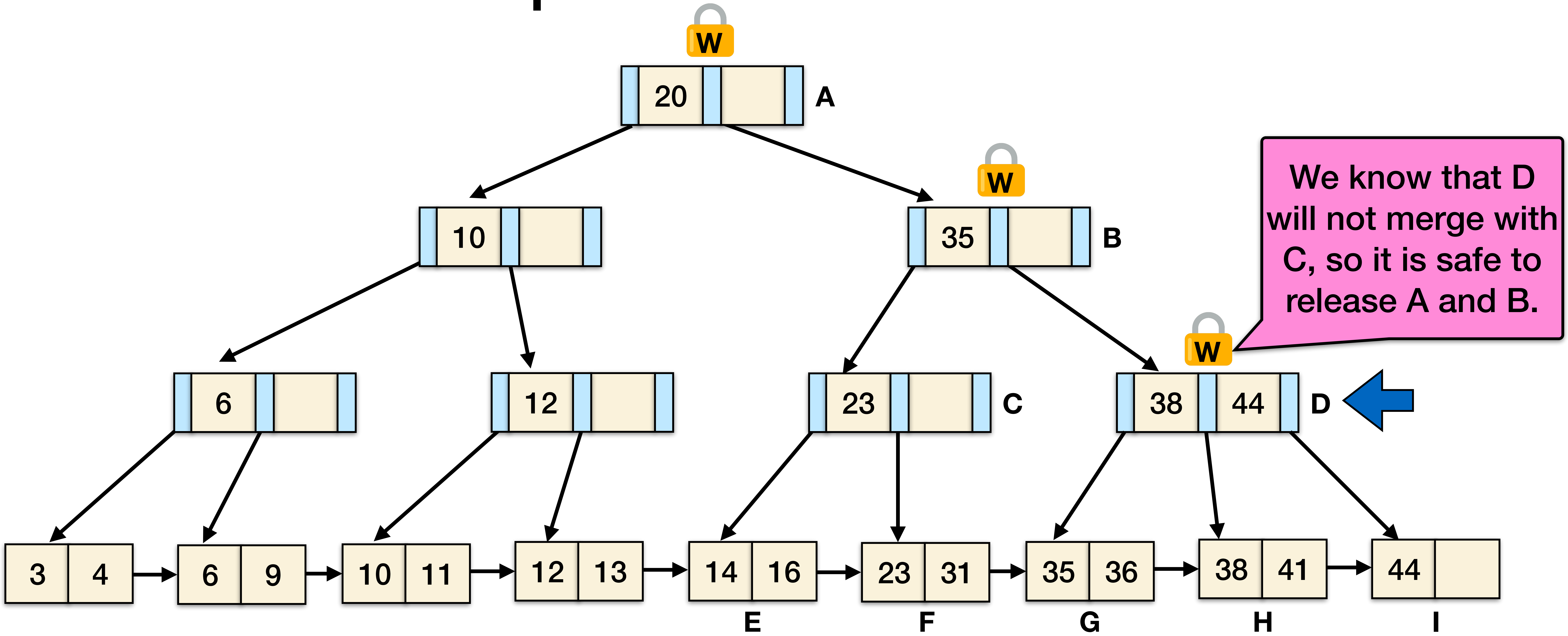
Example #2 - Delete 38



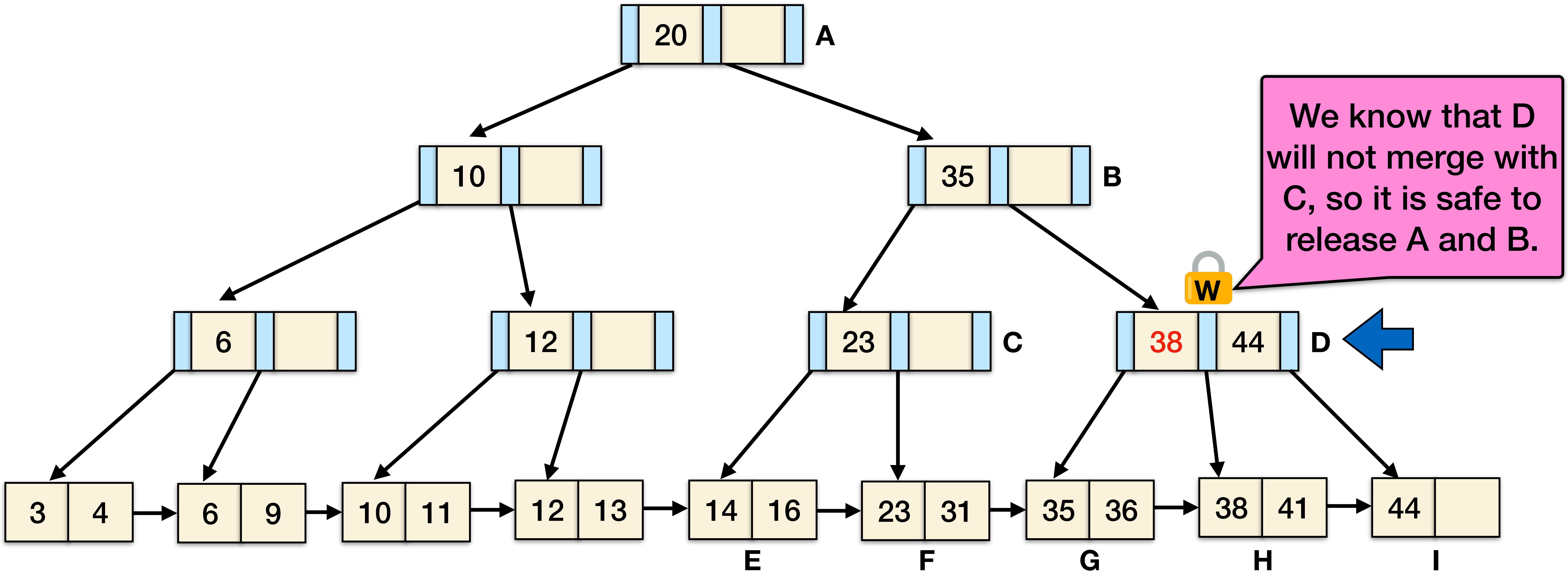
