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).

Lazy synchronization: remove

In lazy synchronization, `remove(x)` proceeds as follows:

- Search (without locks) for a node `c` with a key $\geq$ `hash(x)`.
- Lock its predecessor `p` and `c` itself.
- Check whether `p` (1) isn't marked, and (2) points to `c`.
- If this validation fails, then release the locks and start over.
- Else, if the key of `c` is greater than `hash(x)`, return *false*.

  If the key of `c` equals `hash(x)`:

  - mark `c`,
  - redirect `p` to the successor of `c`, and
  - return *true*.

  Release the locks.

Lazy synchronization: add

`add(x)` proceeds similarly:

- Search for a node `c` with a key $\geq$ `hash(x)`.
- Lock its predecessor `p` and `c` itself.
- Check whether `p` (1) isn't marked, and (2) points to `c`.
- If this validation fails, then release the locks and start over.
- Else, if the key of `c` equals `hash(x)`, return *false*.

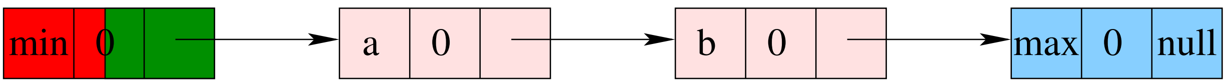
  If the key of `c` is greater than `hash(x)`:

  - create a node `n` with key `hash(x)`, value `x`, bit 0, and link to `c`,
  - redirect `p` to `n`, and
  - return *true*.

  Release the locks.

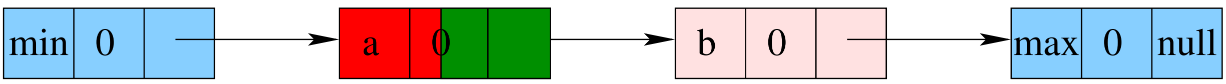
Lazy synchronization: validation is needed

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



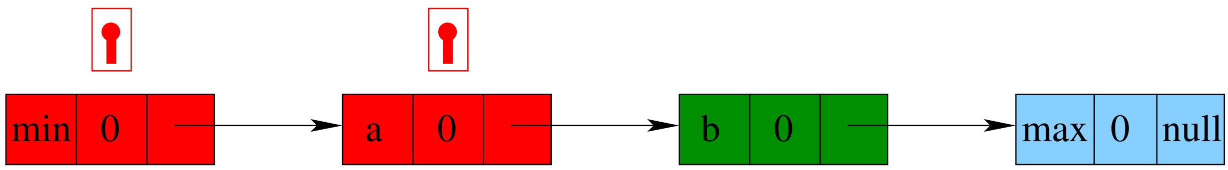
Lazy synchronization: validation is needed

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



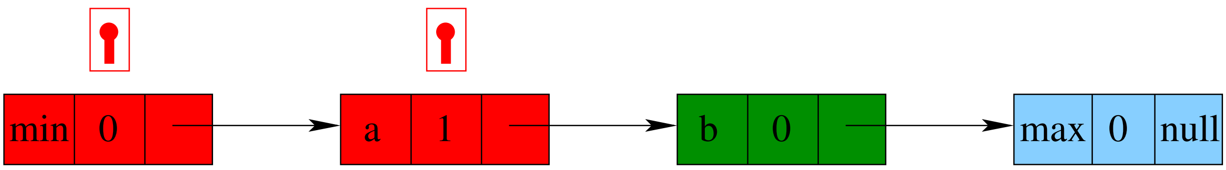
Lazy synchronization: validation is needed

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



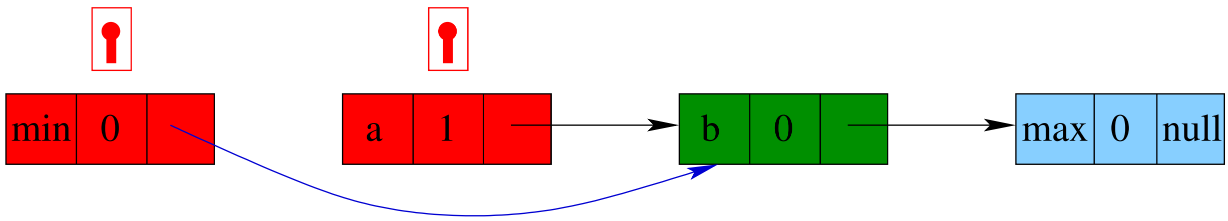
Lazy synchronization: validation is needed

