Welcome to the jungle of concurrent objects

Method calls on concurrent threads can overlap (in time).

As a result, an object may *never* be between method calls.

All possible interactions between method calls must be taken into account.

What does it mean for a concurrent object to be correct ?

Linearizability

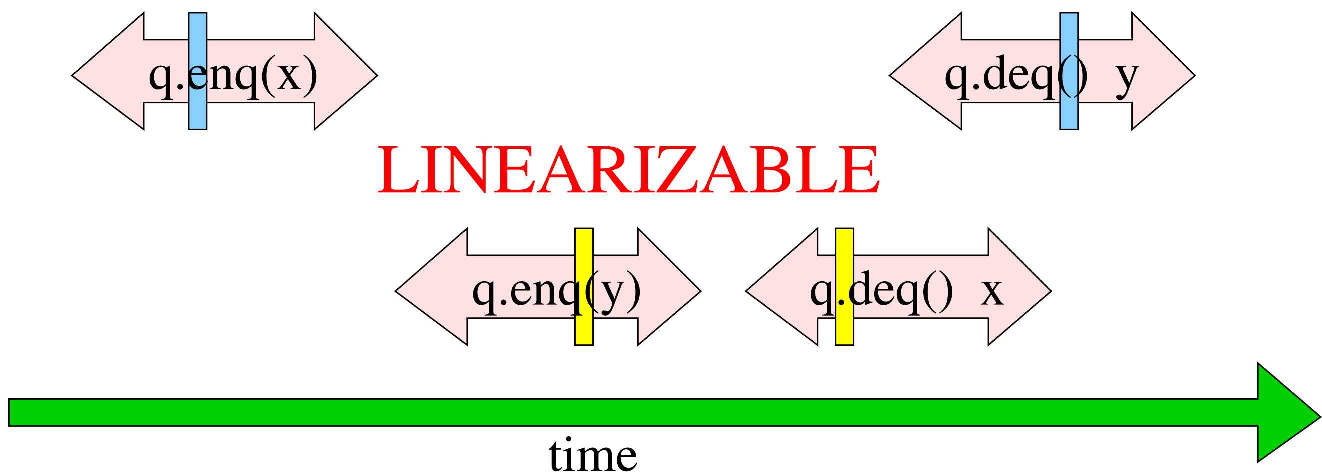
We order the method calls in an execution, by associating them to a moment in time when they are active.

That is: an execution on a concurrent object is linearizable if each method call in the execution:

- appears to take effect instantaneously,
- at a moment in time between its invocation and return events,
- in line with the system specification.

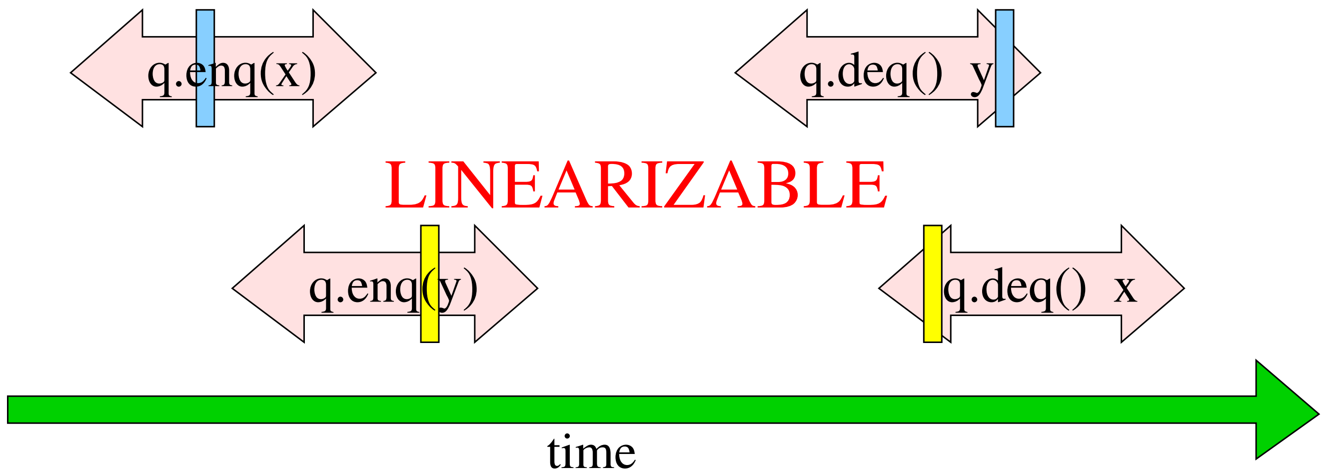
An object is linearizable if all its possible executions are linearizable.

Linearizability: example 1

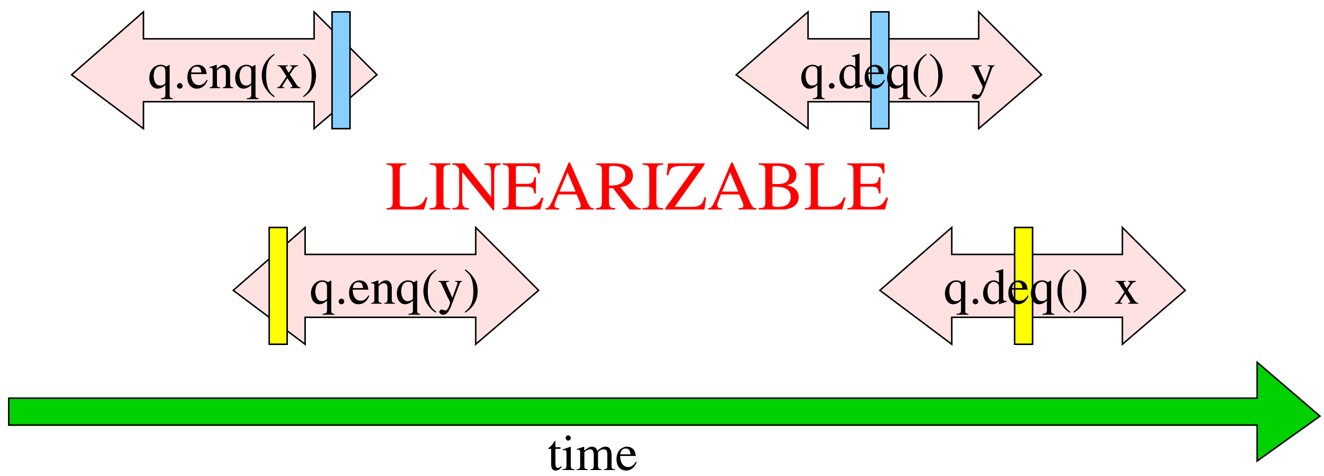


Since x is enqueued before y (in the FIFO queue),
it should also be dequeued before y.

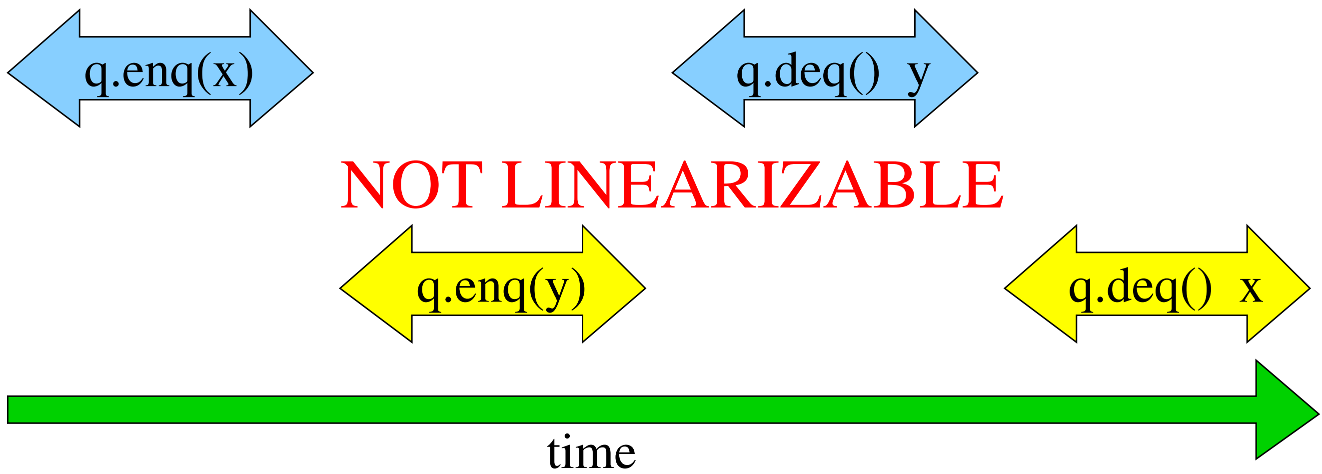
Linearizability: example 2



Linearizability: example 2



Linearizability: example 3



Linearization of unfinished method calls

For a method call that has an invocation but no return event, one can either:

- let it take effect before the end of the execution; or
- omit the method call from the execution altogether (i.e., it didn't take effect before the end of the execution).

