



Parallel linked lists

Lecture 10 of TDA384/DIT391
(Principles of Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg
SP3 2017/2018

Today's menu

The burden of locking

Linked set implementations

- Nodes, lists, and sets

- Sequential access

Parallel linked sets

- Coarse-grained locking

- Fine-grained locking

- Optimistic locking

- Lazy node removal

- Lock-free access

The burden of locking

Synchronization costs

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

Synchronization costs

A number of factors **challenge** designing correct and efficient **parallelizations**:

- sequential dependencies
- synchronization costs
- spawning costs
- error proneness and composability

In **this class**, we focus on reducing the **synchronization costs** associated with **locking**.

The trouble with locks

Standard techniques for concurrent programming are ultimately based on **locks**. Programming with locks has several **drawbacks**:

- performance overheads
- lock granularity is hard to choose:
 - not enough locking: race conditions
 - too much locking: not enough parallelism
- risk of deadlock and starvation
- lock-based implementations do not compose
- lock-based programs are hard to maintain and modify

Message-passing programming is higher-level, but it also inevitably incurs **synchronization costs** – of magnitude comparable to those associated with locks.

Breaking free of locks

Lock-free programming takes a fresh look at the problems of concurrency and tries to dispense with using locks altogether:

- lock-based programming is **pessimistic**: be prepared for the worst possible conditions:
 - if things can go **wrong**, they will
- lock-free programming is **optimistic**: do what you have to do without worrying about race conditions
 - if things go wrong, just **try again**

Lock-free programming

Lock-free programming relies on:

- using **stronger primitives** for atomic access
- building **optimistic** algorithms using those primitives

Compare-and-set operations are an example of stronger primitives:

```
public class AtomicInteger {  
    // atomically set to 'update' if current value is 'expect'  
    // otherwise do not change value and return false  
    boolean compareAndSet(int expect, int update)  
}
```

To update an **AtomicInteger** variable k:

```
do { // keep trying until no one changes k in between  
    int oldValue = k.get();  
    int newValue = compute(oldValue);  
} while (!k.compareAndSet(oldValue, newValue));
```


Lock-free vs. wait-free

Two **classes** of lock-free algorithms, collectively called **non-blocking**:

lock-free: guarantee system-wide progress: infinitely often, some process makes progress

wait-free: guarantee per-process progress: every process eventually makes progress

Wait-free is **stronger** than lock-free:

- Lock-free algorithms are free from deadlock
- Wait-free algorithms are free from deadlock and starvation

Thread-safe data structures

Programming correctly without using locks is challenging.

Instead of trying to develop general techniques, we focus on implementing **reusable data structures** that make minimal usage of locking. The effort involved in developing correct implementations pays off since very many applications can then use such **thread-safe data structure** implementations to synchronize **safely** and **implicitly** by accessing the structures through their APIs.

A data structure is **thread safe** if its operations are free from race conditions when executed by multi-threaded clients.

Our lock-free and wait-free algorithms are some of those used in the implementations of **thread safe** structures in **java.util.concurrent** (**non-blocking data structures** atomically accessible in parallel).

Linked set implementations

Parallel linked lists

In the rest of this class, we go through several implementations of **linked lists** that support **parallel access**; the implementations differ in how much locking they use to guarantee correctness and, correspondingly, in how much parallelism they allow.

We will use **pseudo-code** that is very close to regular Java syntax but occasionally takes some liberties to simplify the notation. On the course website you can download fully working implementations of some of the classes.

Linked set implementations

Nodes, lists, and sets

The interface of a set

We use linked lists to implement a **set** data structure with interface:

```
public interface Set<T>
{
    // add 'item' to set; return false if 'item' is already in the set
    boolean add(T item);

    // remove 'item' from set; return false if 'item' not in the set
    boolean remove(T item);

    // is 'item' in set?
    boolean has(T item);
}
```

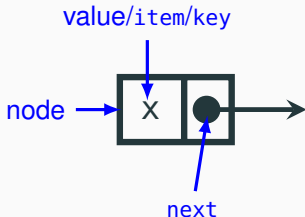
Nodes

The underlying implementations of sets use **singly-linked lists**, which are made of chains of nodes. Every **node**:

- stores an **item** – its **value**
- has a unique **key** – the value's **hash code**
- points to the **next** node in the chain

In the graphical representations of nodes, we do not distinguish between items and their keys – and represent both by characters:

```
interface Node<T>
{
    // value of node
    T item();
    // hash code of value
    int key();
    // next node in chain
    Node<T> next();
}
```



Lists as chains of nodes

A list with special **head** and **tail** nodes implements a set:

- the **elements** of the set are items in different nodes
- to facilitate searching, the nodes are maintained **sorted** in ascending **keys**
- to facilitate searching, the head has the smallest possible key, the tail has the largest possible key, and all elements have **finitely many** keys that are in between

For example, the **set** $\{b, e, a, f, g\}$ is implemented by:



Relaxing these assumptions is possible at the cost of complicating the implementations a bit.

Linked set implementations


Sequential access

Sequential set: basic linked implementation

We start with a standard linked-list-based implementation of sets, which only works for **sequential access**.

```
class SequentialSet<T> implements Set<T>
{
    // nodes at beginning and end
    protected Node<T> head, tail;

    // empty set
    public SequentialSet() {
        head = new SequentialNode<>(Integer.MIN_VALUE); // smallest key
        tail = new SequentialNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail);
    }
}
```

Empty set: head  tail

Nodes in a sequential set

A node's implementation uses **private attributes** with **getters** and **setters**; this is a bit tedious now (we could just let the set implementations access the attributes directly), but it will lead to nicer designs in the several variants of set implementations we'll describe.

```
class SequentialNode<T> implements Node<T> {  
    private T item;           // value stored in node  
    private int key;         // hash code of item  
    private Node<T> next;    // next node in chain  
    // getters  
    T item()                 { return item; }  
    int key()                 { return key; }  
    Node<T> next()           { return next; }  
    // setters  
    void setItem(T item)      { this.item = item; }  
    void setKey(int key)      { this.key = key; }  
    void setNext(Node<T> next) { this.next = next; }  
}
```

Finding a position inside a list

Since we maintain nodes in order of key, and every item has a unique key, we can **search** for the position of any given key by going through the list from head to tail.

Method `find` implements this frequently used operation of finding the **position** of a key inside a list. The position of key is a **pair** $(\text{pred}, \text{curr})$ of **adjacent nodes**, such that $\text{pred.key()} < \text{key} \leq \text{curr.key()}.$

For example, the position of c in the following list is:



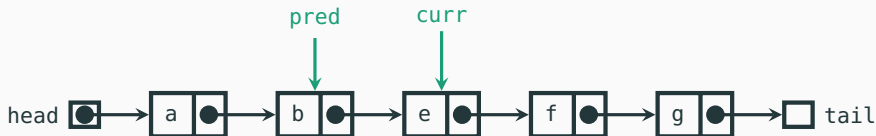
Thanks to the boundary keys chosen for head and tail, searching for **any value key** returns a **valid position** in the list.

Finding a position inside a list

Since we maintain nodes in order of key, and every item has a unique key, we can **search** for the position of any given key by going through the list from head to tail.

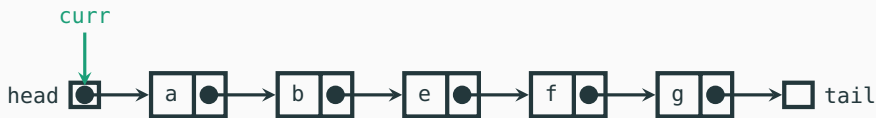
Method `find` implements this frequently used operation of finding the **position** of a key inside a list. The position of key is a **pair** $(\text{pred}, \text{curr})$ of **adjacent nodes**, such that $\text{pred.key()} < \text{key} \leq \text{curr.key()}.$

For example, the position of c in the following list is:



Thanks to the boundary keys chosen for `head` and `tail`, searching for **any value key** returns a **valid position** in the list.

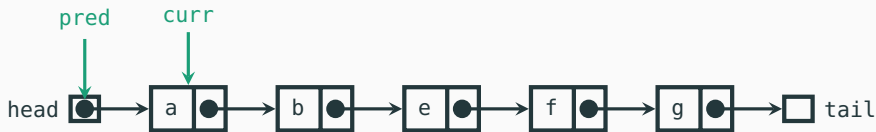
Finding a position inside a list



```
// first position from 'start' whose key is no smaller than 'key'  
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    curr = start;      // from start node  
    do {  
        pred = curr; curr = curr.next(); // move to next node  
    } while (curr.key() < key); // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Finding a position inside a list

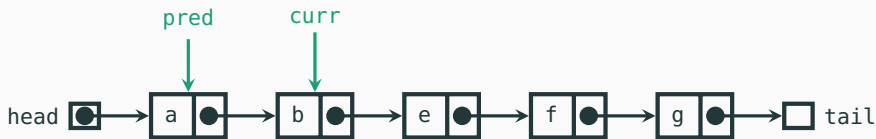


// first position from 'start' whose key is no smaller than 'key'

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    curr = start;      // from start node  
    do {  
        pred = curr; curr = curr.next(); // move to next node  
    } while (curr.key() < key); // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Finding a position inside a list

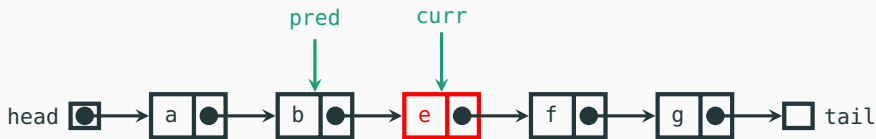


// first position from 'start' whose key is no smaller than 'key'

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    curr = start;      // from start node  
    do {  
        pred = curr; curr = curr.next(); // move to next node  
    } while (curr.key() < key); // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Finding a position inside a list



// first position from 'start' whose key is no smaller than 'key'

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    curr = start;      // from start node  
    do {  
        pred = curr; curr = curr.next(); // move to next node  
    } while (curr.key() < key); // until curr.key >= key  
    return (pred, curr); // return position  
}
```

← pseudo-code for: **new** Position<T>(pred, curr)

Sequential set: method has

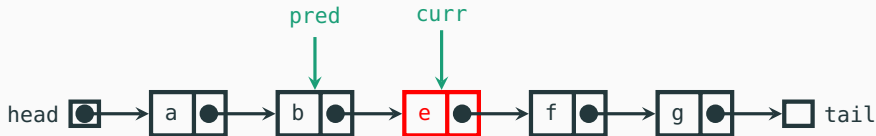
A set **has** `item` if and only if `item` is the first element whose key is greater than or equal to `item`'s (the **position** of `item`'s key).



```
// is 'item' in set?
public boolean has(T item) {
    int key = item.key();           // item's key
    // find position of key from head
    Node<T> pred, curr = find(head, key);
    // curr.key() >= key
    return curr.key() == key;      // item can only appear here!
}
```

Sequential set: method has

A set **has** *item* if and only if *item* is the first element whose key is greater than or equal to *item*'s (the **position** of *item*'s key).



```
// is 'item' in set?  
public boolean has(T item) {  
    int key = item.key();           // item's key  
    // find position of key from head  
    Node<T> pred, curr = find(head, key);  
    // curr.key() >= key  
    return curr.key() == key;      // item can only appear here!  
}
```

Sequential set: method add

A new item must be added **between** pred and curr, where (pred, curr) is item's **position** in the list.



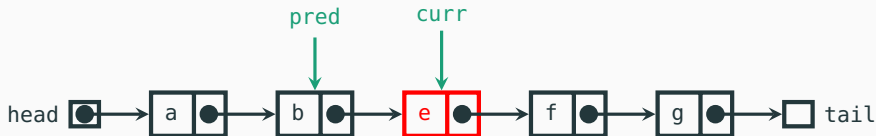
node:



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key()) return false; // item already in set  
    else // item not already in set: add node between pred and curr  
    { node.setNext(curr); pred.setNext(node); return true; }  
}
```

Sequential set: method add

A new item must be added **between** pred and curr, where (pred, curr) is item's **position** in the list.



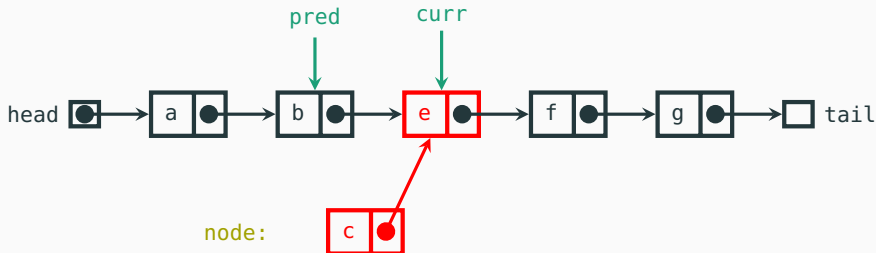
node:



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key()) return false; // item already in set  
    else // item not already in set: add node between pred and curr  
    { node.setNext(curr); pred.setNext(node); return true; }  
}
```

Sequential set: method add

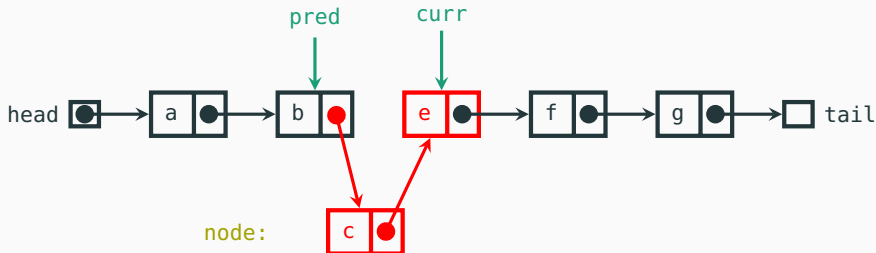
A new item must be added **between** pred and curr, where (pred, curr) is item's **position** in the list.