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



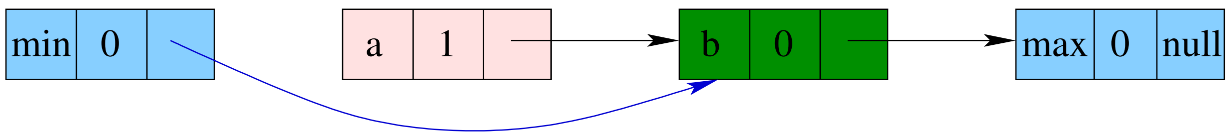
Lazy synchronization: validation is needed

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



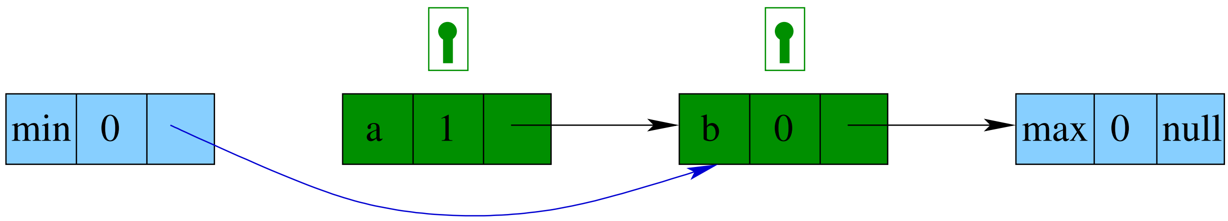
Lazy synchronization: validation is needed

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



Lazy 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 is marked for removal.

Lazy synchronization: contains

`contains(x)` doesn't require locks:

- Search for a node with the key `hash(x)`.
- If no such node is found, return *false*.
- If such a node is found, check whether it is marked.
- If so, return *false*, else return *true*.

Lazy synchronization: linearization

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

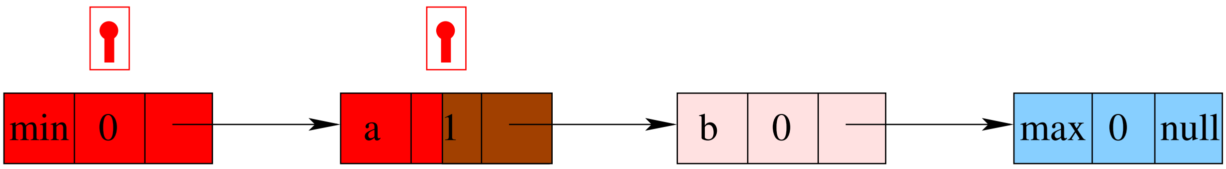
The linearization points:

- successful `add`: When the predecessor is redirected to the added node.
- successful `remove`: *When the mark is set*.
- successful `contains`: When the (unmarked) node is found.
- unsuccessful `add` and `remove`: When validation is completed successfully.
- unsuccessful `contains`: ???

Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

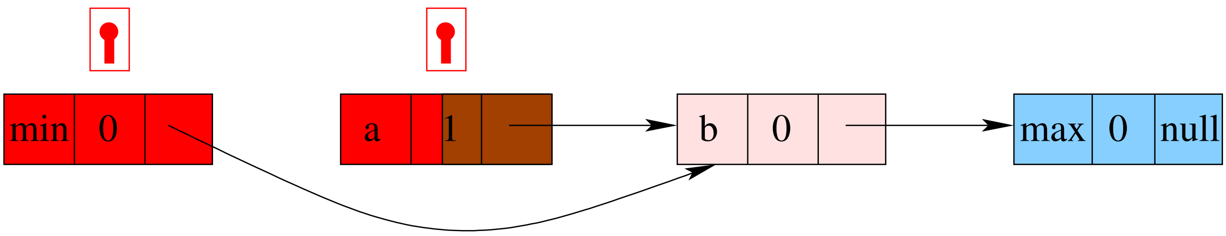
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