Wait-free bounded FIFO queue for two threads

As an example, consider a *concurrent* bounded FIFO queue, with an enqueue (`q.enq(x)`) and a dequeue (`q.deq()`) method.

Interactions between these methods can be avoided by protecting the queue using a lock.

But this results in an inefficient implementation (recall *Amdahl's law*).

The next slide shows an implementation of a FIFO queue without locks for two threads: An enqueuer and a dequeuer.

It is wait-free: The enqueuer and dequeuer can progress by themself.

Wait-free bounded FIFO queue for two threads

```
class WaitFreeQueue⟨T⟩ {
  volatile int head, tail;
  T[] items;
  public WaitFreeQueue(int capacity) {
    items = T[] new Object[capacity];
    head = 0; tail = 0;
  }
  public void enq(T x) throws FullException {
    if tail - head == items.length throw new FullException();
    items[tail % items.length] = x; tail++;
  }
  public T deq() throws EmptyException {
    if tail == head throw new EmptyException();
    T y = items[head % items.length]; head++;
    return y;
  }
}
```

Wait-free bounded FIFO queue: correctness

Intuitively, this algorithm is correct for the following reasons:

- Only the *enqueuer* writes to `tail` and `items[...]`, and only the *dequeuer* writes to `head`.
- The condition `if tail - head == items.length` stops the *enqueuer* from overwriting an element in the queue before the *dequeuer* has read it.

  Here it is used that `head` is volatile, and the dequeuer only increases `head` after it read `items[head % items.length]`.
- The condition `if tail == head` stops the *dequeuer* from reading an element in the queue before the *enqueuer* has placed it in the queue.

  Here it is used that `tail` is volatile, and the enqueuer only increases `tail` after it wrote to `items[tail % items.length]`.

Question

Why don't we need to declare the slots in the `items` array volatile ?

Answer: Reading or writing to a volatile variable (here `head` or `tail`) imposes a memory barrier in which the entire cache is flushed/invalidated.

Wait-free bounded FIFO queue is linearizable

Linearization point of `enq()` is `if tail - head == items.length`.

(If this condition is *false*, the enqueued item can only be dequeued after `tail++`.)

Linearization point of `deq()` is `if tail == head`.

(If this condition is *false*, the dequeued item can only be overwritten after `head++`.)

So any execution of the wait-free bounded FIFO queue is linearizable.

Flawed bounded FIFO queue

In the dequeuer, let us swap the order of two program statements:
`head++; T item = items[(head - 1) % capacity];`

Let the `capacity` be 1.

Suppose the enqueuer performs the methods `enq(a)` and `enq(b)`, and the dequeuer performs the method `deq()`.

The following execution isn't linearizable:

```
enq:  tail - head == 0        enq:  tail - head == 0
enq:  items[0] = a            enq:  items[0] = b
enq:  tail = 1                deq:  y = b
deq:  tail - head == 1        deq:  return b
deq:  head = 1
```

Sequential consistency

For multiprocessor memory architectures,
linearizability is often considered too strict.

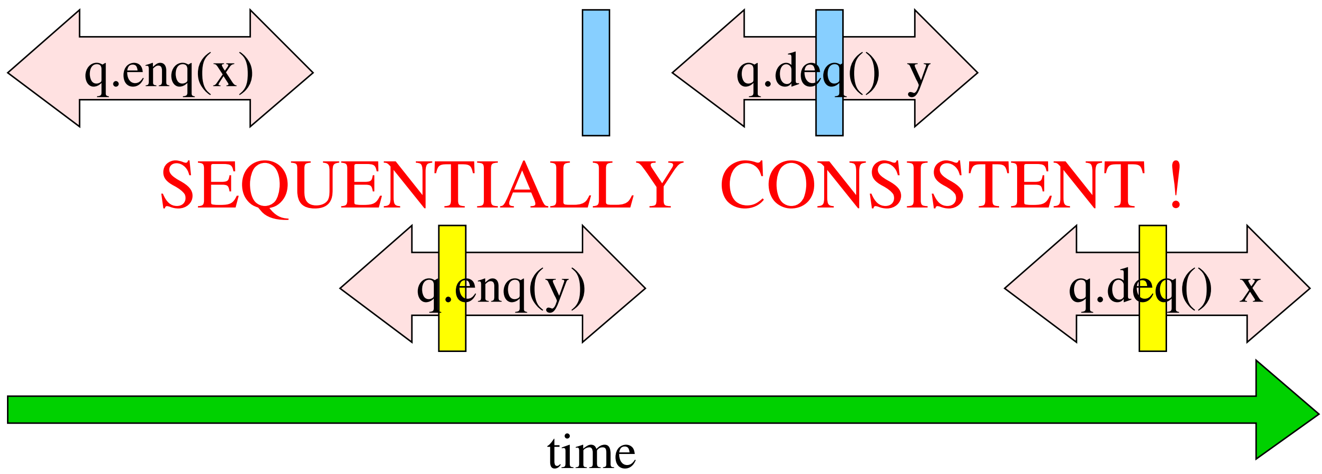
An execution on a concurrent object is sequentially consistent
if each method call in the execution:

- appears to take effect instantaneously,
- *in program order on each thread*,
- in line with the system specification.

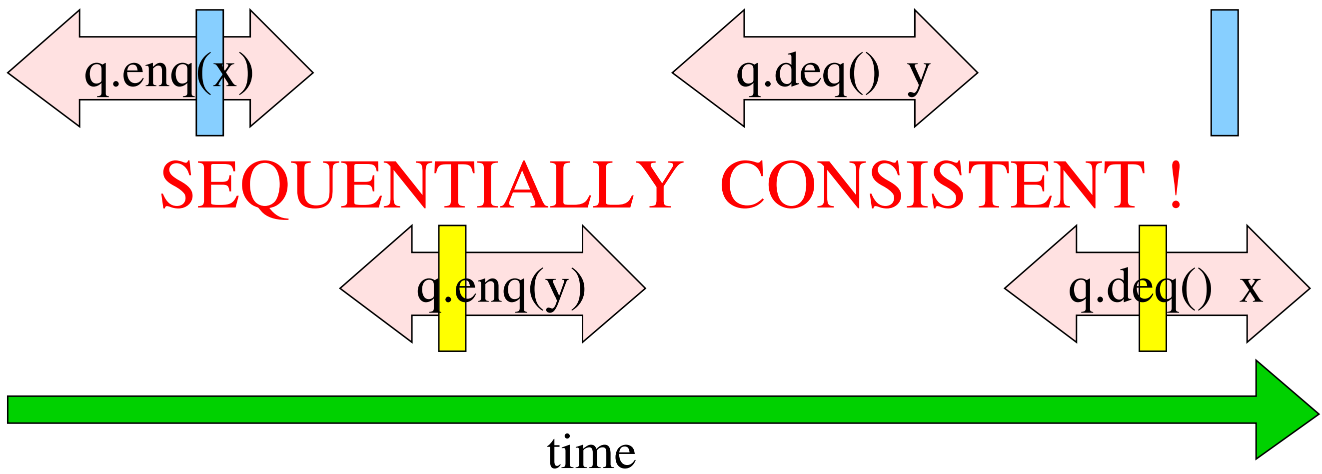
An object is sequentially consistent if all its possible executions
are sequentially consistent.

Sequentially consistent is less restrictive than linearizability,
because it allows method calls to take effect *after* they returned.