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key()) return false; // item already in set  
    else // item not already in set: add node between pred and curr  
    { node.setNext(curr); pred.setNext(node); return true; }  
}
```

Sequential set: method add

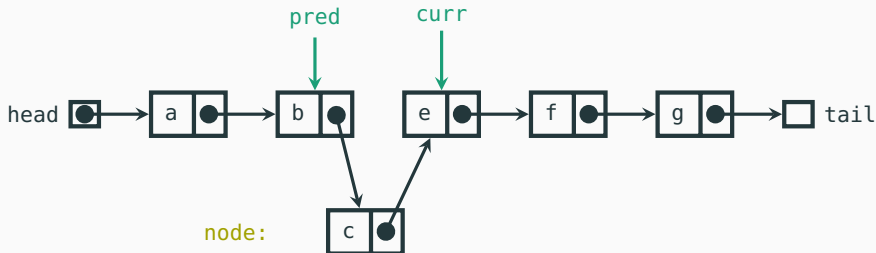
A new item must be added **between** pred and curr, where (pred, curr) is item's **position** in the list.



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key()) return false; // item already in set  
    else // item not already in set: add node between pred and curr  
    { node.setNext(curr); pred.setNext(node); return true; }  
}
```

Sequential set: method add

A new item must be added **between** pred and curr, where (pred, curr) is item's **position** in the list.



```
public boolean add(T item) {  
    Node<T> node = new Node<>(item);           // new node  
    Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key()  
    if (curr.key() == item.key()) return false; // item already in set  
    else // item not already in set: add node between pred and curr  
    { node.setNext(curr); pred.setNext(node); return true; }  
}
```


Sequential set: method remove

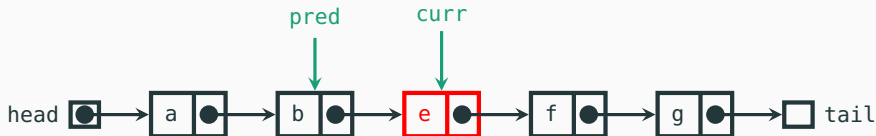
An element `item` is removed from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is `item`'s position in the list.



```
public boolean remove(T item) {  
    Node<T> pred, curr = find(head, item.key());  
    // curr.key() >= item.key()  
    if (curr.key() > item.key()) return false; // item not in set  
    else // item in set: remove node curr  
        { pred.setNext(curr.next()); return true; }  
}
```

Sequential set: method remove

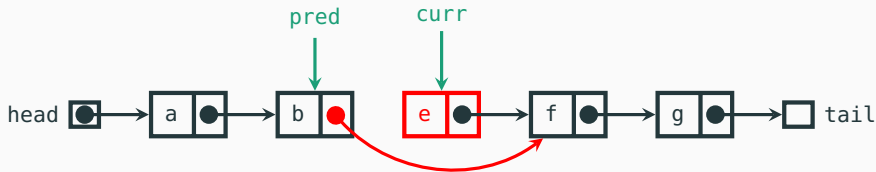
An element `item` is **removed** from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is `item`'s **position** in the list.



```
public boolean remove(T item) {  
    Node<T> pred, curr = find(head, item.key());  
    // curr.key() >= item.key()  
    if (curr.key() > item.key()) return false; // item not in set  
    else // item in set: remove node curr  
        { pred.setNext(curr.next()); return true; }  
}
```

Sequential set: method remove

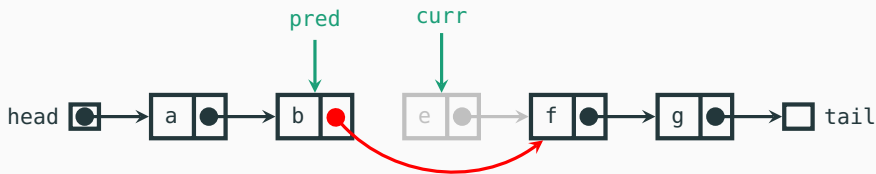
An element `item` is removed from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is `item`'s position in the list.



```
public boolean remove(T item) {  
    Node<T> pred, curr = find(head, item.key());  
    // curr.key() >= item.key()  
    if (curr.key() > item.key()) return false; // item not in set  
    else // item in set: remove node curr  
        { pred.setNext(curr.next()); return true; }  
}
```

Sequential set: method remove

An element `item` is **removed** from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is `item`'s **position** in the list.



```
public boolean remove(T item) {  
    Node<T> pred, curr = find(head, item.key());  
    // curr.key() >= item.key()  
    if (curr.key() > item.key()) return false; // item not in set  
    else // item in set: remove node curr  
        { pred.setNext(curr.next()); return true; }  
}
```

Sequential set: method remove

An element `item` is **removed** from a set by redirecting `pred.next` to skip over `curr`, where `(pred, curr)` is `item`'s **position** in the list.



```
public boolean remove(T item) {  
    Node<T> pred, curr = find(head, item.key());  
    // curr.key() >= item.key()  
    if (curr.key() > item.key()) return false; // item not in set  
    else // item in set: remove node curr  
        { pred.setNext(curr.next()); return true; }  
}
```

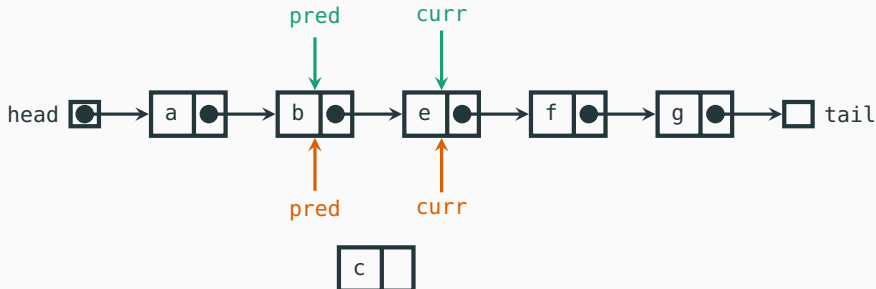
Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

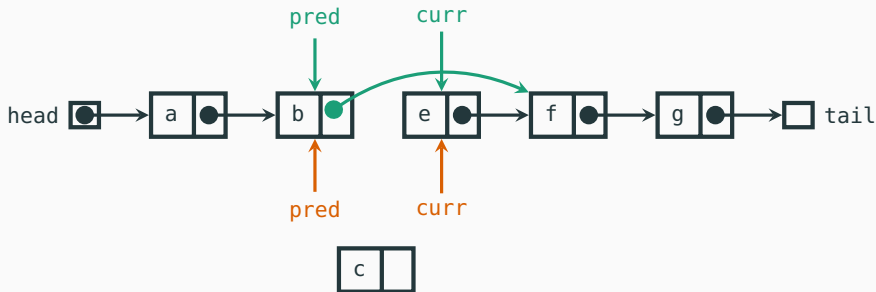
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `remove` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

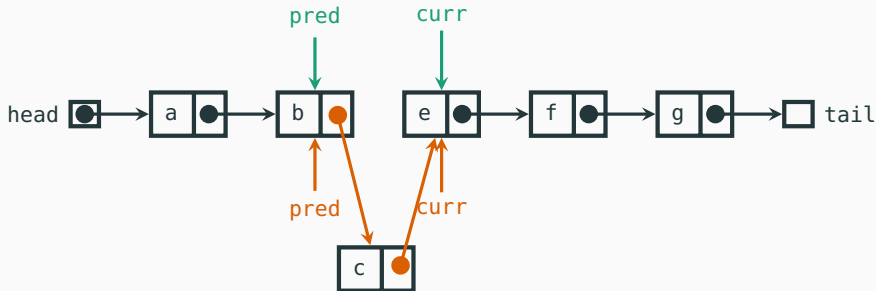
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `remove` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

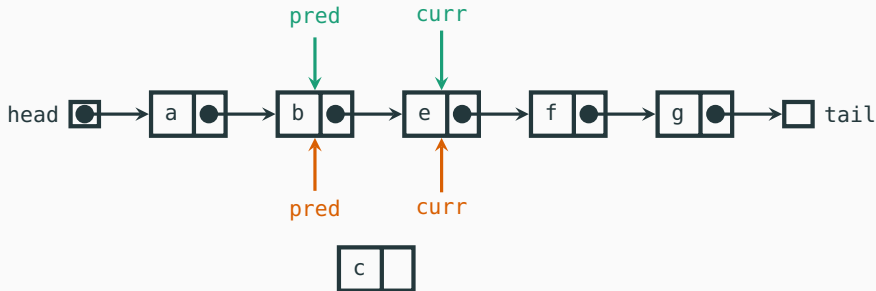
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `remove` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

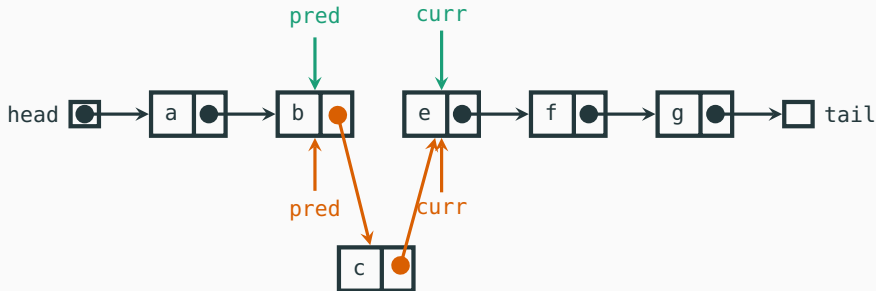
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `add` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

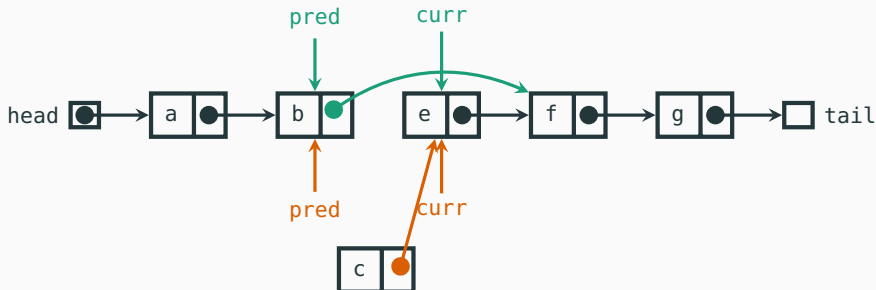
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `add` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

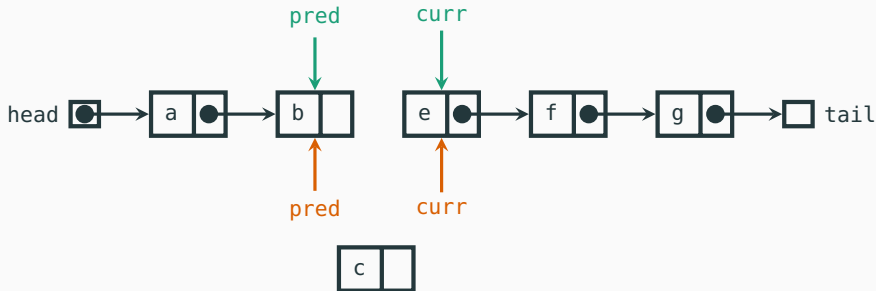
For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `add` is reverted:



Sequential set does not work under concurrency

If **multiple threads** are active on the same instance of `SequentialSet`, they can easily **interfere** with each other's operations – and possibly leave the set in an inconsistent state.

For example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `add(c)`: in some interleavings, `add` is reverted:



If `find` goes through the list while another thread is modifying it, even more subtle errors may occur.