Example #2 - Delete 38



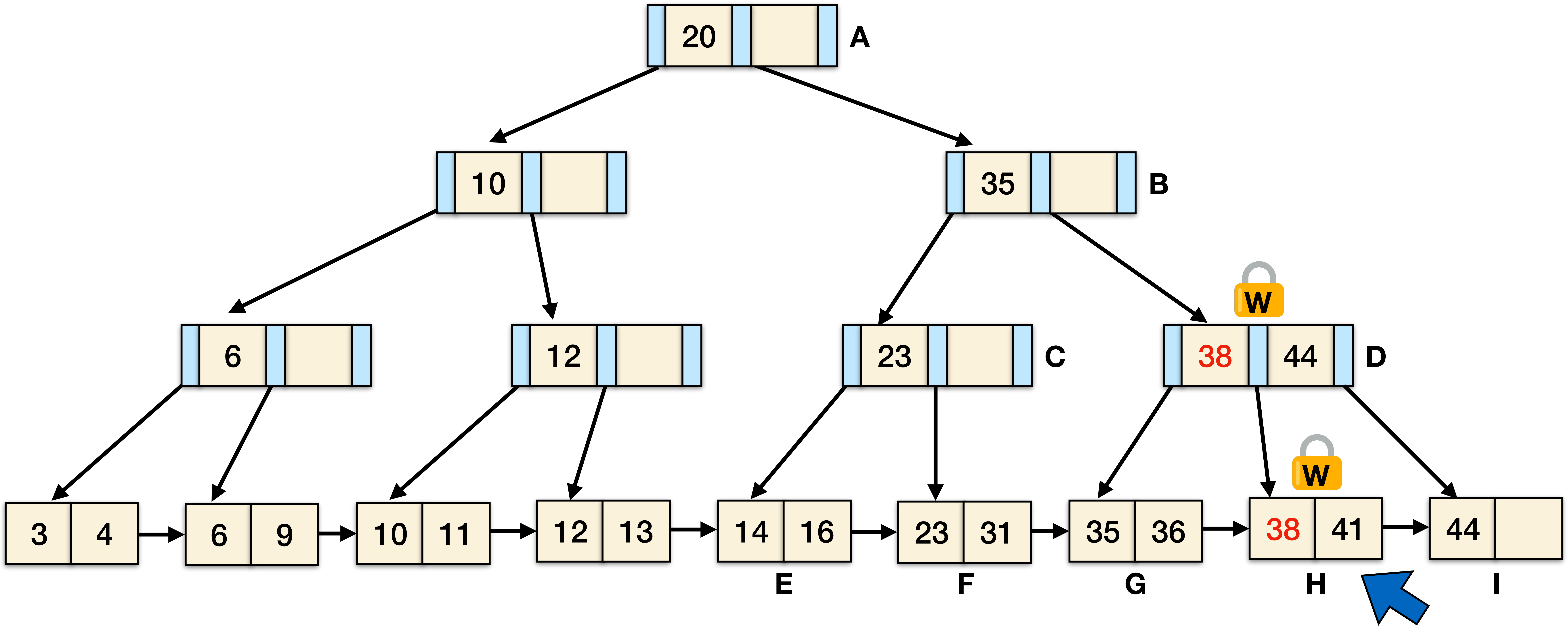
Example #2 - Delete 38