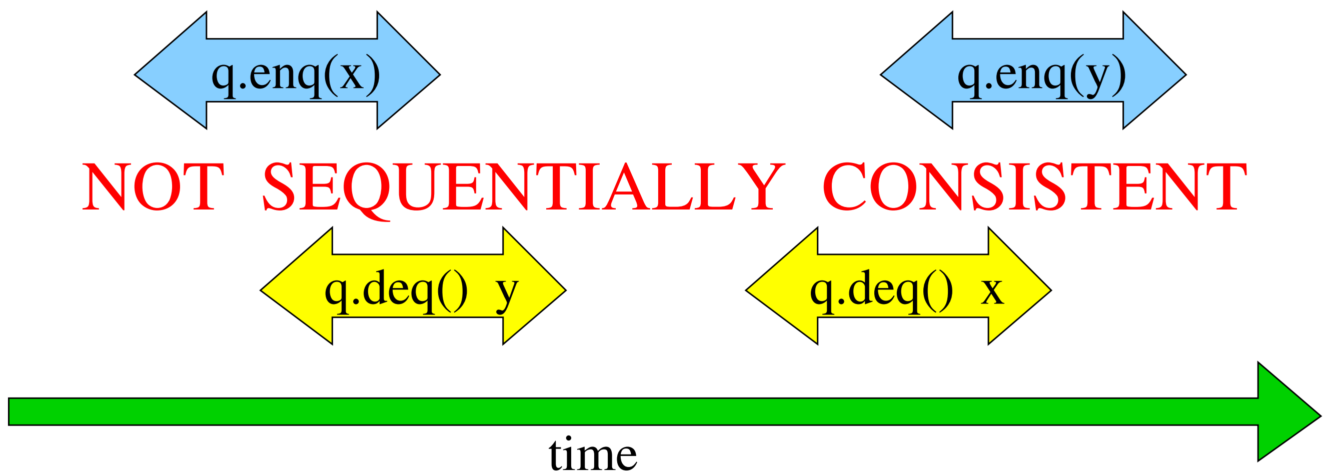
Sequential consistency: example 1



Sequential consistency: example 1



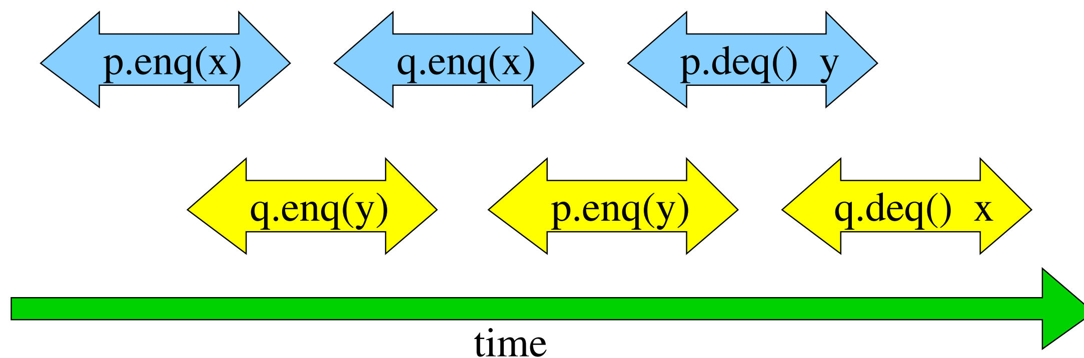
Sequential consistency: example 2



Compositionality

Linearizability is compositional:
the composition of linearizable objects is again linearizable.

By contrast, sequential consistency isn't compositional.



The executions for objects p and q are by themselves sequentially consistent, but their composition isn't.

Out-of-order execution

Most hardware architectures *don't* support sequential consistency, as it would outlaw widely used compiler optimizations.

Since memory access is very slow compared to processor speed, processors read/write to a cache, instead of directly to memory.

Processors can execute thousands of instructions, while writes in the cache are lazily written back to memory.

This makes sense because the vast majority of reads/writes isn't for synchronization.

Reads/writes for synchronization should be announced explicitly.

This comes with a performance penalty.

We stick to linearizability

Which correctness requirement is right for a given application ?

This depends on the needs of the application.

- A printer server can allow jobs to be reordered.
- A banking server better be sequentially consistent.
- A stock-trading server requires linearizability.

We will stick to linearizability, as it is well-suited for high-level objects.

Monitors

In Java, a monitor is associated with an object.

It combines data, methods and synchronization in one modular package.

Always at most one thread may be executing a method of the monitor.

A monitor provides mechanisms for threads to:

- temporarily give up exclusive access, until some *condition* is met,
- or *signal* to other threads that such a condition may hold.

Conditions

While a thread is waiting, say for a queue to become nonempty, the lock on this queue should be released.

Else other threads would never be able to dequeue an item.

In Java, a `Condition` object associated with a lock allows a thread to release the lock temporarily, by calling the `await()` method.

A thread can be awakened:

- by another thread (that performs `signal()` or `signalAll()`),
- or because some condition becomes true.

Conditions

An awakened thread must:

- try to reclaim the lock;
- when this has happened, retest the property it is waiting for;
- if the property doesn't hold, release the lock by calling `await()`.

wrong: `if` "*boolean expression*" `condition.await()`

correct: `while` "*boolean expression*" `condition.await()`

The thread may be woken up if another thread calls `condition.signal()` (which wakes up one thread) or `condition.signalAll()` (which wakes up all threads).

Bounded FIFO queue with locks and conditions: enqueue

```
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();
public void enq(T x) {
  lock.lock();
  try {
    while count == items.length
      notFull.await();
    items[tail] = x;
    if ++tail == items.length
      tail = 0;
    ++count;
    notEmpty.signal();
  } finally {
    lock.unlock();
  }
}
```

Bounded FIFO queue with locks and conditions: dequeue

```
public T deq() {
 lock.lock();
 try {
   while count == 0
     notEmpty.await();
   T y = items[head];
   if ++head == items.length
     head = 0;
   --count;
   notFull.signal();
   return y;
 } finally {
   lock.unlock();
 }
}
```

Lost-wakeup problem

`Condition` objects are vulnerable to lost wakeups: A thread may wait forever without realizing the condition it is waiting for has become true.

Example: In `enq()`, let `notEmpty.signal()` only be performed if the queue turns from empty to nonempty:

```
if ++count == 1 { notEmpty.signal(); }
```

A lost wakeup can occur if multiple dequeuers are waiting; only one dequeuer is woken up.

Programming practices to avoid lost wakeups:

- Signal *all* threads waiting for a condition (not just one).
- Specify a timeout for waiting threads.

Relaxing mutual exclusion

The strict mutual exclusion property of locks is often relaxed.

Three examples are:

- Readers-writers lock: Allows concurrent readers, while a writer disallows concurrent readers and writers.
- Reentrant lock: Allows a thread to acquire the same lock multiple times, to avoid deadlock.
- Semaphore: Allows at most c concurrent threads in their critical section, for some given capacity c.

Readers-writers lock: reader lock

```
class ReadLock implements Lock {
   public void lock() {
      lock.lock();
      try {
         while writer
            condition.await();
         readers++;
      } finally {
         lock.unlock();
      }
   }
```

Readers-writers lock: reader lock

```
    public void unlock() {
        lock.lock();
        try {
            readers--;
            if readers == 0
                condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

Readers-writers lock: writer lock

```
class WriteLock implements Lock {
  public void lock() {
    lock.lock();
    try {
      while (writer || readers > 0)
        condition.await();
      writer = true;
    } finally {
      lock.unlock();
    }
  }
```

Readers-writers lock: writer lock

```
  public void unlock() {
    lock.lock();
    try {
      writer = false;
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
}
```

The unlock method needs to grab the lock: A thread can only signal or start to await a condition if it owns the corresponding lock.

Readers-writers lock: writer lock

Question: What is the drawback of this writer lock ?

Answer: A writer can be delayed indefinitely by a continuous stream of readers.

Question: How can this be resolved ?

Answer: Allow one writer to set `writer = true` if `writer == false`.

Then other threads can't start to read.

Readers-writers lock: writer lock