Parallel linked sets

Parallel linked sets

Coarse-grained locking

Concurrent set with coarse-grained locking

A straightforward way to make `SequentialSet` work correctly under concurrency is using a **lock** to ensure that **at most one thread** at a time is operating on the structure.

```
class CoarseSet<T> extends SequentialSet<T>
{
    // lock controlling access to the whole set
    private Lock lock = new ReentrantLock();

    // overriding of add, remove, and has
```

Every method `add`, `remove`, and `has` simply works as follows:

1. acquires the **lock** on the set
2. performs the **operation** as in `SequentialSet`
3. releases the **lock** on the set

Coarse-locking set: method add



node:



```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add

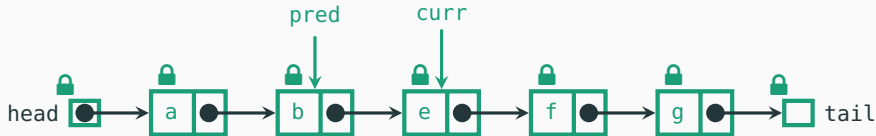


node:



```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add

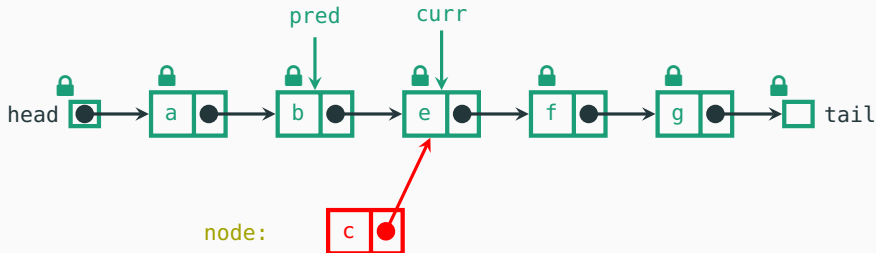


node:



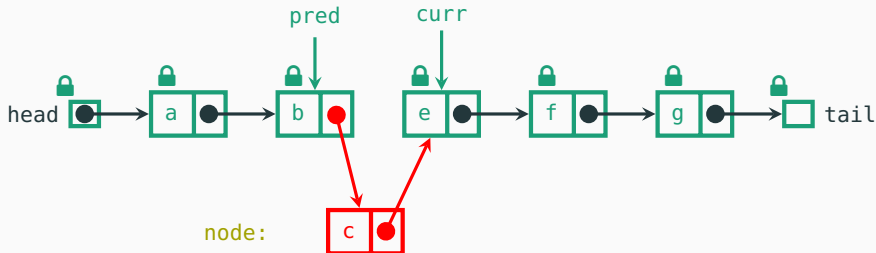
```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add



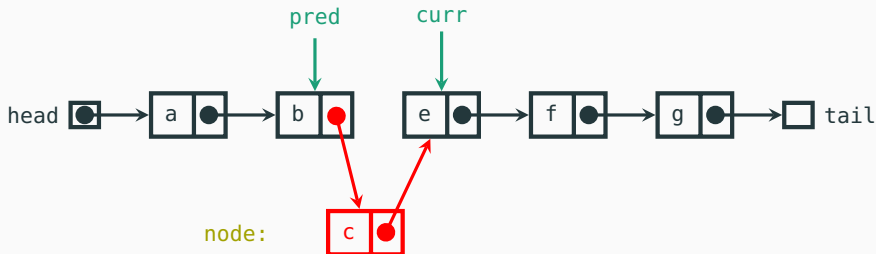
```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add



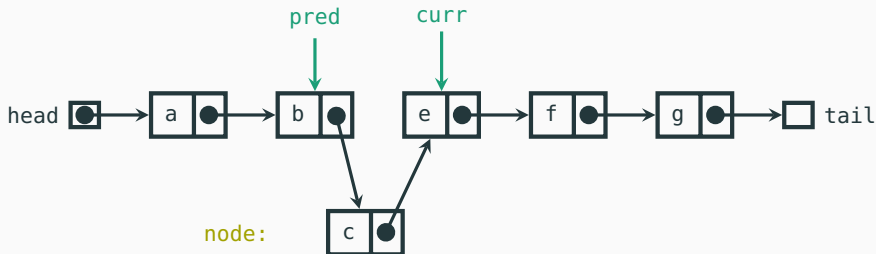
```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add



```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method add



```
public boolean add(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.add(item); // execute 'add' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method remove



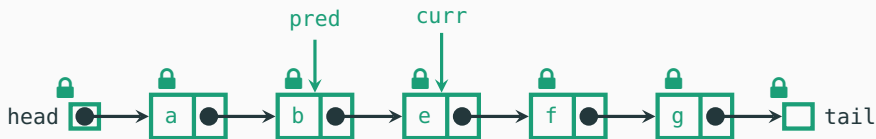
```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```


Coarse-locking set: method remove



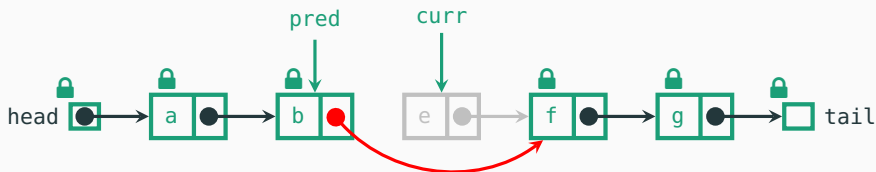
```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method remove



```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method remove



```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method remove



```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method remove



```
public boolean remove(T item) {  
    lock.lock(); // lock whole set  
    try {  
        return super.remove(item); // execute 'remove' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: method has



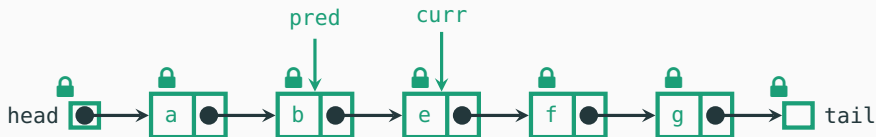
```
public boolean has(T item) {  
    lock.lock();           //lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock();       // done: release lock  
    }  
}
```

Coarse-locking set: method has



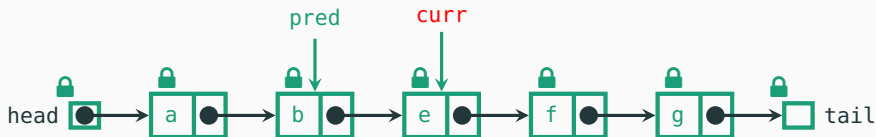
```
public boolean has(T item) {  
    lock.lock();           //lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock();       // done: release lock  
    }  
}
```

Coarse-locking set: method has



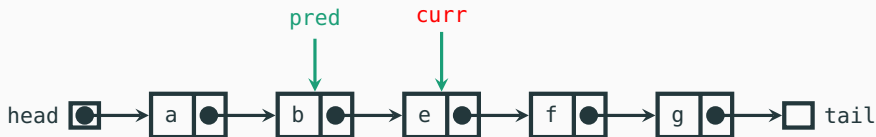
```
public boolean has(T item) {  
    lock.lock(); //lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```


Coarse-locking set: method has



```
public boolean has(T item) {  
    lock.lock();           //lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock();       // done: release lock  
    }  
}
```

Coarse-locking set: method has



```
public boolean has(T item) {  
    lock.lock(); //lock whole set  
    try {  
        return super.has(item); // execute 'has' while locking  
    } finally {  
        lock.unlock(); // done: release lock  
    }  
}
```

Coarse-locking set: pros and cons

Pros:

- obviously correct – it avoids race conditions and deadlocks
- if the lock is fair, so is access to the set
- if contention is low (not many threads accessing the set concurrently), CoarseSet is quite efficient

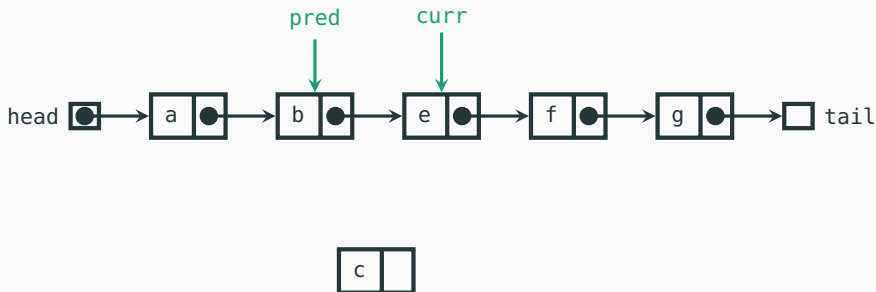
Cons:

- access to the set is essentially sequential – missing opportunities for parallelization
- if contention is high (many threads accessing the set concurrently), CoarseSet is quite slow

Locking after finding?

Can we reduce the size of the critical sections by executing **find without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

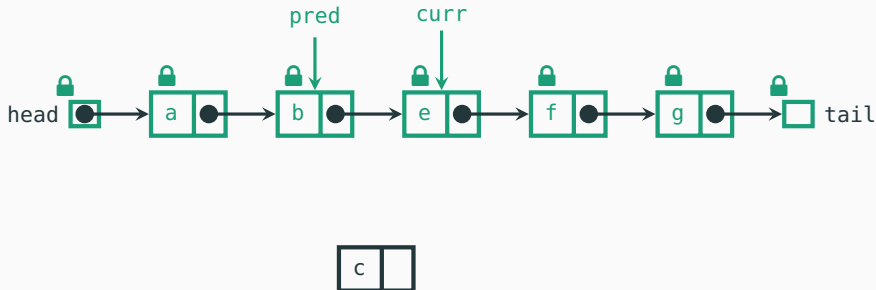
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing **find without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

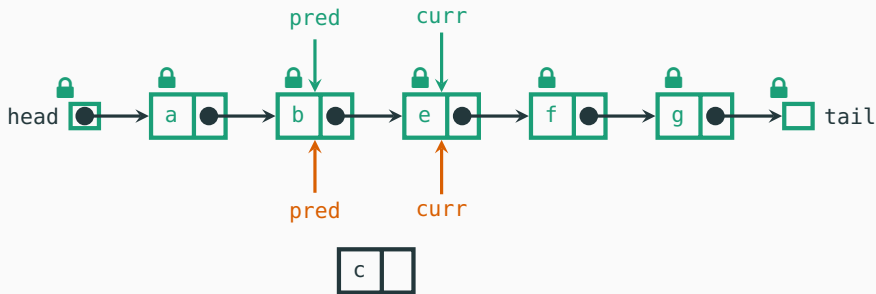
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing `find` **without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

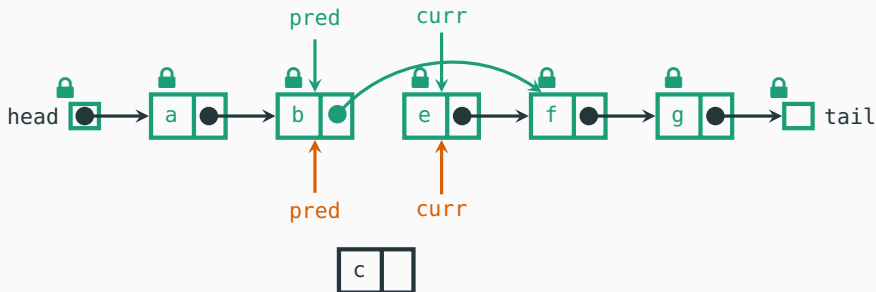
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing `find` **without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

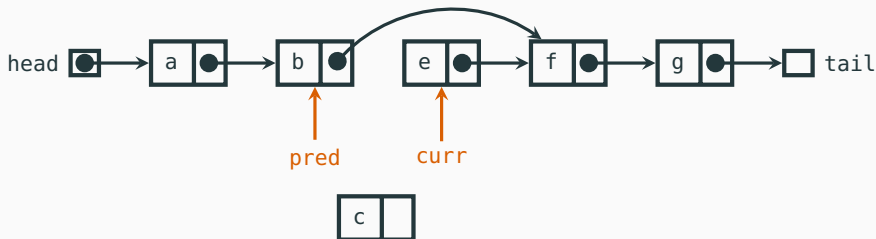
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing **find without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

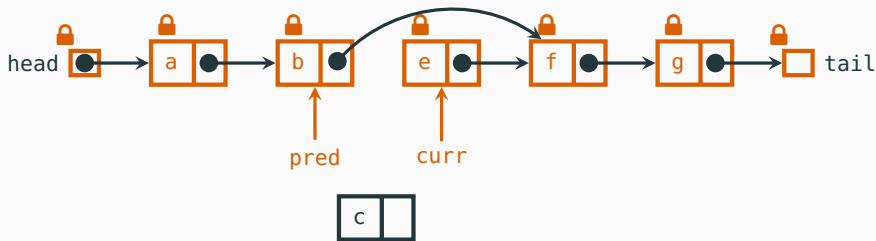
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing `find` **without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

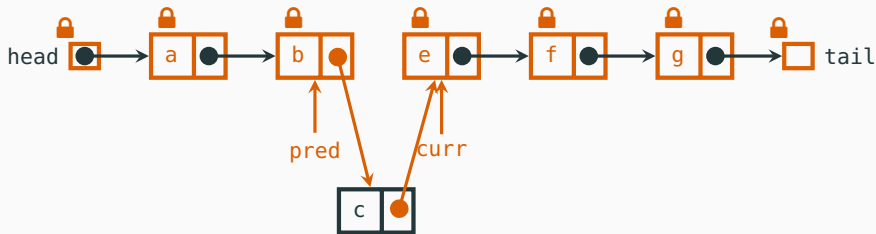
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing `find` **without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

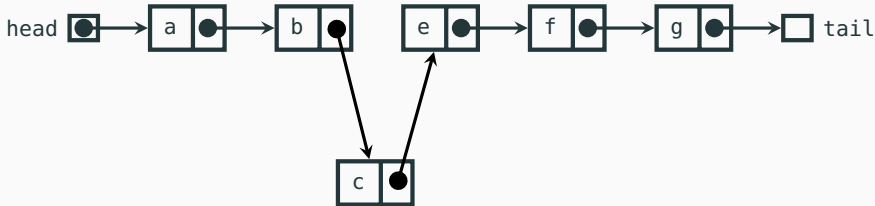
For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Locking after finding?

Can we reduce the size of the critical sections by executing `find` **without locking**, and then acquiring the lock only before modifying the list? **No**, because the list may be modified between when a thread performs `find` and when it acquires the lock.