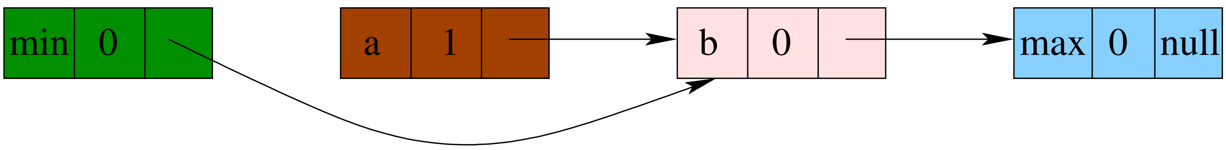
- remove(x) with hash(x)=a and contains(y) with hash(y)=b are being executed
- remove(y) and add(y) are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

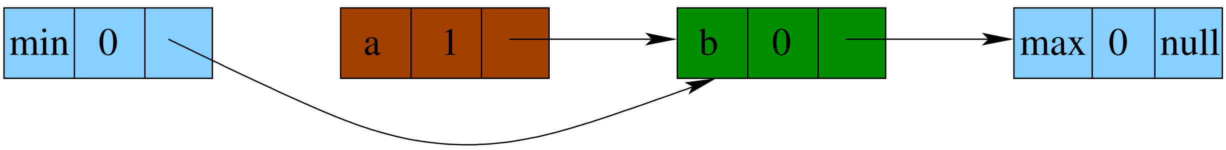
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

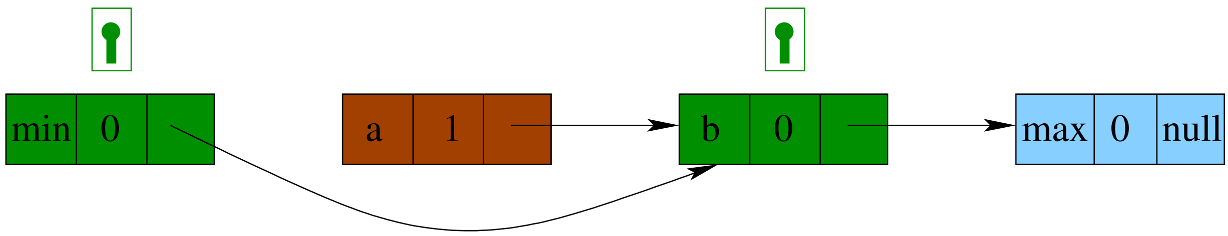
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

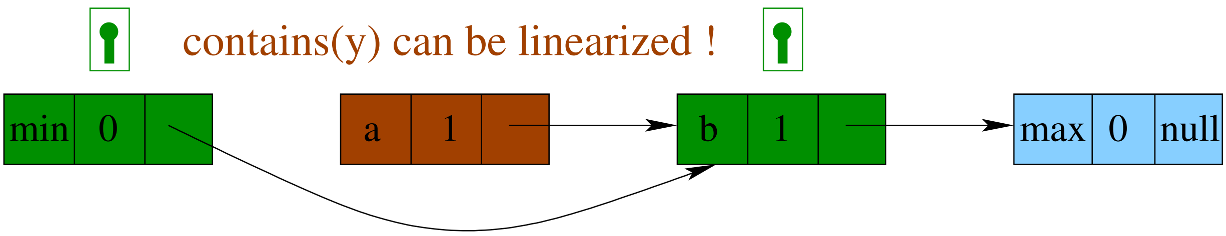
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

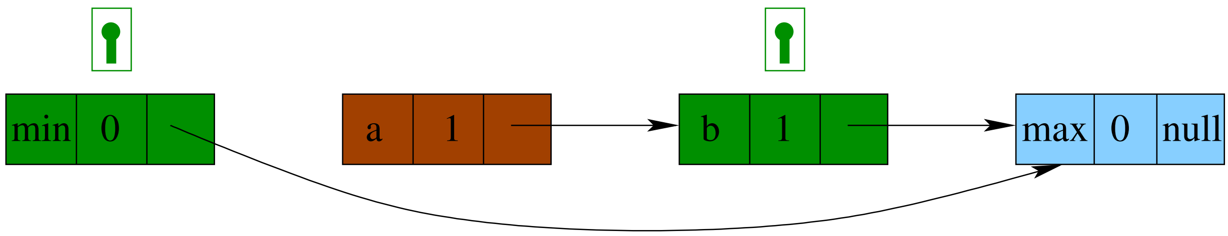
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