```
class WriteLock implements Lock {
   public void lock() {
      lock.lock();
      try {
         while writer
            condition.await();
         writer = true;
         while readers > 0
            condition.await();
      } finally {
         lock.unlock();
      }
   }
```

The unlock method is as before.

Synchronized methods in Java

While a thread is executing a synchronized method on an object, other threads that invoke a synchronized method on this object block.

A synchronized method acquires the intrinsic lock of the object on which it is invoked, and releases the lock when it returns.

A synchronized method imposes a memory barrier:

- At the start, the cache is invalidated.
- At completion, modified fields in working memory are written back to shared memory.

Synchronized methods

Synchronized methods are *reentrant*.

Monitors are provided for synchronized methods:

- `wait()` causes a thread to wait until another thread notifies it of a condition change;
- `notify()` wakes up one waiting thread;
- `notifyAll()` wakes up all waiting threads.

Synchronized methods: drawbacks

Synchronized methods

1. aren't starvation-free
2. are rather coarse-grained
3. can give a false feeling of security
4. may use other locks than one might expect

Synchronized methods: drawback 3

Example: Instead of using a lock and conditions,
we make the `enq()` method on the bounded queue synchronized.

(So `await` and `signal` are replaced by `wait` and `notify`.)

One might expect that other method calls will always see
a proper combination of values of the variables `count` and `tail`.

But this is only guaranteed for other *synchronized* methods on queues.

Synchronized methods: drawback 4

A *static* synchronized method acquires the intrinsic lock of a *class* (instead of an object).

A static synchronized method on an *inner* class only acquires the intrinsic lock of the inner class, and not of the enclosing class.

Synchronized blocks

A synchronized block must specify the object that provides the intrinsic lock.

Example:

```
public void addName(String name) {
  synchronized(this) {
    lastName = name;
    nameCount++;
  }
  nameList.add(name);
}
```

Danger: Nested synchronized blocks may cause deadlock if they acquire locks in opposite order.

Barrier synchronization

Suppose a number of tasks must be completed before an overall task can proceed.

This can be achieved with barrier synchronization.

A barrier keeps track whether all threads have reached it.

When the last thread reaches the barrier, all threads resume execution.

Waiting at a barrier resembles waiting to enter a critical section.

It can be based on spinning (remote or on a locally cached copy), or on being woken up.

Synchronization approaches

Coarse-grained synchronization, in which each method call locks the entire object, can become a sequential bottleneck.

Four synchronization approaches for concurrent access to an object:

- Fine-grained: Split the object in *components* with their own locks.
- Optimistic: Search *without locks* for a certain component, lock it, *check* if it didn't change during the search, and only then adapt it.
- Lazy: Search without locks for a certain component, lock it, check if it isn't *marked*, mark it if needed, and only then adapt it.
- Non-blocking: Avoid locks, by *read-modify-write* operations.

List-based sets

We will show the four techniques on a running example: sets.

Consider a linked list in which each node has three fields:

- the actual `item` of interest
- the `key`, being the `item`'s (unique) hash code
- `next` contains a reference to the next node in the list

There are sentinel nodes `head` and `tail`.

Nodes in the list are sorted in key order.

`head` and `tail` carry the *smallest* and *largest* key, respectively.

An implementation of sets based on lists (instead of e.g. trees) will of course never have a very good performance...

List-based sets: methods

We define three methods:

- `add(x)`: Add x to the set;
  return *true* only if x wasn't in the set.
- `remove(x)`: Remove x from the set;
  return *true* only if x was in the set.
- `contains(x)`: Return *true* only if x is in the set.

These methods should be *linearizable* such that they act as on a sequential set (on a uniprocessor).

An abstraction map maps each *linked list* to the *set* of `items` that reside in a node reachable from `head`.

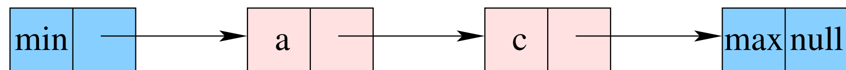
Coarse-grained synchronization

With coarse-grained locking, these methods lock the *entire* list.

The methods search without contention whether `x` is in the list.

add(x)

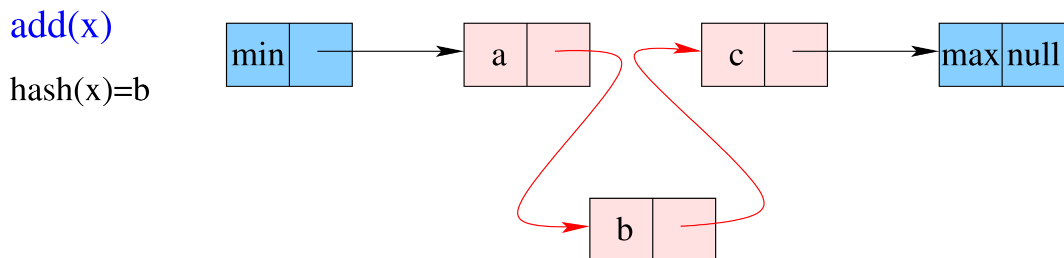
hash(x)=b



Coarse-grained synchronization

With coarse-grained locking, these methods lock the *entire* list.

The methods search without contention whether `x` is in the list.



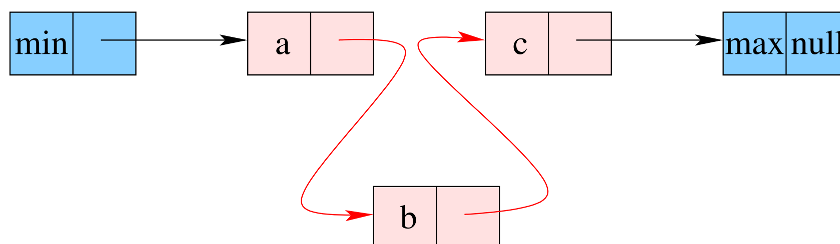
Coarse-grained synchronization

With coarse-grained locking, these methods lock the *entire* list.

The methods search without contention whether `x` is in the list.

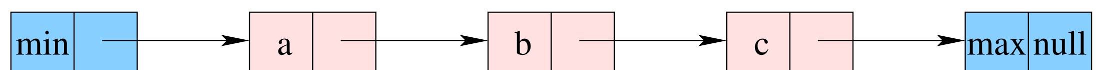
add(x)

hash(x)=b



remove(y)

hash(y)=c



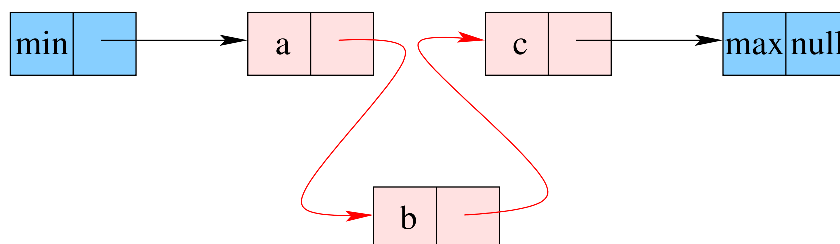
Coarse-grained synchronization

With coarse-grained locking, these methods lock the *entire* list.

The methods search without contention whether `x` is in the list.

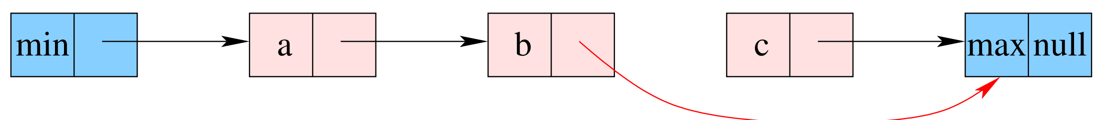
add(x)

hash(x)=b



remove(y)

hash(y)=c



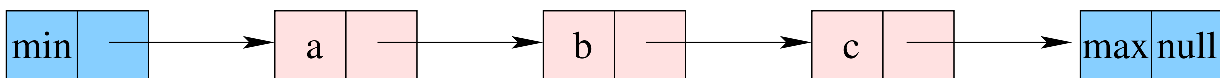
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a hand-over-hand fashion.

Example: A search for c.



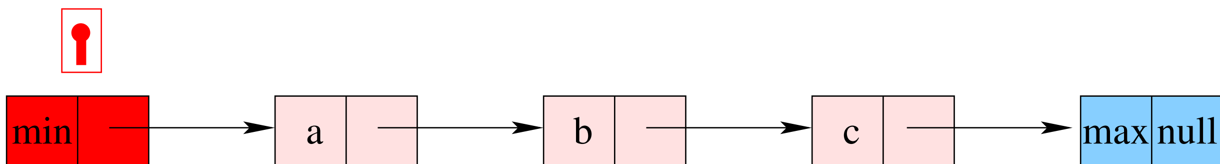
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a hand-over-hand fashion.