For example, suppose **thread t** runs `remove(e)` while **thread u** runs `add(c)`, and t acquires the lock first:



Parallel linked sets

Fine-grained locking

Concurrent set with fine-grained locking

Rather than locking the whole linked list at once, we add a **lock** to each node. Then, threads only **lock the individual nodes** on which they are operating.

```
public class FineSet<T> extends SequentialSet<T>
{
    // empty set
    public FineSet() {
        head = new LockableNode<>(Integer.MIN_VALUE); // smallest key
        tail = new LockableNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail);
    }

    // overriding of find, add, remove, and has
```

Nodes in a fine-locking set

Each node includes a **lock object**, and `lock` and `unlock` methods that access the lock.

```
class LockableNode<T> extends SequentialNode<T>
{
    private Lock lock = new ReentrantLock();

    void lock()    { lock.lock(); }    // lock node
    void unlock() { lock.unlock(); } // unlock node
}
```

How many nodes do we have to lock?

We have seen (in `CoarseSet`) that we have to lock as soon as we start executing `find`. Thus, we start locking the head node and **pass the lock along** the chain of nodes.

How many nodes do we have to hold locked at once? Even though `pred`'s node is the only node that is actually modified, **only locking `pred` is not enough**.

For example, if **thread `t`** runs `remove(e)` while **thread `u`** runs `remove(b)`, it may happen that only `b`'s removal takes place:

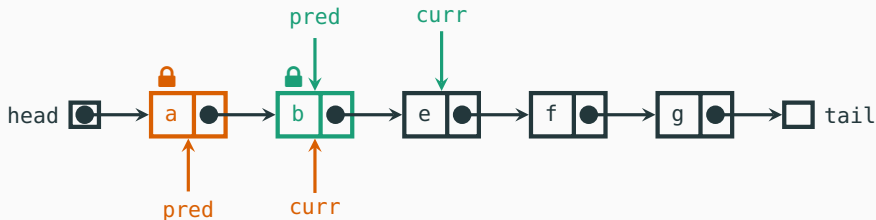


How many nodes do we have to lock?

We have seen (in `CoarseSet`) that we have to lock as soon as we start executing `find`. Thus, we start locking the head node and **pass the lock along** the chain of nodes.

How many nodes do we have to hold locked at once? Even though `pred`'s node is the only node that is actually modified, **only locking `pred` is not enough**.

For example, if **thread `t`** runs `remove(e)` while **thread `u`** runs `remove(b)`, it may happen that only `b`'s removal takes place:

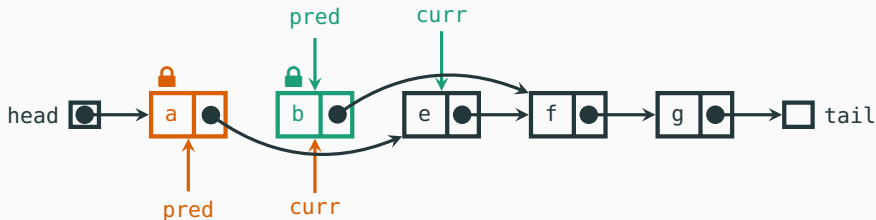


How many nodes do we have to lock?

We have seen (in `CoarseSet`) that we have to lock as soon as we start executing `find`. Thus, we start locking the head node and **pass the lock along** the chain of nodes.

How many nodes do we have to hold locked at once? Even though `pred`'s node is the only node that is actually modified, **only locking `pred` is not enough**.

For example, if **thread `t`** runs `remove(e)` while **thread `u`** runs `remove(b)`, it may happen that only `b`'s removal takes place:



Thus, we **lock both `pred` and `curr`** at once.

Fine-locking set: method find

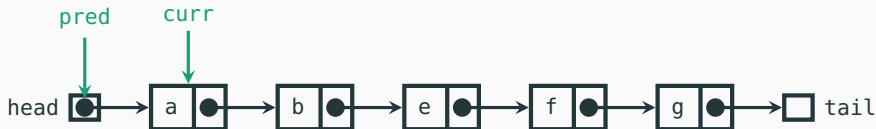


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

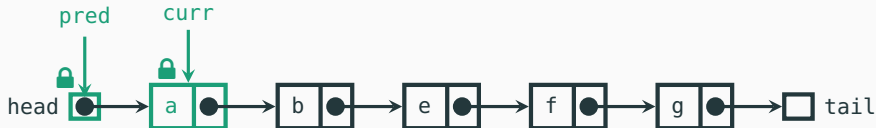


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

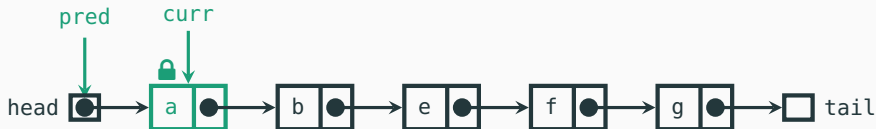


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: `new Position<T>(pred, curr)`

Fine-locking set: method find

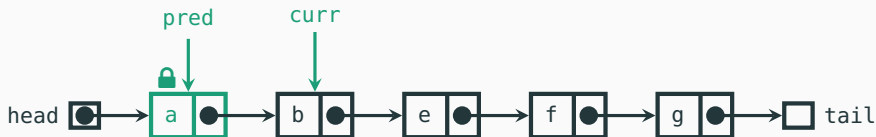


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

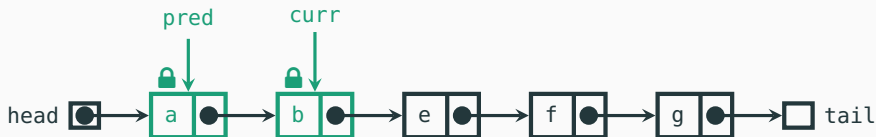


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

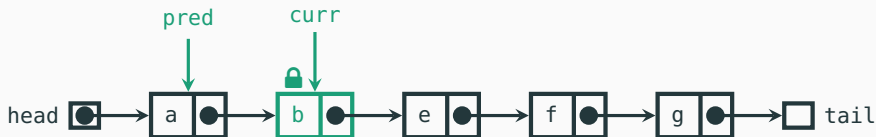


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: `new Position<T>(pred, curr)`

Fine-locking set: method find

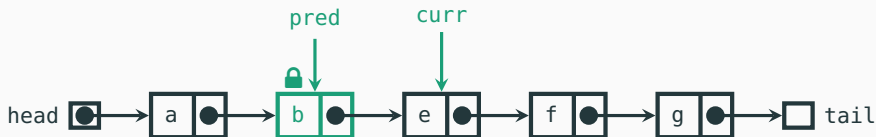


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

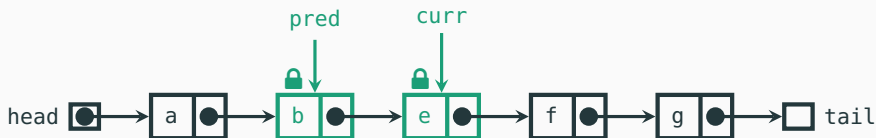


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

← pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find

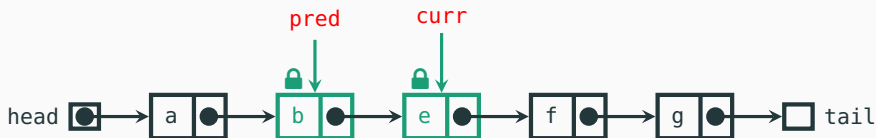


// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

pseudo-code for: **new** Position<T>(pred, curr)

Fine-locking set: method find



// find while locking pred and curr, return locked position

```
protected Node<T>, Node<T> find(Node<T> start, int key) {  
    Node<T> pred, curr; // predecessor and current node in iteration  
    pred = start; curr = start.next(); // from start node  
    pred.lock(); curr.lock(); // lock pred and curr nodes  
    while (curr.key < key) {  
        pred.unlock(); // unlock pred node  
        pred = curr; curr = curr.next(); // move to next node  
        curr.lock(); // lock next node  
    } // until curr.key >= key  
    return (pred, curr); // return position  
}
```

← pseudo-code for: **new** Position<T>(pred, curr)

Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

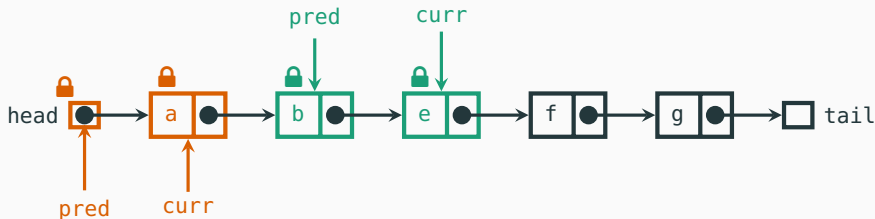
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

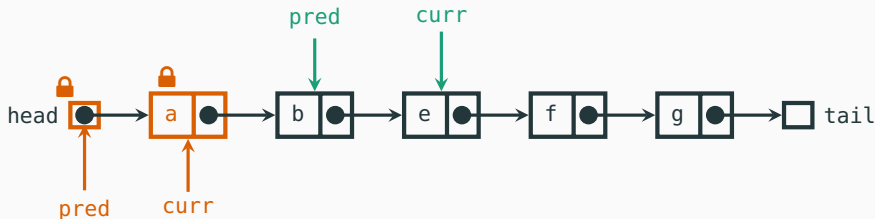
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

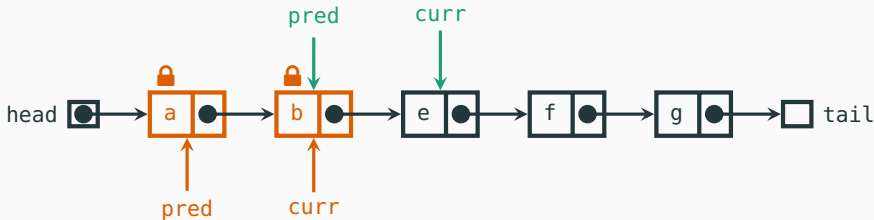
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

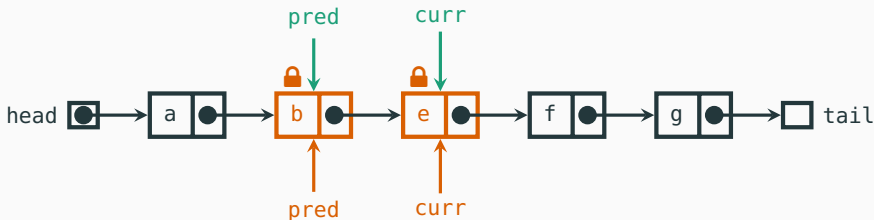
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

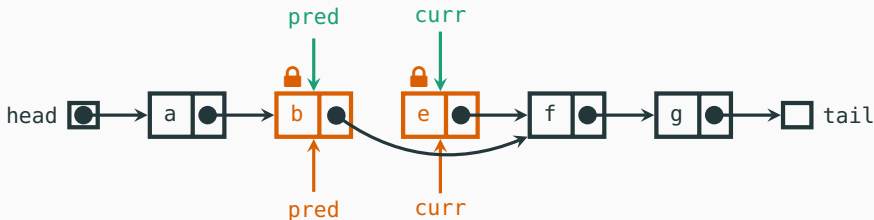
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

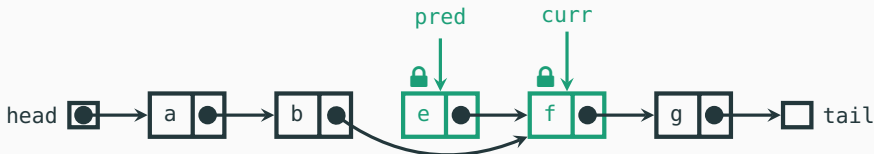
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

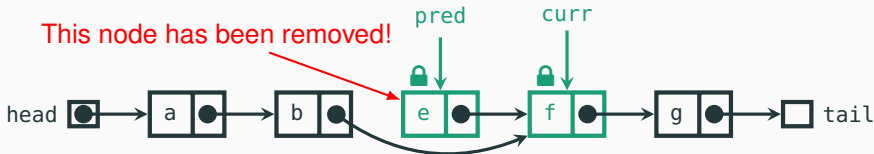
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

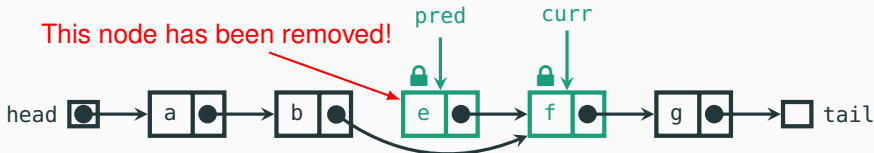
- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



Hand-over-hand locking

The lock acquisition protocol used by `find` in `FineSet` is called **hand-over-hand** locking or **lock coupling**.