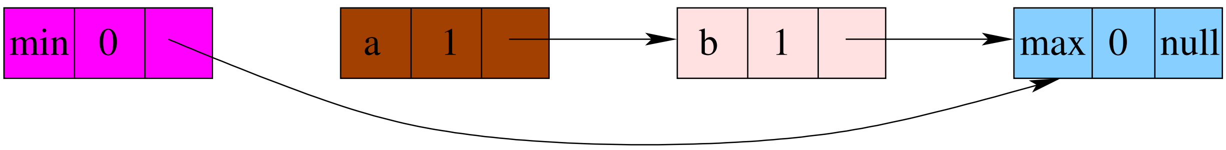
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

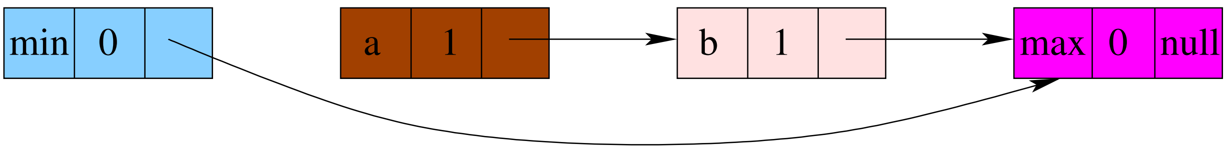
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

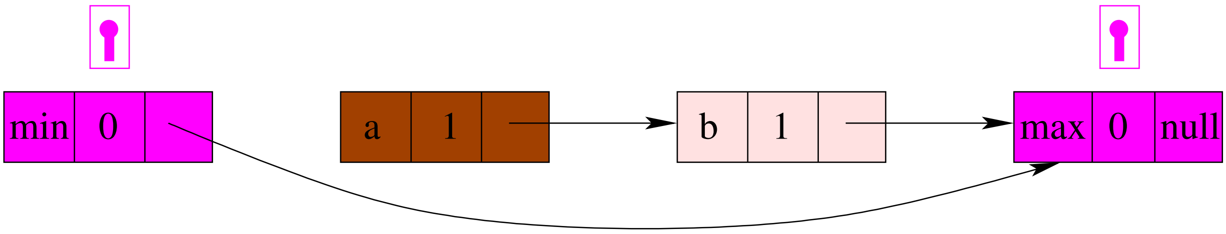
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

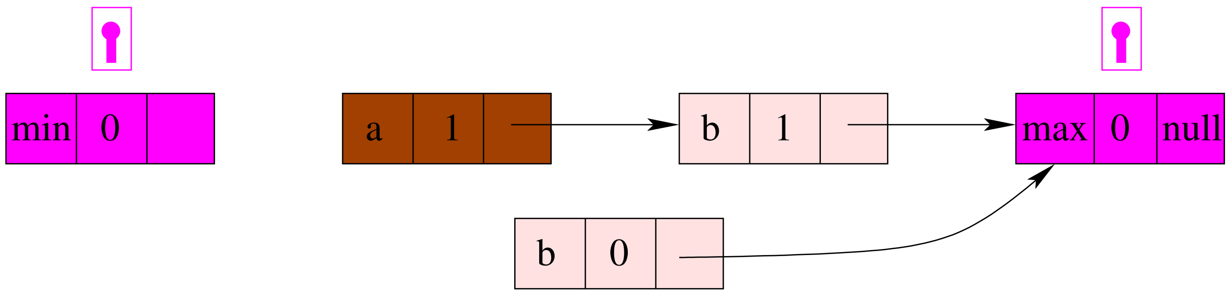
- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked

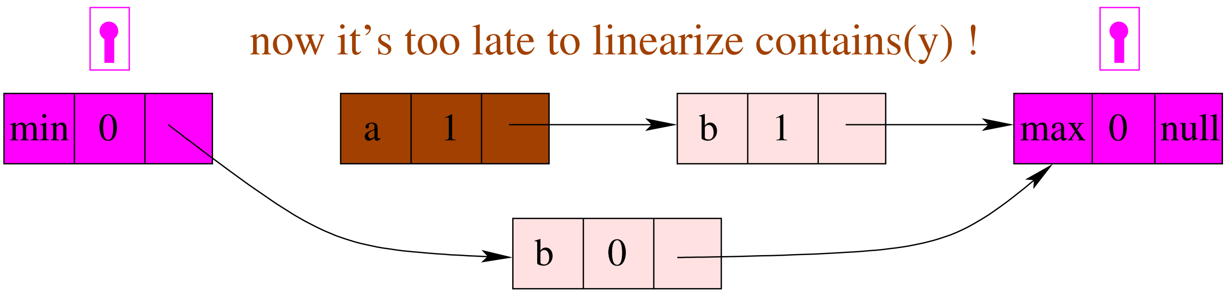


Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked

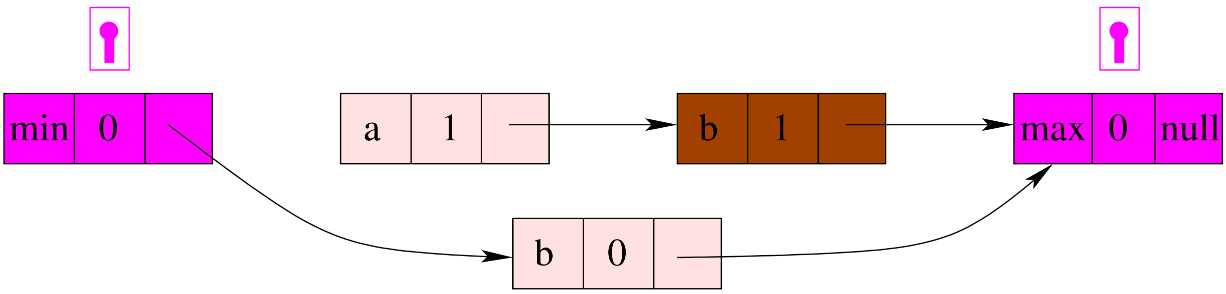
now it's too late to linearize contains(y) !



Lazy synchronization: linearizing unsuccessful contains

Example: Four methods are applied concurrently:

- `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: linearizing unsuccessful contains

An unsuccessful `contains(x)` can in each execution be linearized at a moment when `x` isn't in the set.

- If `x` isn't present in the set at the moment `contains(x)` is invoked, then we linearize `contains(x)` when it is invoked.
- Else, a `remove(x)` has its linearization point between the moments when `contains(x)` is invoked and returns.

  We linearize `contains(x)` right after the linearization point of such a `remove(x)`.

Lazy synchronization: progress property

The lazy synchronization algorithm isn't starvation-free,
since validation of `add` and `remove` by a thread may be unsuccessful an infinite number of times.

However, `contains` is wait-free.

Drawbacks:

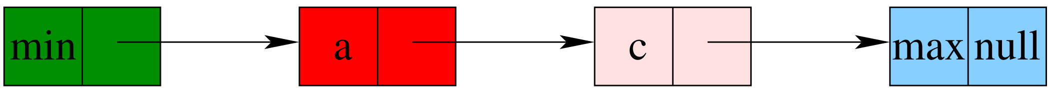
- contended `add` and `remove` calls retraverse the list
- `add` and `remove` are still blocking

Lock-free synchronization: simple idea is flawed

We will now look at a *lock-free* implementation of sets, using `compareAndSet` to redirect links.

This simple idea is flawed...

Example 1: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

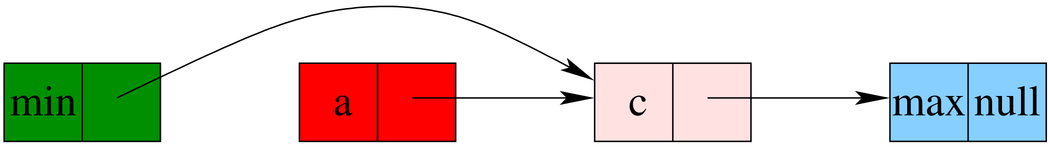


Lock-free synchronization: simple idea is flawed

We will now look at a *lock-free* implementation of sets, using `compareAndSet` to redirect links.