Example: A search for c.



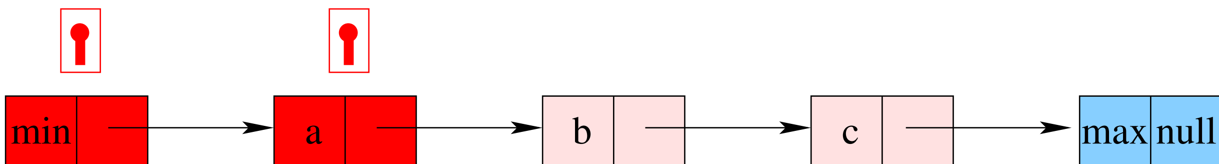
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a hand-over-hand fashion.

Example: A search for `c`.



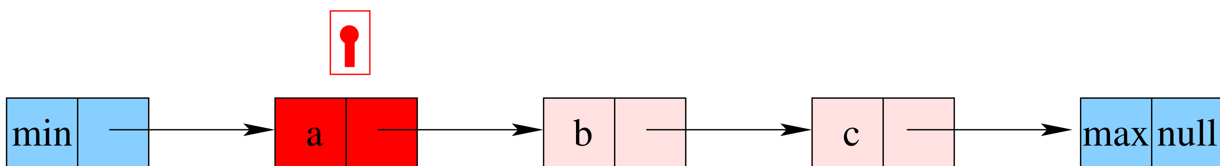
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order,
until they find `x` or conclude it isn't present.

Threads acquire locks in a hand-over-hand fashion.

Example: A search for c.



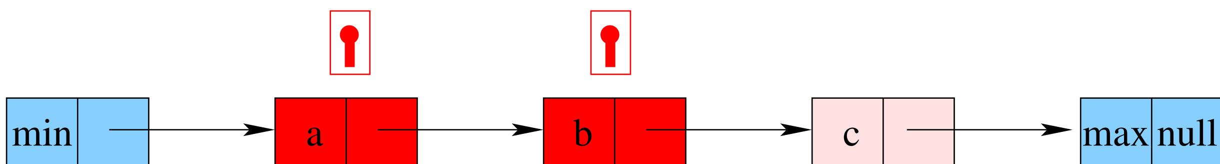
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a hand-over-hand fashion.

Example: A search for `c`.



Fine-grained synchronization: remove

`remove(x)` continues until:

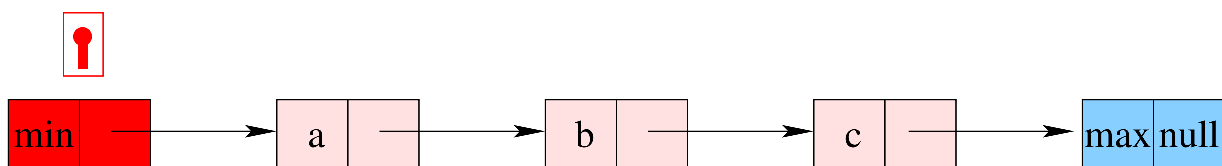
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that `x` isn't in the list, and returns *false*.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: remove

`remove(x)` continues until:

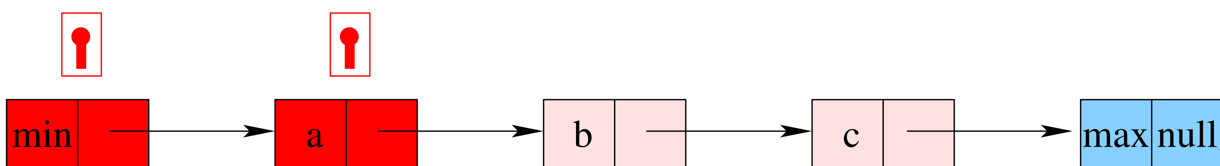
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that `x` isn't in the list, and returns *false*.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: remove

`remove(x)` continues until:

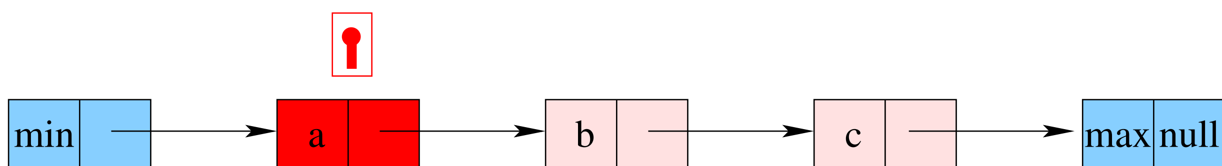
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that `x` isn't in the list, and returns *false*.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: remove

`remove(x)` continues until:

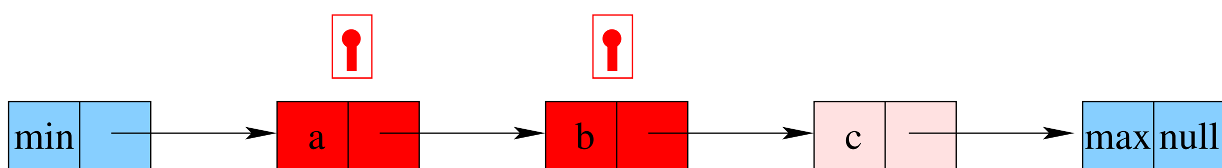
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that `x` isn't in the list, and returns *false*.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: remove

`remove(x)` continues until:

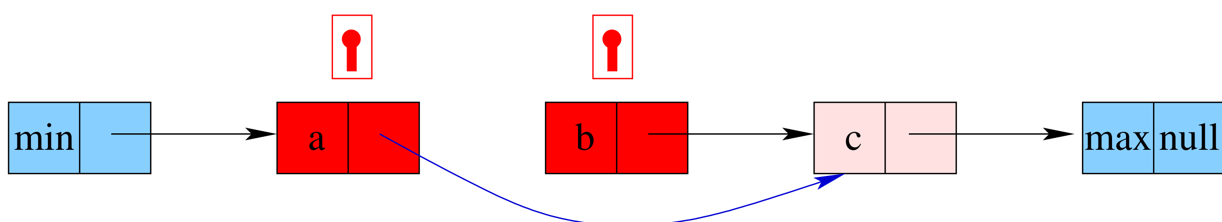
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that x isn't in the list, and returns *false*.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: remove

`remove(x)` continues until:

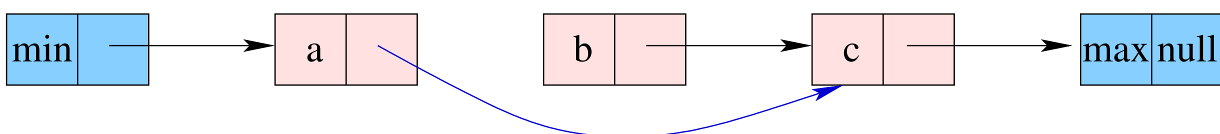
- either it locks a node with key `hash(x)`;

  then it removes this node by redirecting the link of the predecessor to the successor of this node, and returns *true*.

- or it locks a node with a key *greater than* `hash(x)`;

  then it concludes that `x` isn't in the list, and returns *false*.

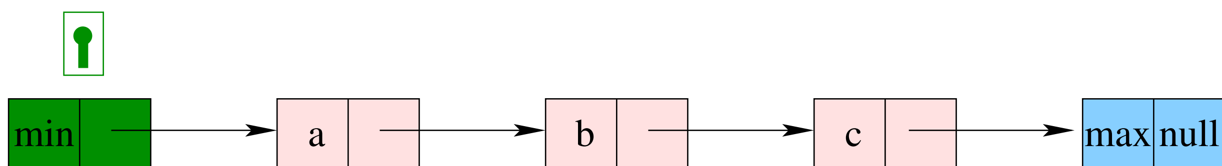
Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

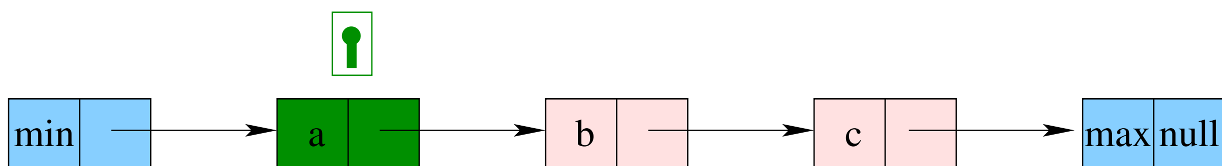
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

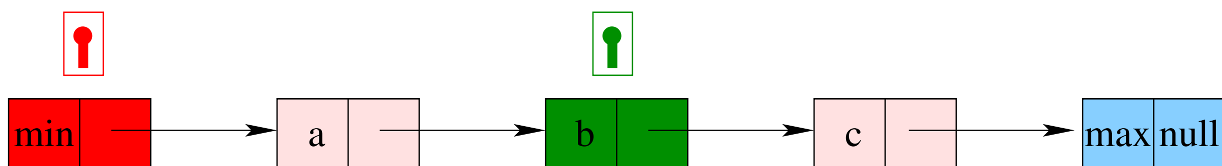
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

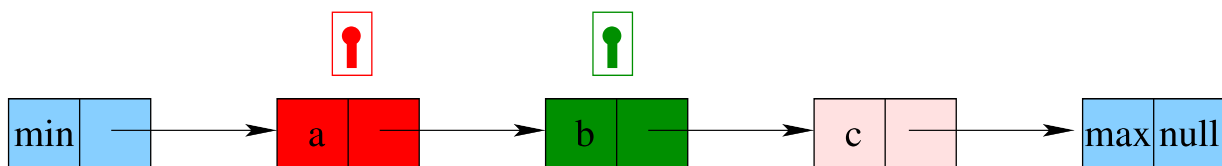
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

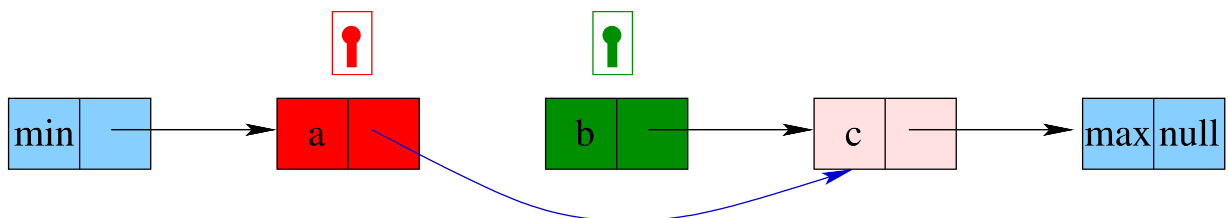
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

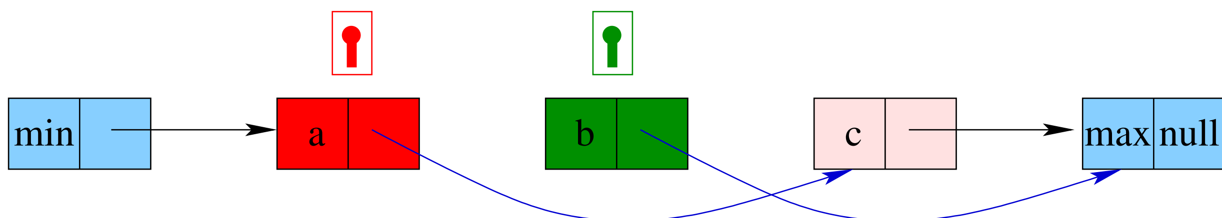
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: two locks are needed

If threads would hold only *one* lock at a time (instead of two), this algorithm would be incorrect.

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.

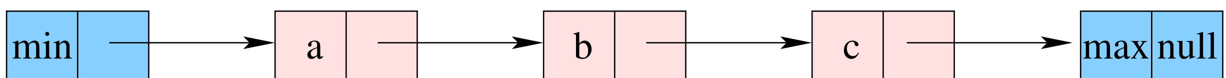


Node c isn't removed !

Fine-grained synchronization

Since threads are required to hold *two* locks at a time,
this problem doesn't occur.

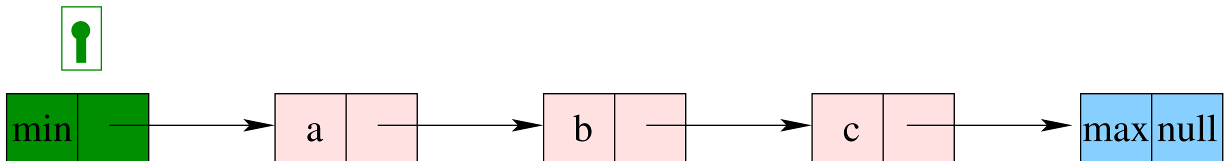
Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

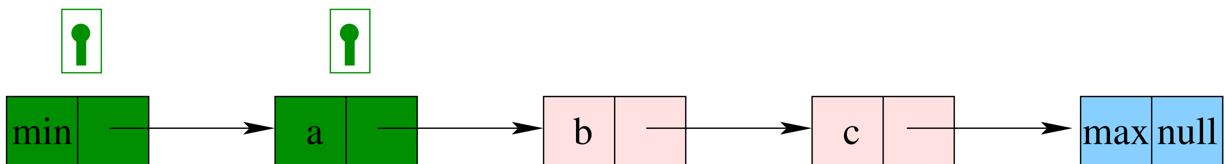
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

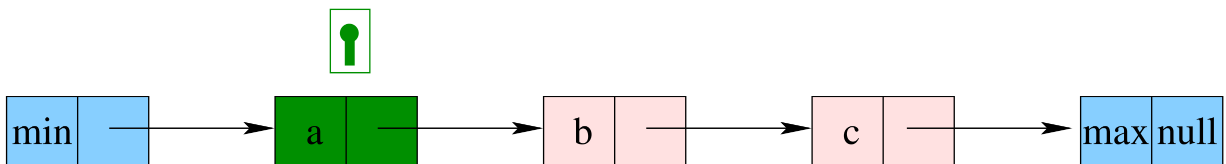
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

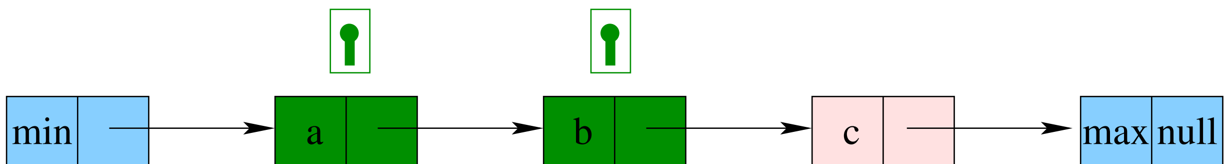
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time,
this problem doesn't occur.

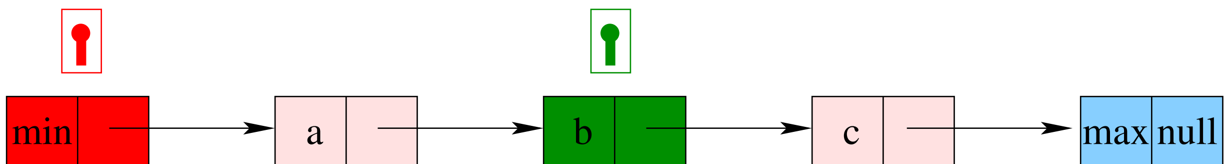
Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time,
this problem doesn't occur.