- Always keeping **at least one node locked** prevents interference between threads; otherwise this may happen:



- Locking two nodes at once is sufficient to **prevent** problems with **conflicting operations**: threads proceed along the linked list in order, without one thread “overtaking” another thread that is further out
- The protocol ensures that locks are acquired by all threads in the same order, thus **avoiding deadlocks**

Fine-locking set: method add

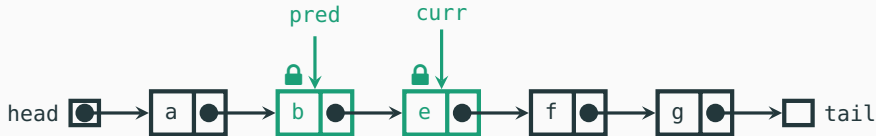


node:



```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method add

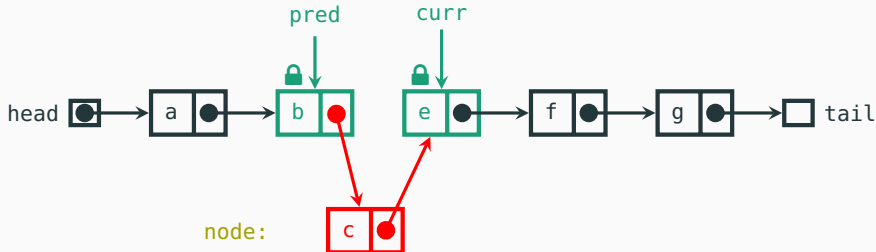


node:



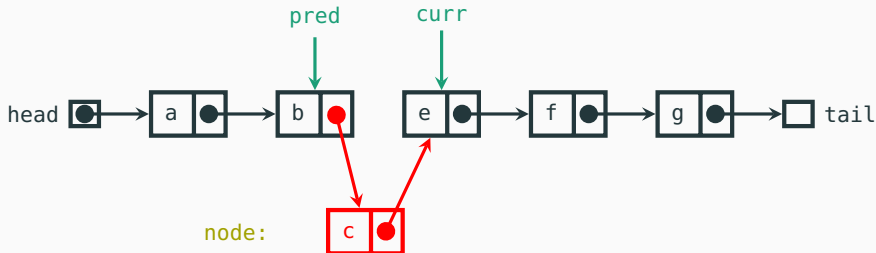
```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method add



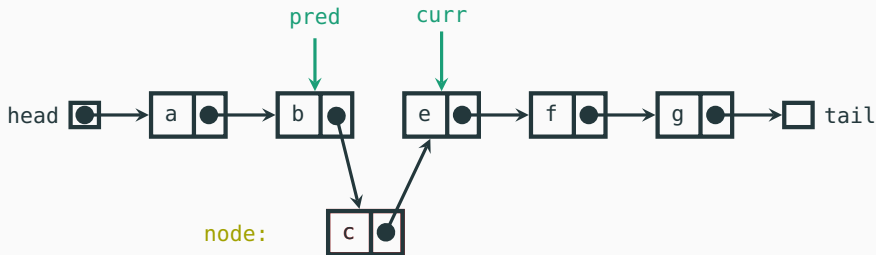
```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method add



```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```


Fine-locking set: method add



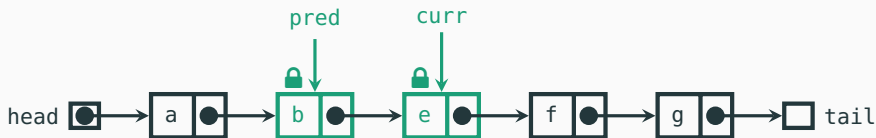
```
public boolean add(T item) {  
    Node<T> node = new LockableNode<>(item); // new node  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // add node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method `remove`



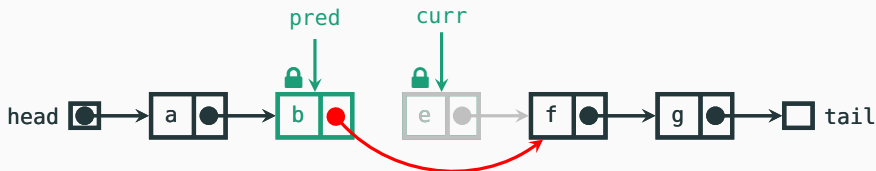
```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key >= item.key  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method `remove`



```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key >= item.key  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method `remove`



```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key >= item.key  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method `remove`



```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key >= item.key  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method `remove`



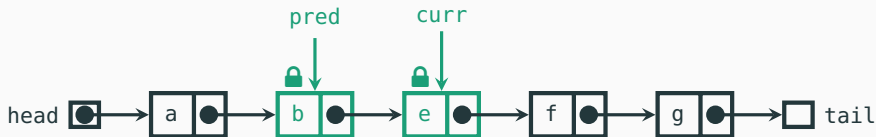
```
public boolean remove(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key >= item.key  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // remove node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method has



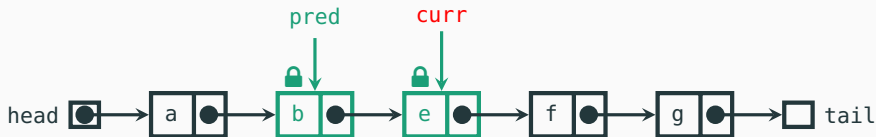
```
public boolean has(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // check node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method has



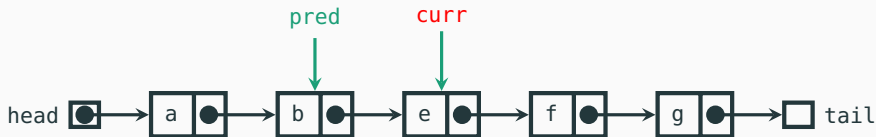
```
public boolean has(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // check node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```


Fine-locking set: method has



```
public boolean has(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // check node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: method has



```
public boolean has(T item) {  
    try { // find with hand-over-hand locking  
        // the first position such that curr.key() >= item.key()  
        Node<T> pred, curr = find(head, item.key()); // locking  
        ... // check node as in SequentialSet, while locking  
    } finally { pred.unlock(); curr.unlock(); } // done: unlocking  
}
```

Fine-locking set: pros and cons

Pros:

- if locks are fair, so is access to the set, because threads proceed along the list one after the other without changing order
- threads operating on disjoint portions of the list may be able to operate in parallel

Cons:

- it is still possible that one thread prevents another thread from operating in parallel on a disjoint portion of the list – for example, if one thread wants to access the end of the list but another thread blocks it while locking the beginning of the list
- the hand-over-hand locking protocol may be quite slow, as it involves a significant number of lock operations

Parallel linked sets

Optimistic locking

Concurrent set with optimistic locking

Let us revisit the idea of performing **find without locking**. We have seen that problems may occur if the list is modified between when a thread finds a position and when it acquires locks on that position. Thus, we **validate** a position **after finding it** and while locking it, to verify that no interference took place.

```
public class OptimisticSet<T> extends SequentialSet<T>
{
    public FineSet()
    { head = new ReadWriteNode<>(Integer.MIN_VALUE); // smallest key
      tail = new ReadWriteNode<>(Integer.MAX_VALUE); // largest key
      head.setNext(tail); }

    // is (pred, curr) a valid position?
    protected boolean valid(Node<T> pred, Node<T> curr) // ...

    // overriding of find, add, remove, and has
```

Nodes in an optimistic-locking set

Since we need to be able to **follow** the chain of **next** references **without locking**, attribute `next` must be declared **volatile** in Java – so that modifications to it (which occur while locking) are propagated to all threads (even if they are not locking a node). Other than for this detail, a `ReadWriteNode` is the same as a `LockableNode`.

With a little abuse of notation, we can pretend that `ReadWriteNode` inherits from `LockableNode` and overrides its `next` attribute. Overriding of attributes is however not possible in Java (shadowing takes place instead); the actual implementation that we make available does not reuse `LockableNode`'s code through inheritance.

```
class ReadWriteNode<T> extends LockableNode<T>
{
    private volatile Node<T> next;    // next node in chain
}
```

Delayed locking as optimistic locking

In `OptimisticSet`, operations work as follows:

1. **find** the item's position inside the list without locking – as in `SequentialSet`
2. **lock** the position's nodes `pred` and `curr`
3. **validate** the position while locking:
 - 3.1 if the position is valid, **perform the operation** while locking, then release locks
 - 3.2 if the position is invalid, release locks and **repeat the operation** from scratch

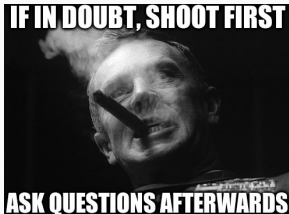
This approach is **optimistic** because it works well when validation is often successful (so we don't have to repeat operations).

Delayed locking as optimistic locking

In `OptimisticSet`, operations work as follows:

1. **find** the item's position inside the list without locking – as in `SequentialSet`
2. **lock** the position's nodes `pred` and `curr`
3. **validate** the position while locking:
 - 3.1 if the position is valid, **perform the operation** while locking, then release locks
 - 3.2 if the position is invalid, release locks and **repeat the operation** from scratch

This approach is **optimistic** because it works well when validation is often successful (so we don't have to repeat operations).



Optimistic set: method add

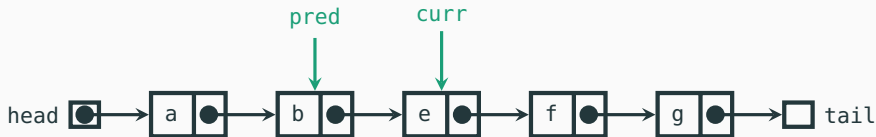


node:



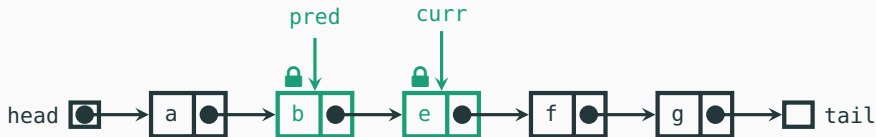
```
public boolean add(T item) {
    Node<T> node = new ReadWriteNode<>(item);           // new node
    do { Node<T> pred, curr = find(head, item.key()); // no locking
        pred.lock(); curr.lock();                       // now lock position
        try { // if position still valid, while locking:
            if (valid(pred, curr)) { ... } // physically add node
        } finally { pred.unlock(); curr.unlock(); } // done: unlock
    } while (true);                                     // if not valid: try again!
}
```

Optimistic set: method add



```
public boolean add(T item) {  
    Node<T> node = new ReadWriteNode<>(item);           // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock();                       // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true);                                     // if not valid: try again!  
}
```

Optimistic set: method add

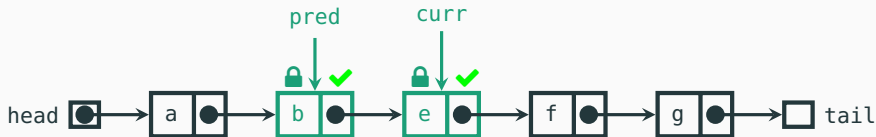


node:



```
public boolean add(T item) {  
    Node<T> node = new ReadWriteNode<>(item);           // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock();                       // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true);                                     // if not valid: try again!  
}
```

Optimistic set: method add

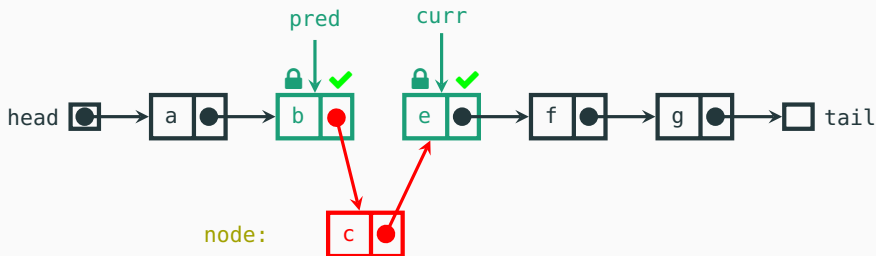


node:

c

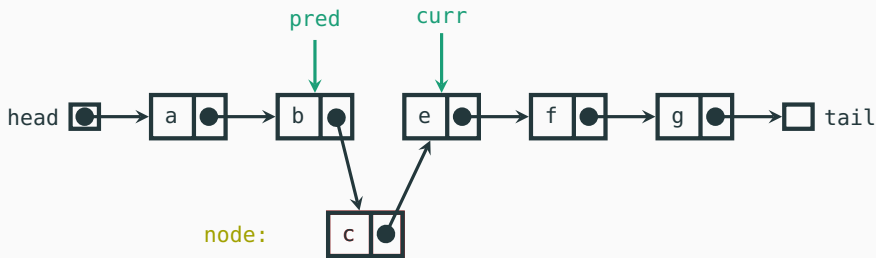
```
public boolean add(T item) {  
    Node<T> node = new ReadWriteNode<>(item);           // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock();                       // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true);                                     // if not valid: try again!  
}
```

Optimistic set: method add