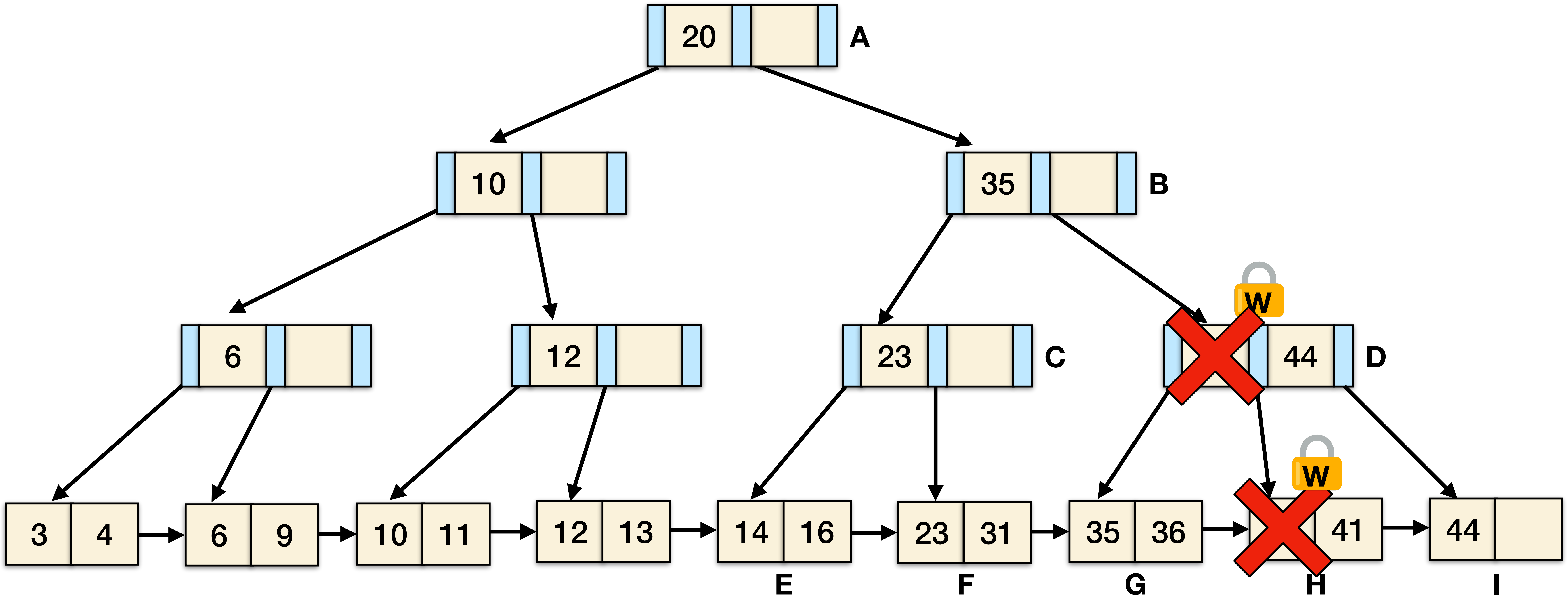
Example #2 - Delete 38



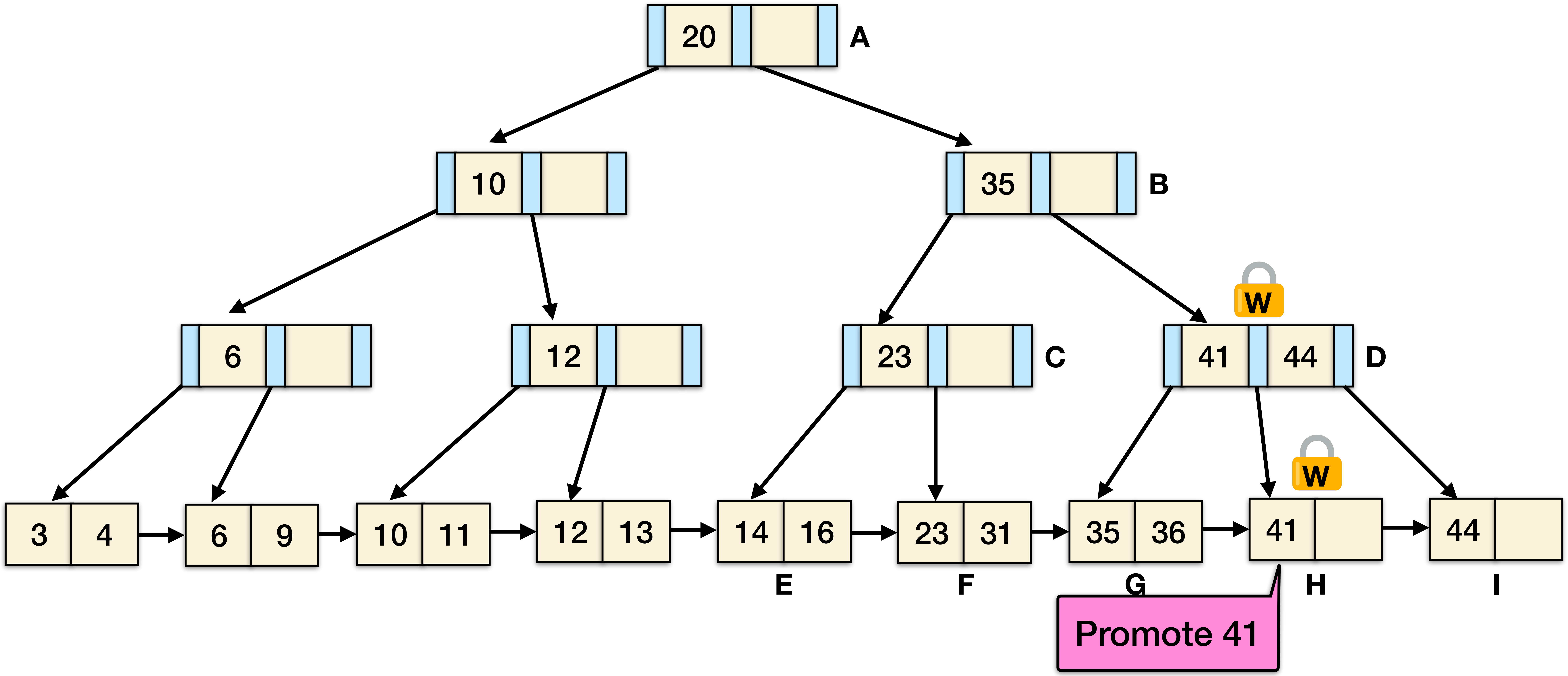
Example #2 - Delete 38



Example #2 - Delete 38



Example #2 - Delete 38



Example #2 - Delete 38