This simple idea is flawed...

Example 1: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

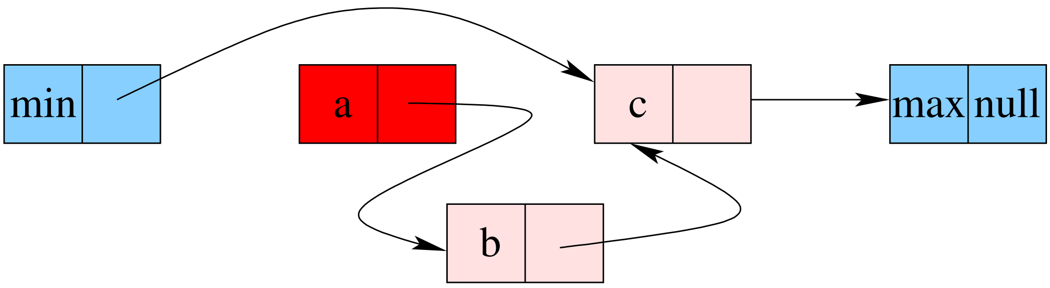


Lock-free synchronization: simple idea is flawed

We will now look at a *lock-free* implementation of sets, using `compareAndSet` to redirect links.

This simple idea is flawed...

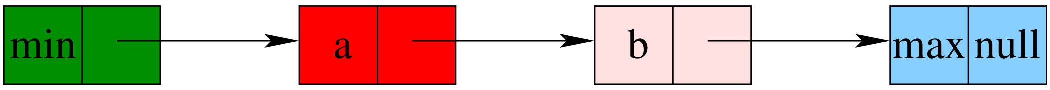
Example 1: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.



Node `b` isn't added !

Lock-free synchronization: simple idea is flawed

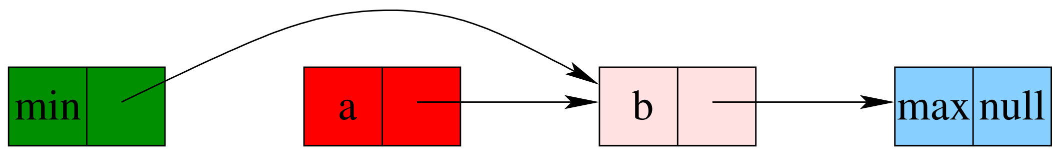
Example 2: `remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.



Question: How can this problem be resolved ?

Lock-free synchronization: simple idea is flawed

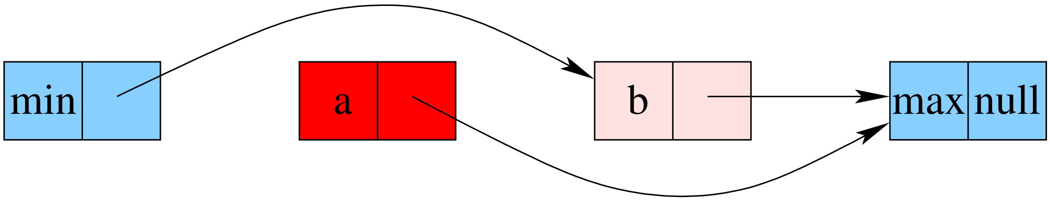
Example 2: `remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.



Question: How can this problem be resolved ?

Lock-free synchronization: simple idea is flawed

Example 2: `remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.



Node `b` isn't removed !

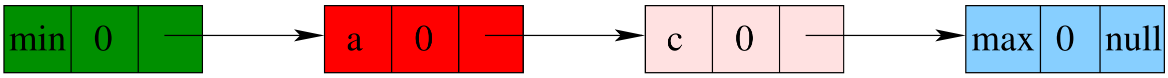
Question: How can this problem be resolved ?

Lock-free synchronization

Solution: Again nodes are supplied with a bit to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit (using the `AtomicMarkableReference` class).