```
public boolean add(T item) {  
    Node<T> node = new ReadWriteNode<>(item);           // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock();                       // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true);                                     // if not valid: try again!  
}
```

Optimistic set: method add



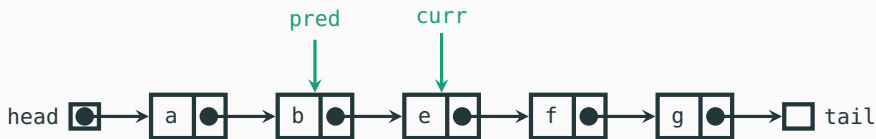
```
public boolean add(T item) {  
    Node<T> node = new ReadWriteNode<>(item);           // new node  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock();                       // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically add node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true);                                     // if not valid: try again!  
}
```

Optimistic set: method remove



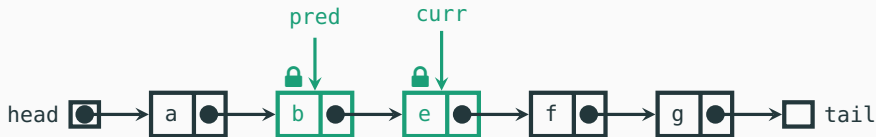
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method remove



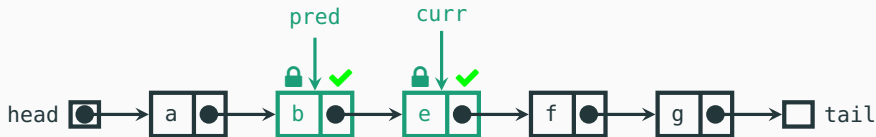
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```


Optimistic set: method remove



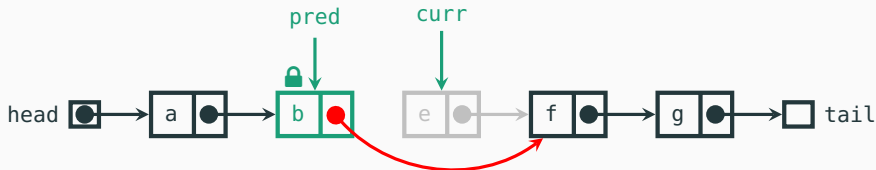
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method remove



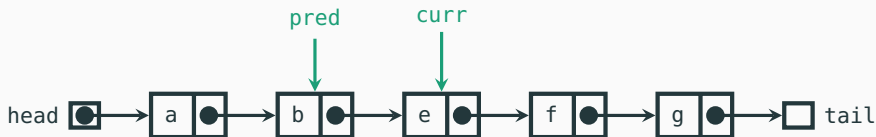
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, while locking:  
            if (valid(pred, curr)) { ... } // physically remove node  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method has



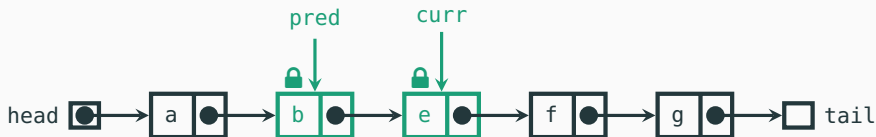
```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method has



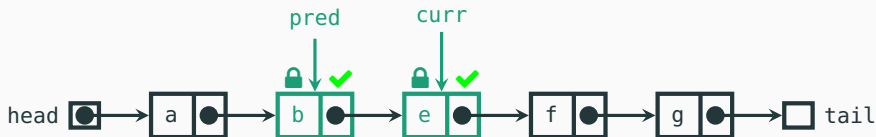
```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method has



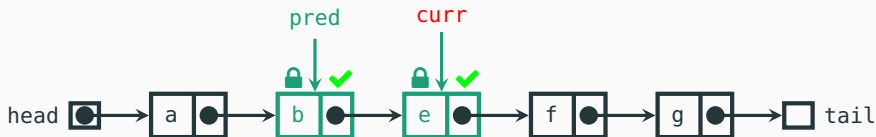
```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: method has



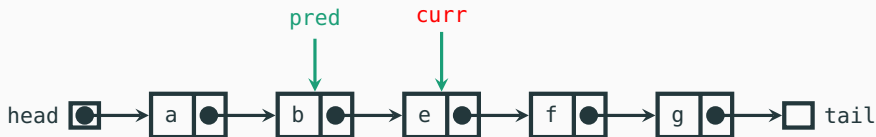
```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```


Optimistic set: method has



```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

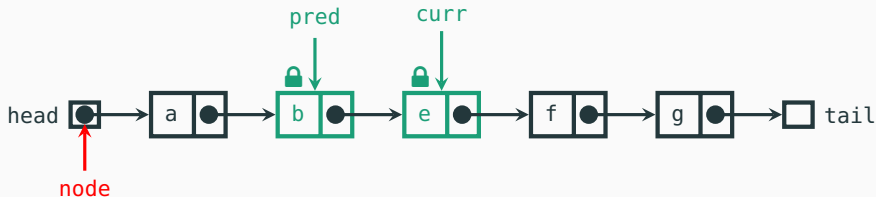
Optimistic set: method has



```
public boolean has(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // no locking  
        pred.lock(); curr.lock(); // now lock position  
        try { // if position still valid, check key while locking  
            if (valid(pred, curr)) return curr.key() == item.key();  
        } finally { pred.unlock(); curr.unlock(); } // done: unlock  
    } while (true); // if not valid: try again!  
}
```

Optimistic set: validating a position

Validation goes through the nodes until it reaches the given position.



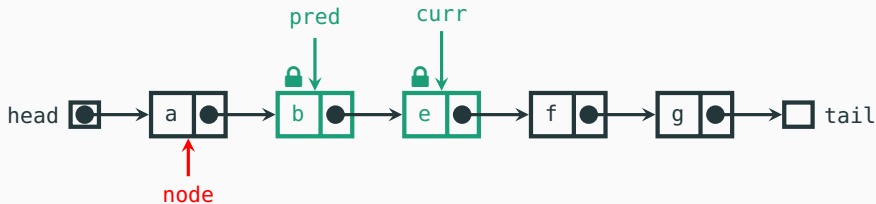
```
// is pred reachable from head, and does it point to curr?
```

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    Node<T> node = head; // start from head  
    while (node.key() <= pred.key()) { // does pred point to curr?  
        if (node == pred) return pred.next() == curr;  
        node = node.next(); // continue to the next node  
    } // until node.pred > pred.key  
    return false; // pred could not be reached  
}
```

// or does not point to curr

Optimistic set: validating a position

Validation goes through the nodes until it reaches the given position.



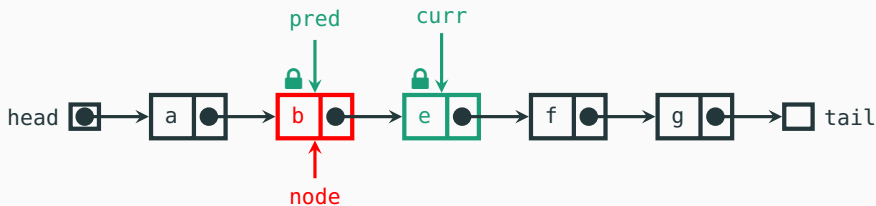
```
// is pred reachable from head, and does it point to curr?
```

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    Node<T> node = head; // start from head  
    while (node.key() <= pred.key()) { // does pred point to curr?  
        if (node == pred) return pred.next() == curr;  
        node = node.next(); // continue to the next node  
    } // until node.pred > pred.key  
    return false; // pred could not be reached  
}
```

// or does not point to curr

Optimistic set: validating a position

Validation goes through the nodes until it reaches the given position.



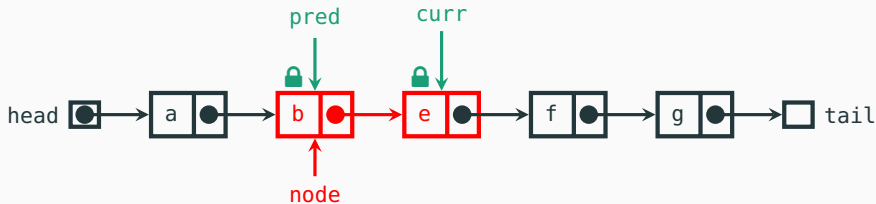
```
// is pred reachable from head, and does it point to curr?
```

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    Node<T> node = head; // start from head  
    while (node.key() <= pred.key()) { // does pred point to curr?  
        if (node == pred) return pred.next() == curr;  
        node = node.next(); // continue to the next node  
    } // until node.pred > pred.key  
    return false; // pred could not be reached  
}
```

// or does not point to curr

Optimistic set: validating a position

Validation goes through the nodes until it reaches the given position.



```
// is pred reachable from head, and does it point to curr?
```

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    Node<T> node = head; // start from head  
    while (node.key() <= pred.key()) { // does pred point to curr?  
        if (node == pred) return pred.next() == curr;  
        node = node.next(); // continue to the next node  
    } // until node.pred > pred.key  
    return false; // pred could not be reached  
}
```

How validation works

What can happen **between** the time when a thread **finds** a position (pred, curr) **and** when it **locks** nodes pred and curr?

- Node pred is removed: validation fails because pred is not reachable
- Node curr is removed: validation fails because pred does not point to curr
- A node is added between pred and curr: validation fails because pred does not point to curr
- Any other modification of the set: validation succeeds because operations leave the set in a consistent state

Is validation safe?

What happens if the set is being **modified while** a thread is **validating** a locked position (`pred`, `curr`)?

- If a node following `curr` is modified: validation is not affected because it only goes up until `curr`
- If a node `n` before `pred` is removed: validation succeeds even if it goes through `n`, since `n` still leads back to `pred`
- If a node `n` is added before `pred`: validation succeeds even if it skips over `n`

Optimistic-locking set: pros and cons

Pros:

- threads operating on disjoint portions of the list can operate in parallel
- when validation often succeeds, there is much less locking involved than in `FineSet`

Cons:

- `OptimisticSet` is not starvation free: a thread t may fail validation forever if other threads keep removing and adding `pred/curr` between when t performs `find` and when it locks `pred` and `curr`
- if traversing the list twice without locking is not significantly faster than traversing it once with locking, `OptimisticSet` does not have a clear advantage over `FineSet`

Parallel linked sets

Lazy node removal

Testing membership without locking

In many applications, operation has is executed **many more times** than add and remove. Can has work correctly **without locking**?

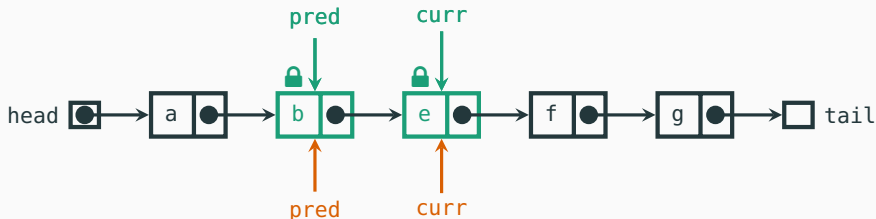
Problems may occur if another thread removes curr between find and has's check: since remove is not atomic without locking, if has does not acquire locks it may not notice that curr is being removed.

Testing membership without locking

In many applications, operation has is executed **many more times** than add and remove. Can has work correctly **without locking**?

Problems may occur if another thread removes curr between find and has's check: since remove is not atomic without locking, if has does not acquire locks it may not notice that curr is being removed.

For example, if **thread t** runs remove(e) while **thread u** runs has(e) without locking, **u** may incorrectly think that e is in the list even if **t** is about to remove it – that is **thread t** is in its critical section:



Nodes in a lazy-removal set

We need a way to **atomically** share the **information** that a node is being **removed**, but without locking.

To this end, each node includes a **flag valid** with setters and getters:

- `valid() == true`: the node is part of the set
- `valid() == false`: the node is being (or has been) removed

```
class ValidatedNode<T> extends ReadWriteNode<T>
{
    private volatile boolean valid;

    boolean valid()    { return valid; } // is node valid?
    void setValid()    { valid = true; } // mark valid
    void setInvalid() { valid = false; } // mark invalid
}
```

Nodes of type `ValidatedNode` can also be locked, since `ValidatedNode` inherits from `ReadWriteNode`.

Concurrent set with lazy node removal

In a lazy set:

- **Validation** only needs to check the mark `valid`
- Operation **remove** marks a node invalid before removing it
- Operation **has** is lock free
- Operation **add** works as in `OptimisticSet`

```
public class LazySet<T> extends OptimisticSet<T>
{
    public LazySet() {
        head = new ValidatedNode<>(Integer.MIN_VALUE); // smallest key
        tail = new ValidatedNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail);
    }

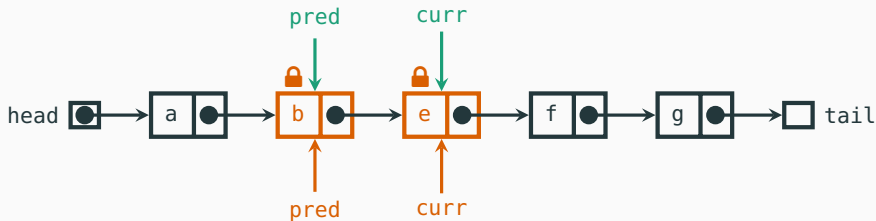
    // overriding of valid, remove, and has
}
```

Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** is marked **valid**

Scenario: `t`'s validation of `curr` succeeds:



// is pred reachable from head, and does it point to curr?

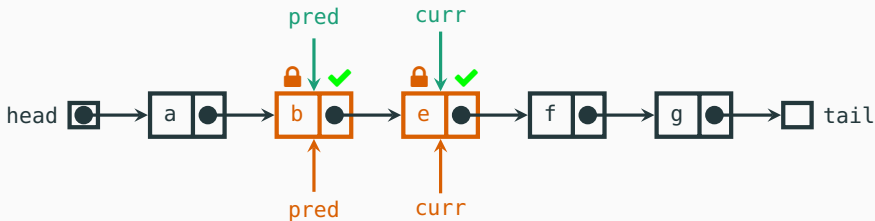
```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```

Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** is marked **valid**

Scenario: `t`'s validation of `curr` succeeds:



// is pred reachable from head, and does it point to curr?

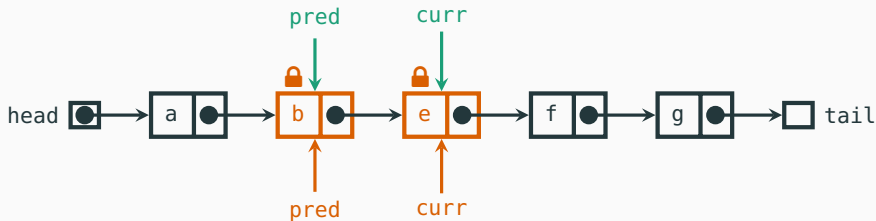
```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```


Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** is marked **valid**

Scenario: *t*'s validation of `curr` fails:



// is pred reachable from head, and does it point to curr?