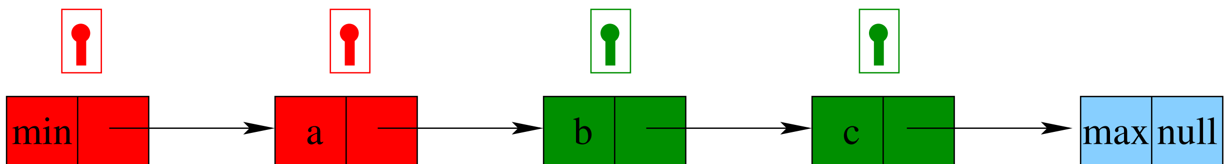
Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

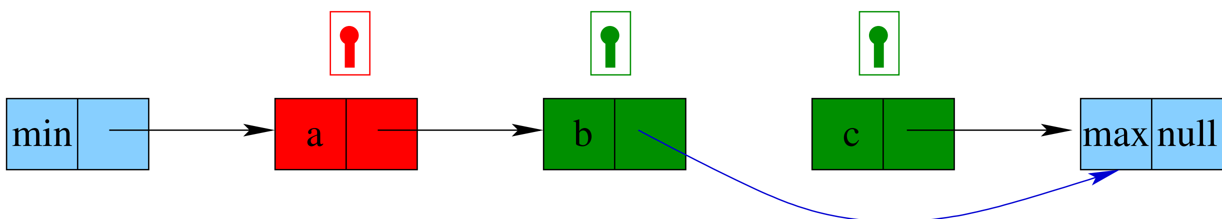
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

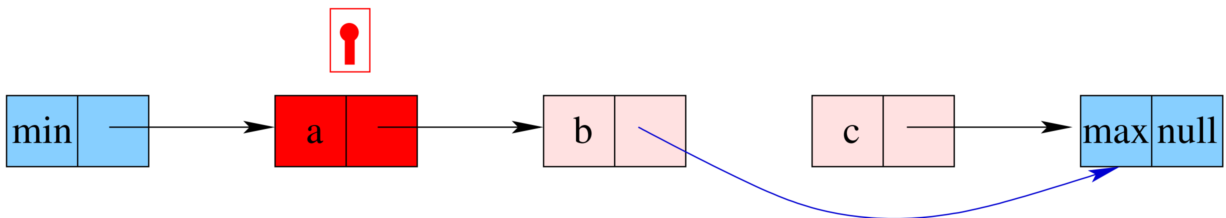
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time,
this problem doesn't occur.

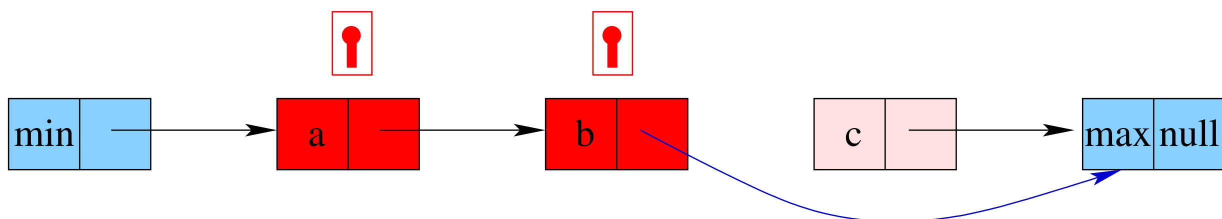
Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time,
this problem doesn't occur.

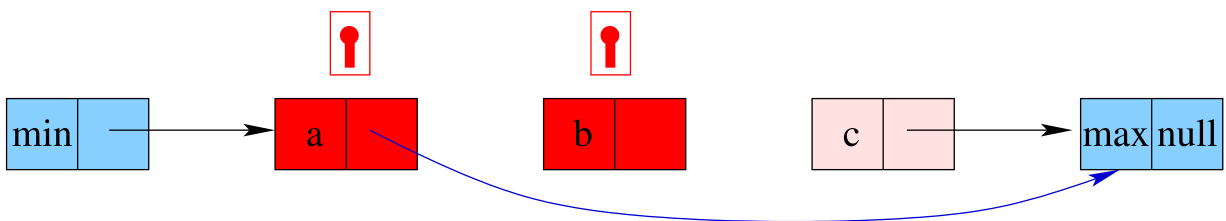
Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: add

`add(x)` continues until:

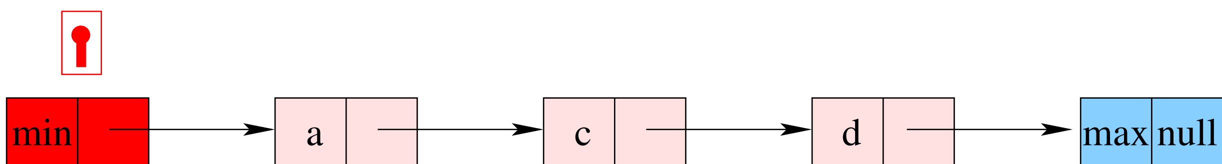
- either it locks a node with key `hash(x)`;

  then it concludes that `x` is in the list, and returns *false*.

- or it locks a node with a key $\mathtt{c} > \mathtt{hash(x)}$;

  then it redirects the link of the predecessor of `c` to a new node with $\mathtt{key} = \mathtt{hash}(\mathtt{x})$ and $\mathtt{next} = \mathtt{c}$, and returns *true*.

Example: We apply `add(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: add

`add(x)` continues until:

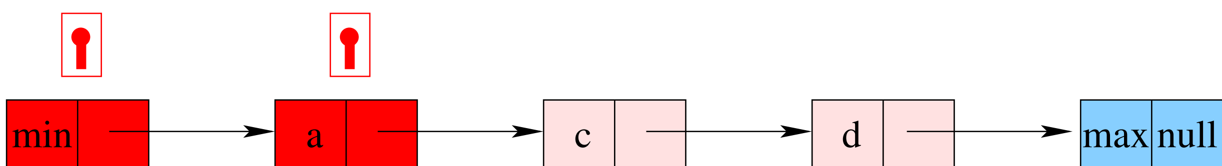
- either it locks a node with key `hash(x)`;

  then it concludes that `x` is in the list, and returns *false*.

- or it locks a node with a key $\mathtt{c} > \mathtt{hash(x)}$;

  then it redirects the link of the predecessor of `c` to a new node with $\mathtt{key} = \mathtt{hash}(\mathtt{x})$ and $\mathtt{next} = \mathtt{c}$, and returns *true*.

Example: We apply `add(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: add

add(x) continues until:

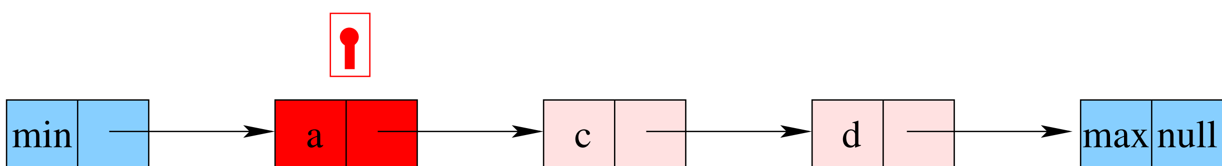
- either it locks a node with key hash(x);

  then it concludes that x is in the list, and returns *false*.

- or it locks a node with a key $c > $ hash(x);

  then it redirects the link of the predecessor of c to a new node with key = hash(x) and next = c, and returns *true*.

Example: We apply add(x) with hash(x)=b to the list below.



Fine-grained synchronization: add

`add(x)` continues until:

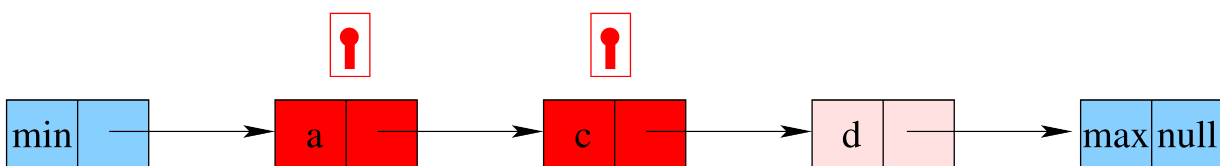
- either it locks a node with key `hash(x)`;

  then it concludes that `x` is in the list, and returns *false*.

- or it locks a node with a key $\texttt{c} > \texttt{hash(x)}$;

  then it redirects the link of the predecessor of `c` to a new node with $\texttt{key} = \texttt{hash}(\texttt{x})$ and $\texttt{next} = \texttt{c}$, and returns *true*.

Example: We apply `add(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: add

add(x) continues until:

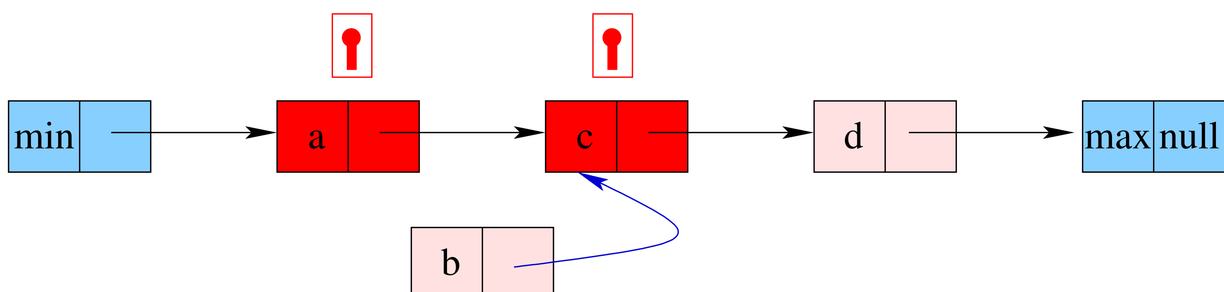
- either it locks a node with key hash(x);

  then it concludes that x is in the list, and returns *false*.

- or it locks a node with a key c $>$ hash(x);

  then it redirects the link of the predecessor of c to a new node with key $=$ hash(x) and next $=$ c, and returns *true*.

Example: We apply add(x) with hash(x)=b to the list below.



Fine-grained synchronization: add

`add(x)` continues until:

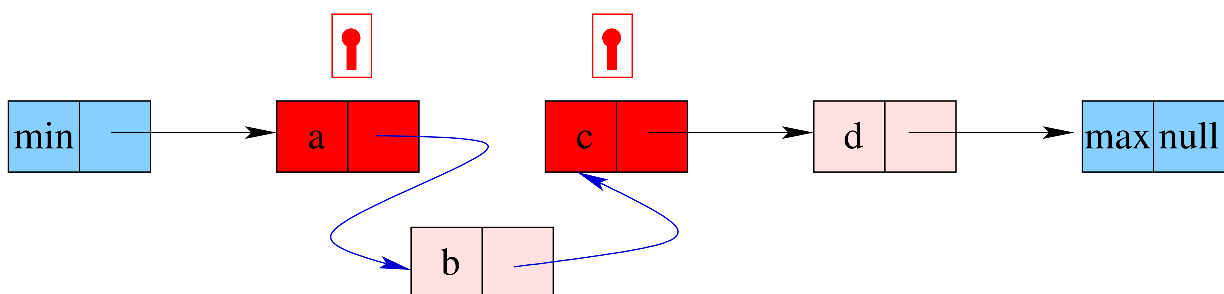
- either it locks a node with key `hash(x)`;

  then it concludes that `x` is in the list, and returns *false*.

- or it locks a node with a key $\texttt{c} > \texttt{hash(x)}$;

  then it redirects the link of the predecessor of `c` to a new node with $\texttt{key} = \texttt{hash}(\texttt{x})$ and $\texttt{next} = \texttt{c}$, and returns *true*.

Example: We apply `add(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: correctness

To remove a node, this node and its predecessor must be locked.

So while a node is being removed, this node, its predecessor
and its successor can't be removed.