Example: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

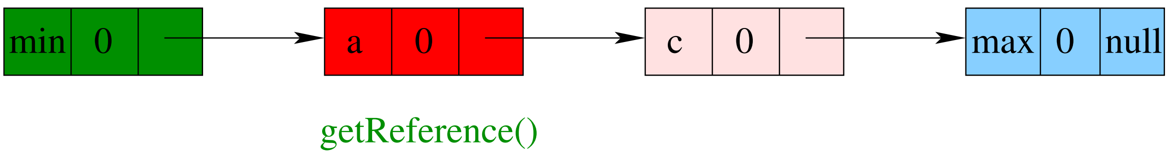


Lock-free synchronization

Solution: Again nodes are supplied with a bit to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit (using the `AtomicMarkableReference` class).

Example: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

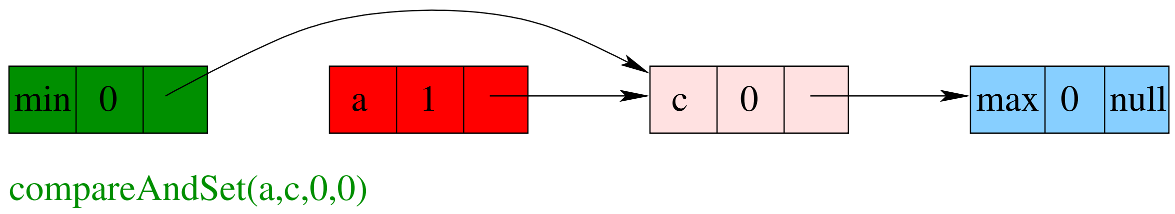


Lock-free synchronization

Solution: Again nodes are supplied with a bit to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit (using the `AtomicMarkableReference` class).

Example: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

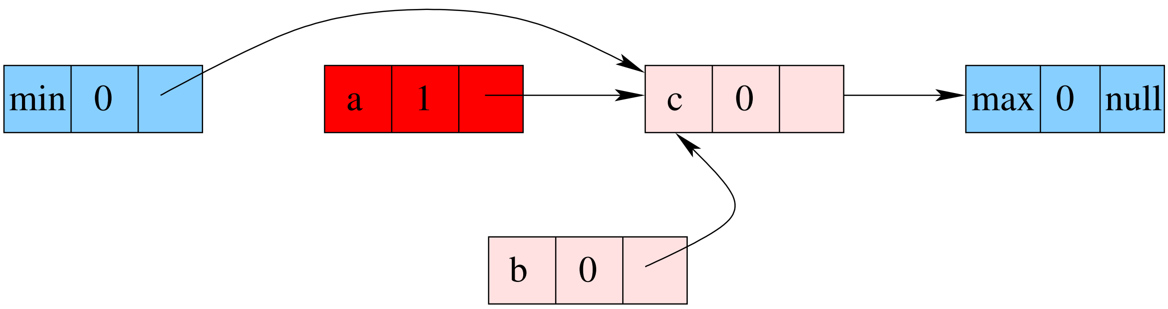


Lock-free synchronization

Solution: Again nodes are supplied with a bit to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit (using the `AtomicMarkableReference` class).

Example: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.

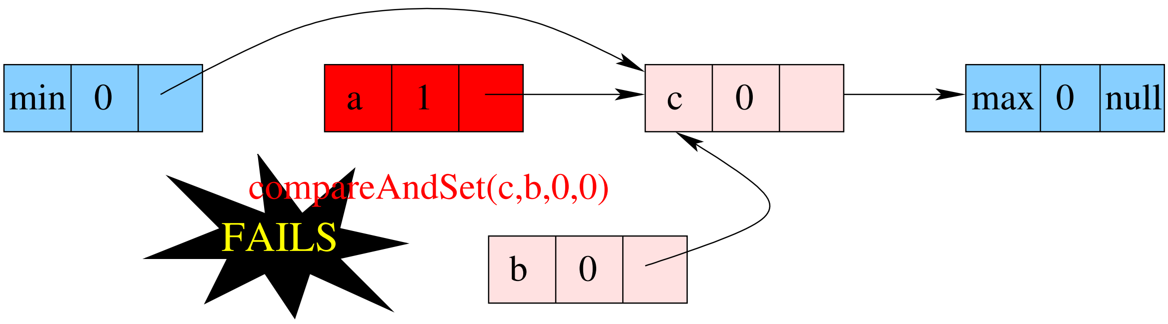


Lock-free synchronization

Solution: Again nodes are supplied with a bit to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit (using the `AtomicMarkableReference` class).