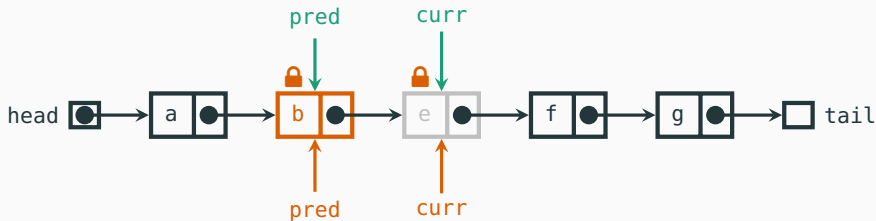
```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```

Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** is marked **valid**

Scenario: *t*'s validation of `curr` fails:



```
// is pred reachable from head, and does it point to curr?
```

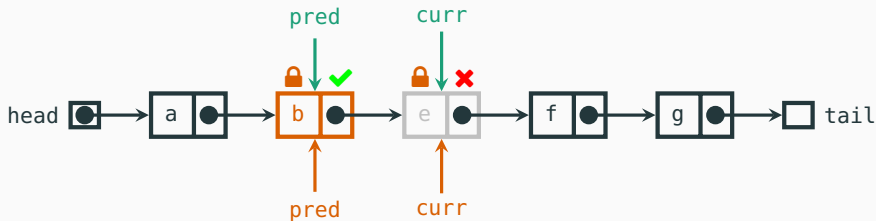
```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```

Lazy set: validating a position

Validation becomes a constant-time operation:

- Node `pred` is **reachable** from the head iff it has not been removed iff it is marked **valid**
- Node `curr` follows `pred` in the list iff `pred.next() == curr` **and** is marked **valid**

Scenario: *t*'s validation of `curr` fails:



// is pred reachable from head, and does it point to curr?

```
protected boolean valid(Node<T> pred, Node<T> curr) {  
    return pred.valid() && curr.valid() && pred.next() == curr;  
}
```

Lazy set: method has

Method `has` runs **without locking**: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s.

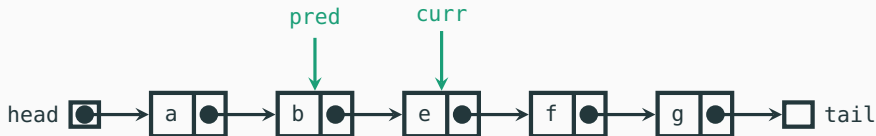


```
public boolean has(T item) {  
    // find position without locking  
    Node<T> pred, curr = find(head, item.key());  
    // check validity and item without locking  
    return curr.valid() && curr.key() == item.key();  
}
```

Method `find` may traverse **invalid nodes**; this does not prevent it from eventually reaching **all valid** nodes in the list.

Lazy set: method has

Method `has` runs **without locking**: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s.

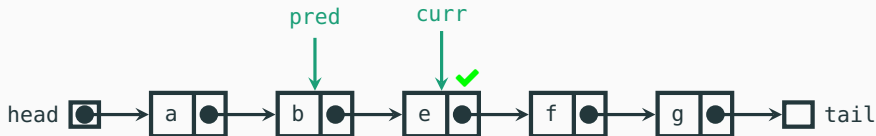


```
public boolean has(T item) {  
    // find position without locking  
    Node<T> pred, curr = find(head, item.key());  
    // check validity and item without locking  
    return curr.valid() && curr.key() == item.key();  
}
```

Method `find` may traverse **invalid nodes**; this does not prevent it from eventually reaching **all valid** nodes in the list.

Lazy set: method has

Method `has` runs **without locking**: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s.

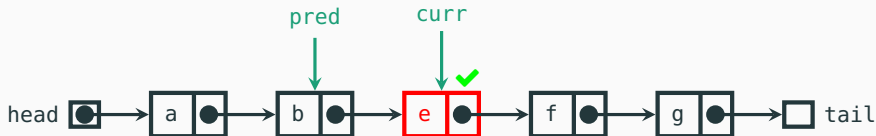


```
public boolean has(T item) {  
    // find position without locking  
    Node<T> pred, curr = find(head, item.key());  
    // check validity and item without locking  
    return curr.valid() && curr.key() == item.key();  
}
```

Method `find` may traverse **invalid nodes**; this does not prevent it from eventually reaching **all valid** nodes in the list.

Lazy set: method has

Method `has` runs **without locking**: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s.



```
public boolean has(T item) {  
    // find position without locking  
    Node<T> pred, curr = find(head, item.key());  
    // check validity and item without locking  
    return curr.valid() && curr.key() == item.key();  
}
```

Method `find` may traverse **invalid nodes**; this does not prevent it from eventually reaching **all valid** nodes in the list.

Lazy set: method add

Method `add` works as in `OptimisticSet`, but using the overridden version of `valid` – which works in constant time.

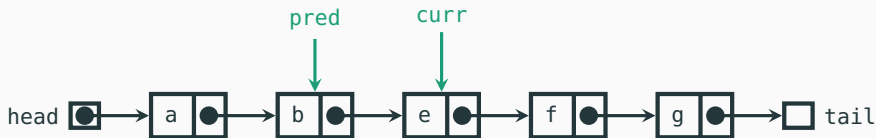


node:



Lazy set: method add

Method `add` works **as in `OptimisticSet`**, but using the overridden version of `valid` – which works in constant time.

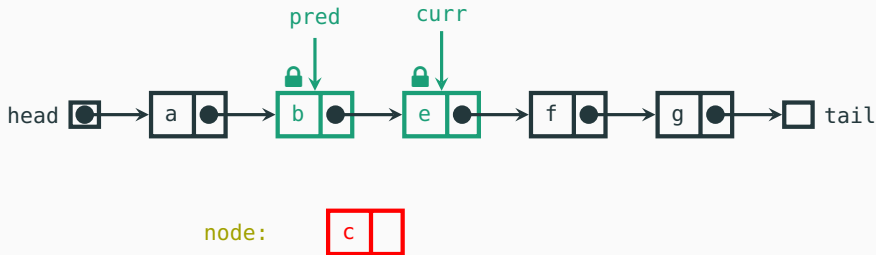


node:



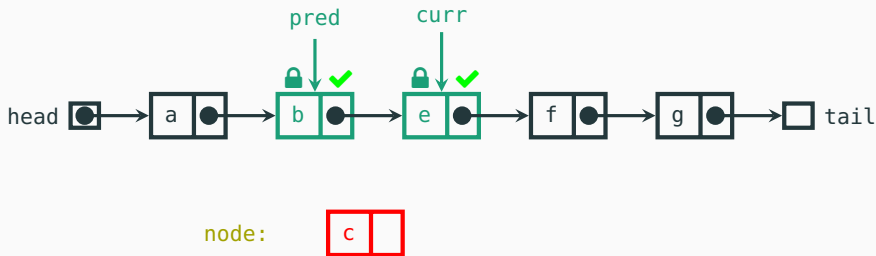
Lazy set: method add

Method `add` works as in `OptimisticSet`, but using the overridden version of `valid` – which works in constant time.



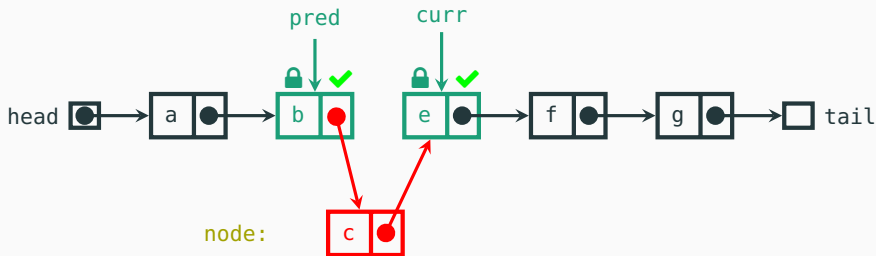
Lazy set: method add

Method `add` works as in `OptimisticSet`, but using the overridden version of `valid` – which works in constant time.



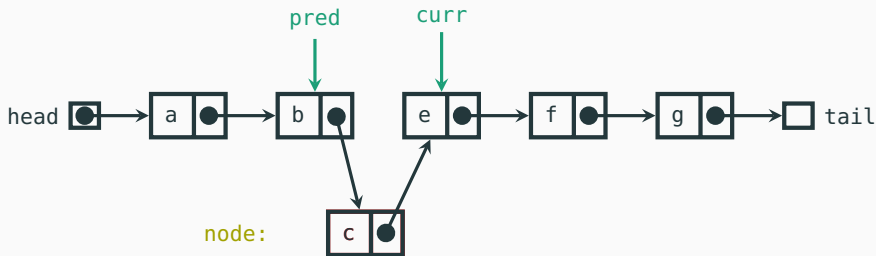
Lazy set: method add

Method `add` works **as in `OptimisticSet`**, but using the overridden version of `valid` – which works in constant time.



Lazy set: method add

Method `add` works **as in `OptimisticSet`**, but using the overridden version of `valid` – which works in constant time.



Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next

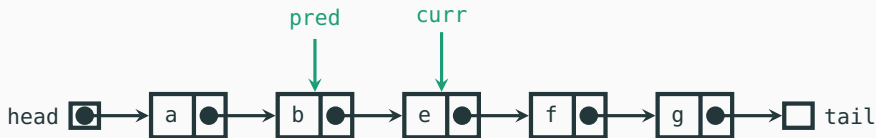


This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next

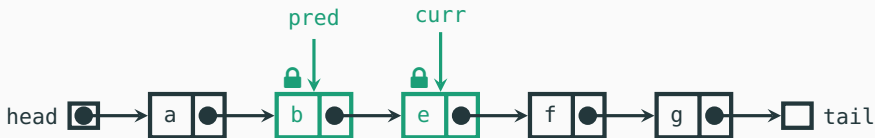


This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next

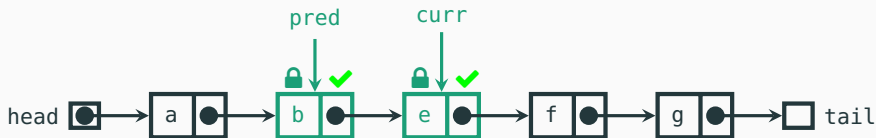


This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next

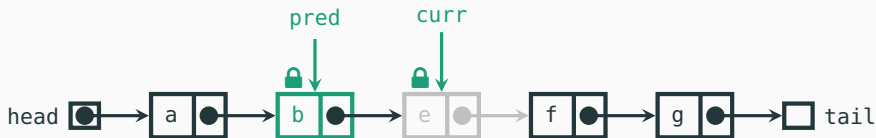


This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next



This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

After finding the position of a node to be removed, the **actual removal** consists of two steps:

1. **logical removal**: mark the node to be removed as invalid
2. **physical removal**: skip over the node by redirecting its predecessor's next



This removal is **lazy** because logical and physical removal may be done at different times: after a node has been logically removed, every thread is aware that it should not be considered part of the list.

Lazy set: method remove

```
public boolean remove(T item) {
    do { Node<T> pred, curr = find(head, item.key()); // no locking
        pred.lock(); curr.lock(); // now lock position
        try { // if position still valid, while locking:
            if (valid(pred, curr)) {
                if (curr.key() != item.key())
                    return false; // item not in the set
                else { // item in the set at curr: remove it
                    curr.setInvalid(); // logical removal
                    pred.setNext(curr.next()); // physical removal
                    return true;
                }
            }
        } finally { pred.unlock(); curr.unlock(); } // done: unlock
    } while (true); // if not valid: try again!
}
```

Lazy-removal set: pros and cons

Pros:

- validation is constant time
- membership checking does not require any locking – it's even **wait free** (it traverses the list once without locking)
- physical removal of logically removed nodes could be **batched** and performed when convenient – thus reducing the number of times the physical chain of nodes is changed, in turn reducing the expensive propagation of information between threads

Cons:

- operations `add` and `remove` still require locking (as in `OptimisticSet`), which may reduce the amount of parallelism

Parallel linked sets

Lock-free access

Atomic references

To implement a set that is correct under concurrent access without using any locks we need to rely on **synchronization primitives** more **powerful** than just reading and writing shared variables.

We are going to use a variant of the **compare-and-set** operation (also called test-and-set).

```
class AtomicReference<V> {  
  
    V get();                // current reference  
    void set(V newRef);    // set reference to newRef  
  
    // if reference == expectRef, set to newRef and return true  
    // otherwise, do not change reference and return false  
    boolean compareAndSet(V expectRef, V newRef);  
}
```

Atomic lock-free access: first attempt

As a **first attempt**, we make attribute `next` of type `AtomicReference<Node<T>>`, and use `compareAndSet` to update it: if one thread changes `next` when another thread is also trying to change it, we **repeat** the operation.

An implementation of `remove()` following this idea:

```
public boolean remove(T item) {
    boolean done;
    do {
        Node<T> pred, curr = find(head, item.key());
        if (curr.key() >= item.key()) return false; // item not in set
        // try to remove curr by setting pred.next using compareAndSet
        else done = pred.next.compareAndSet(pred.next(), curr.next());
    } while (!done); return true;
}
```

↑
`pred.next` may have changed
when `compareAndSet()` executes

Atomic lock-free access: first attempt

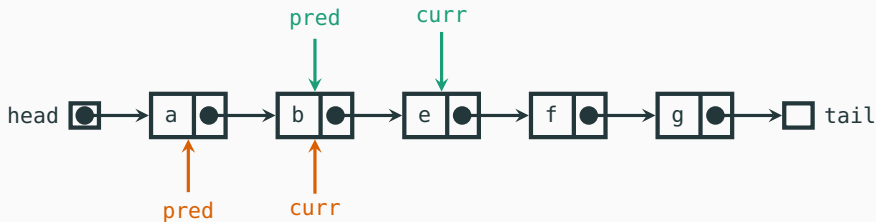
Unfortunately, the first attempt **does not work**: for example, if **thread t** runs `remove(e)` while **thread u** runs `remove(b)`, it may happen that only `b`'s removal takes place.