And no nodes can be added between its predecessor and its successor.

Likewise, to add a node, this node's predecessor and successor
must be locked.

So while a node is being added, its predecessor and successor
can't be removed, and no other nodes can be added between them.

Fine-grained synchronization: linearization

The linearization points of `add` and `remove`:

- successful `add`: When the predecessor is redirected to the added node.
- successful `remove`: When the predecessor is redirected to the successor of the removed node.
- unsuccessful `add` and `remove`: When it is detected that the call is unsuccessful.

Fine-grained synchronization: progress property

The fine-grained synchronization algorithm is deadlock-free: always the thread holding the "furthest" lock can progress.

Moreover, the algorithm can be made starvation-free, by using for instance the bakery algorithm to ensure that any thread can eventually get the lock of `head`.

(Since calls can't overtake each other, for performance, calls on large keys should be scheduled first.)

Fine-grained synchronization: evaluation

Fine-grained synchronization allows threads to traverse the list in parallel.

However, it requires a chain of acquiring and releasing locks, which can be expensive.

And the result is a rather sequential implementation.

Optimistic synchronization

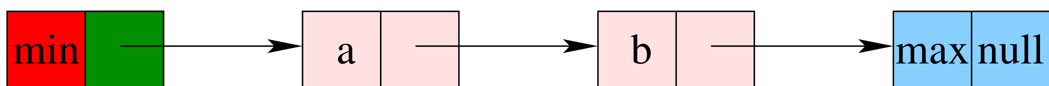
In optimistic synchronization, `add` and `remove` proceed as follows:

- Search *without locks* for a pair of nodes on which the method call can be performed, or turns out to be unsuccessful.
- Lock these nodes *(always predecessor before successor)*.
- Check whether the locked nodes are "correct":

  if the first locked node
  - is reachable from `head`, and
  - points to the second locked node.
- If this validation fails, then release the locks and start over.
- Else, proceed as in fine-grained synchronization.

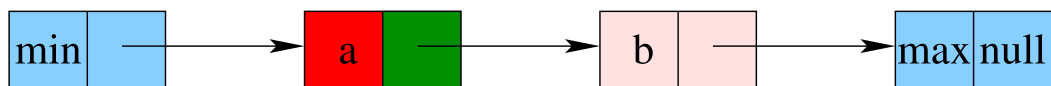
Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



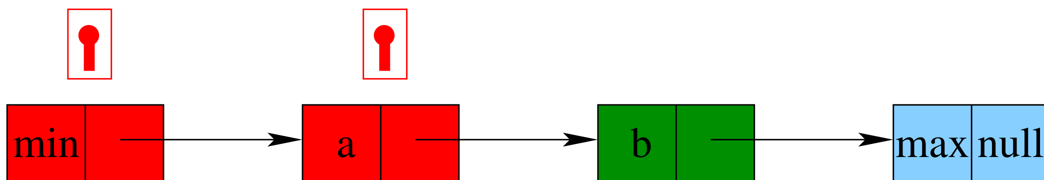
Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



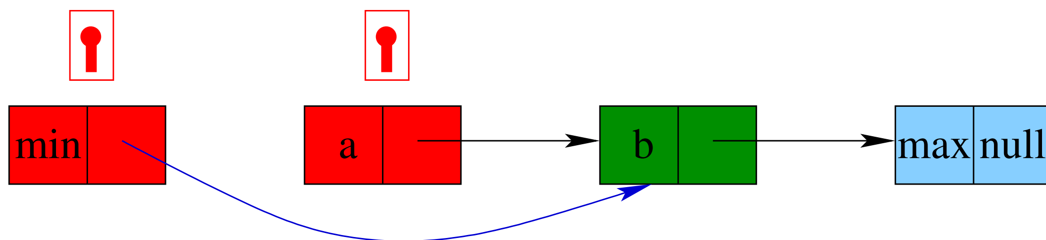
Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



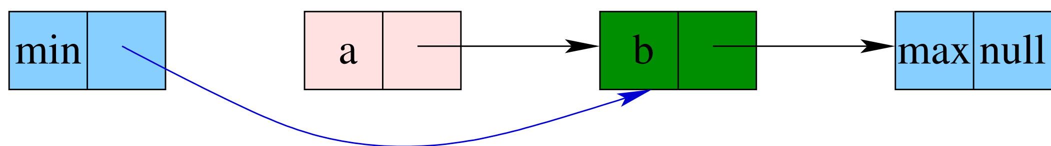
Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



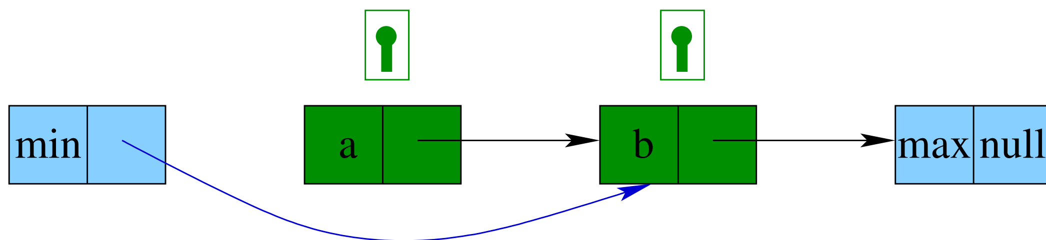
Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Optimistic synchronization: validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Validation shows that node `a` isn't reachable from `head`.

Optimistic synchronization: linearization

The linearization points of `add` and `remove`:

- successful `add`: When the predecessor is redirected to the added node.
- successful `remove`: When the predecessor is redirected to the successor of the removed node.
- unsuccessful `add` and `remove`: *When validation is completed successfully* (but the call itself is unsuccessful).

Optimistic synchronization: progress property

The optimistic synchronization algorithm is deadlock-free, because an unsuccessful validation means that another thread successfully completed an `add` or `remove`.

It is not starvation-free, because validation by a thread may be unsuccessful an infinite number of times.

Optimistic synchronization: evaluation

Optimistic synchronization in general requires less locking than fine-grained synchronization.

However, each method call traverses the list at least twice.

Question: How could validation be simplified ?

Lazy synchronization

A bit is added to each node.

If a reachable node has bit 1, it has been *logically* removed (and will be physically removed).