Example: `add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a` are being executed.



`add(x)` must start over!

AtomicMarkableReference class

`AtomicMarkableReference⟨T⟩` maintains:

- an object reference of type `T`, and
- a Boolean mark bit.

An internal object is created, representing a boxed (reference, bit) pair.

These two fields can be updated in one atomic step.

AtomicMarkableReference class: methods

```
boolean compareAndSet(T expectedRef, T newRef,
                      boolean expectedMark, boolean newMark)
```
Atomically sets reference and mark to `newRef` and `newMark`,
if reference and mark equal `expectedRef` and `expectedMark`.

```
boolean attemptMark(T expectedRef, boolean newMark)
```
Atomically sets mark to `newMark`, if reference equals `expectedRef`.

```
void set(T newRef, boolean newMark)
```
Atomically sets reference and mark to `newRef` and `newMark`.

`T get(boolean[] currentMark)` Atomically returns the value of reference and writes the value of mark at place 0 of the argument array.

`T getReference()` Returns the value of reference.

`boolean isMarked()` Returns the value of mark.

Lock-free synchronization: physical removal

When an `add` or `remove` call that traverses the list encounters a *marked* node `curr`, it attempts to physically remove this node by applying to its predecessor `pred`:

`compareAndSet(curr,succ,0,0)`

to redirect `pred` to the successor `succ` of `curr`.

If such an attempt *succeeds*, then the traversal continues.

If such an attempt *fails*, then the method call must start over, because it may be traversing an unreachable part of the list.

Lock-free synchronization: remove

`remove(x)` proceeds as follows:

- Search for a node c with a key $\geq$ hash(x) *(reference and mark of a node are read in one atomic step using* get() *)*.
- During this search, try to physically remove marked nodes, using compareAndSet.

  If at some point such a physical removal fails, start over.
- If the key of c is greater than hash(x), return *false*.

  If the key of c equals hash(x):
  - apply getReference() to obtain the successor s of c, and
  - apply compareAndSet(s,s,0,1) to try and mark c.
  - If this fails, start over.

    Else, apply compareAndSet(c,s,0,0) to try and redirect the predecessor p of c to s, and return *true*.

Lock-free synchronization: add

`add(x)` proceeds as follows:

- Search for a node `c` with a key $\geq$ `hash(x)`.
- During this search, try to physically remove marked nodes, using `compareAndSet`.

  If at some point such a physical removal fails, start over.
- If the key of `c` equals `hash(x)`, return *false*.

  If the key of `c` is greater than `hash(x)`:
  - create a node `n` with key `hash(x)`, value `x`, bit 0, and link to `c`, and
  - apply `compareAndSet(c,n,0,0)` to try and redirect the predecessor `p` of `c` to `n`.
  - If this fails, start over.

    Else, return *true*.

Lock-free synchronization: contains

`contains(x)` traverses the list *without cleaning up marked nodes*.

- Search for a node with the key `hash(x)`.
- If no such node is found, return *false*.
- If such a node is found, check whether it is marked.
- If so, return *false*, else return *true*.

Lock-free synchronization: linearization

The linearization points:

- successful `add`: When the predecessor is redirected to the added node.
- successful `remove`: When the mark is set.
- successful `contains`: When the (unmarked) node is found.
- unsuccessful `add(x)` and `remove(x)`: When the key is found that is equal to, respectively greater than, `hash(x)`.
- unsuccessful `contains(x)`: At a moment when `x` isn't in the set.

Lock-free synchronization: progress property

The lock-free algorithm is lock-free.

It is not wait-free, because list traversal of `add` and `remove` by a thread may be unsuccessful an infinite number of times.

`contains` is wait-free.

The lock-free algorithm for sets is in the Java Concurrency Package.