We have seen a similar problem before: modifications of the list need to have **control** of **both pred and curr** – even if it is only the former node that is actually modified.

Atomic lock-free access: first attempt

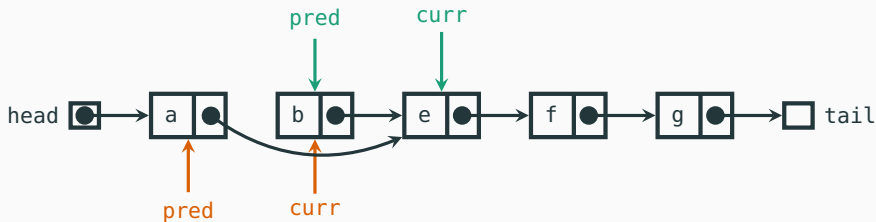
Unfortunately, the first attempt **does not work**: for example, if **thread *t*** runs `remove(e)` while **thread *u*** runs `remove(b)`, it may happen that only `b`'s removal takes place.



We have seen a similar problem before: modifications of the list need to have **control** of **both pred and curr** – even if it is only the former node that is actually modified.

Atomic lock-free access: first attempt

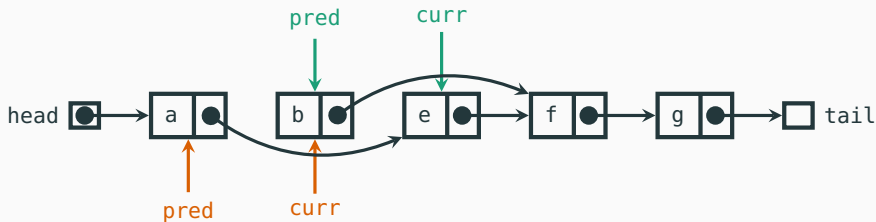
Unfortunately, the first attempt **does not work**: for example, if **thread t** runs `remove(e)` while **thread u** runs `remove(b)`, it may happen that only `b`'s removal takes place.



We have seen a similar problem before: modifications of the list need to have **control** of **both pred and curr** – even if it is only the former node that is actually modified.

Atomic lock-free access: first attempt

Unfortunately, the first attempt **does not work**: for example, if **thread t** runs `remove(e)` while **thread u** runs `remove(b)`, it may happen that only `b`'s removal takes place.



We have seen a similar problem before: modifications of the list need to have **control** of **both `pred` and `curr`** – even if it is only the former node that is actually modified.

Atomic markable references

As in LazySet, nodes can be marked valid or invalid; an invalid node is logically removed. In addition, we need to access the information of **both** attributes `valid` and `next` **atomically**; to this end, every node includes an attribute `nextValid` of type `AtomicMarkableReference<Node<T>>`, which provides methods to update a reference and a mark atomically and together.

```
class AtomicMarkableReference<V> {  
    V, boolean get();           // current reference and mark  
    // if reference == expectRef set mark to newMark and return true  
    // otherwise do not change anything and return false  
    boolean attemptMark(V expectRef, boolean newMark);  
    // if reference == expectRef and mark == expectMark,  
    // set reference to newRef, mark to newMark and return true;  
    // otherwise, do not change anything and return false  
    boolean compareAndSet(V expectRef, V newRef,  
                          boolean expectMark, boolean newMark);  
}
```

Nodes in a lock-free set

Every node includes an **attribute** `nextValid` of type `AtomicMarkableReference<Node<T>>`. The node interface provides methods to retrieve and conditionally update the current value of `nextValid`, which includes **a reference** (corresponding to `next`) and **a mark** (corresponding to `valid`).

```
class LockFreeNode<T> extends SequentialNode<T> {  
    // reference to next node and validity mark of current node  
    private AtomicMarkableReference<Node<T>> nextValid;  
  
    // return next and valid as a pair  
    Node<T>, boolean nextValid() { return nextValid.get(); }  
    pseudo-code for a Node<T> where only next and valid are set  
    Node<T> next()  
    { Node<T> next, boolean valid = nextValid(); return next; }  
    boolean valid()  
    { Node<T> next, boolean valid = nextValid(); return valid; }  
}
```

Nodes in a lock-free set

Every node includes an **attribute** `nextValid` of type `AtomicMarkableReference<Node<T>>`. The node interface provides methods to retrieve and conditionally update the current value of `nextValid`, which includes **a reference** (corresponding to `next`) and **a mark** (corresponding to `valid`).

```
class LockFreeNode<T> extends SequentialNode<T> {  
  
    // try to set invalid; return true if successful  
boolean setInvalid()  
    { Node<T> next = next();  
      return nextValid.compareAndSet(next, next, true, false); }  
  
    // try to update to newNext if valid; return true if successful  
boolean setNextIfValid(Node<T> expectNext, Node<T> newNext)  
    { return nextValid.compareAndSet(expectNext, newNext, true, true); }  
      
    update next only if the node is valid
```

Concurrent set with lock-free access

In a lock-free set:

- Operation `remove` marks a node **invalid** before removing it
- Operations that modify nodes complete successfully **only** if the nodes are **valid** and **not concurrently modified** by another thread
- Failed operations are **repeated** until success (no interference)

```
public class LockFreeSet<T> extends SequentialSet<T>
{
    public LockFreeSet() {
        head = new LockFreeNode<>(Integer.MIN_VALUE); // smallest key
        tail = new LockFreeNode<>(Integer.MAX_VALUE); // largest key
        head.setNext(tail); // unconditionally set next
                           // only in new nodes
    }

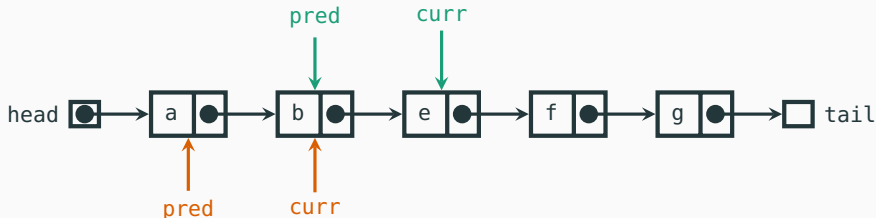
    // overriding of all methods
}
```


Lock-free set: method remove



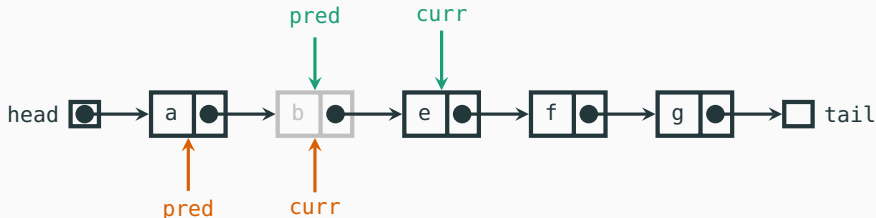
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



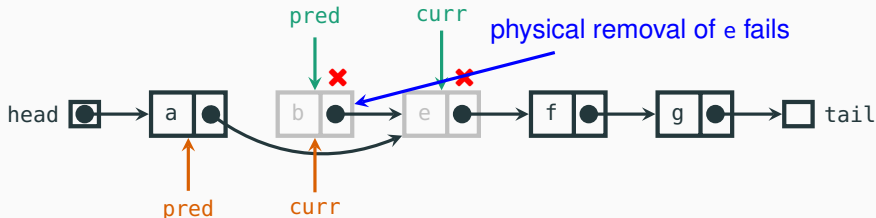
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



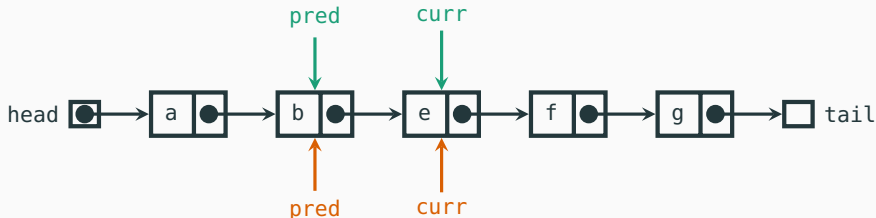
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next()); ← physical removal  
        return true;                                     of e fails: never mind!  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



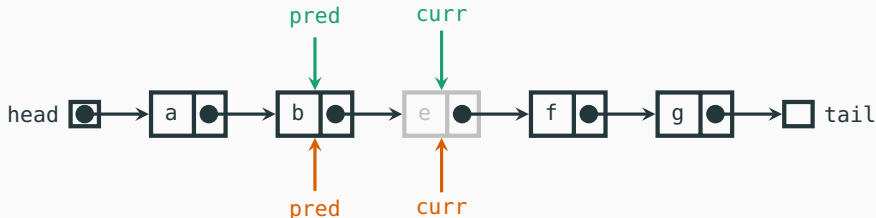
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



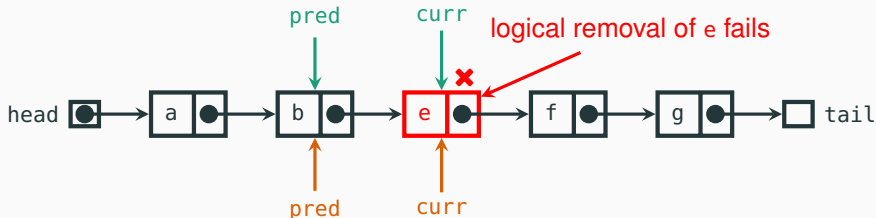
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



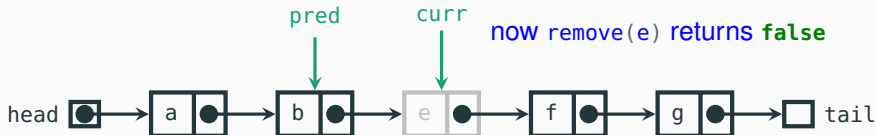
```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue; ← logical removal  
        // try once to physically remove curr of e fails: retry!  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```


Lock-free set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Lock-free set: method remove



```
public boolean remove(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // not in set  
        if (curr.key() != item.key() || !curr.valid()) return false;  
        // try to invalidate; try again if node is being modified  
        if (!curr.setInvalid()) continue;  
        // try once to physically remove curr  
        pred.setNextIfValid(curr, curr.next());  
        return true;  
    } while (true); // changed during logical removal: try again!  
}
```

Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

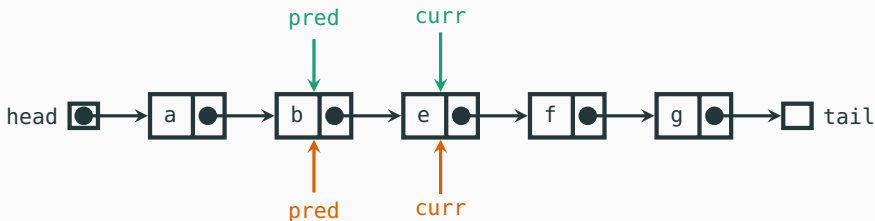


Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

- The same element may be removed twice

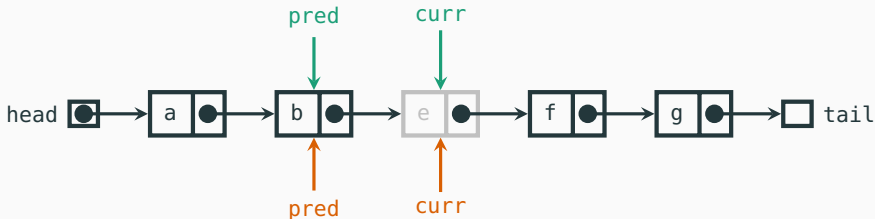


Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

- The same element may be removed twice

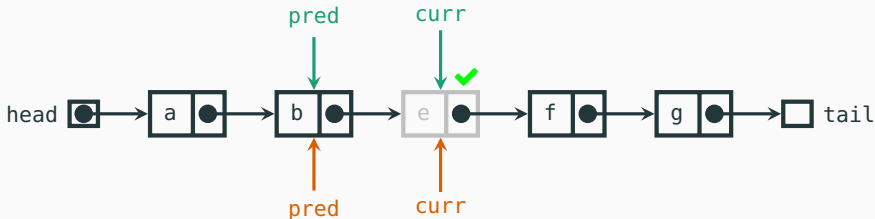


Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

- The same element may be removed twice

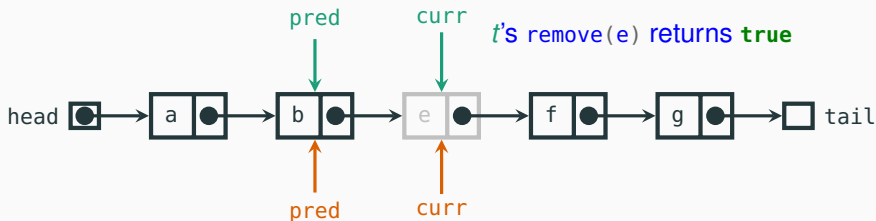


Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

- The same element may be removed twice

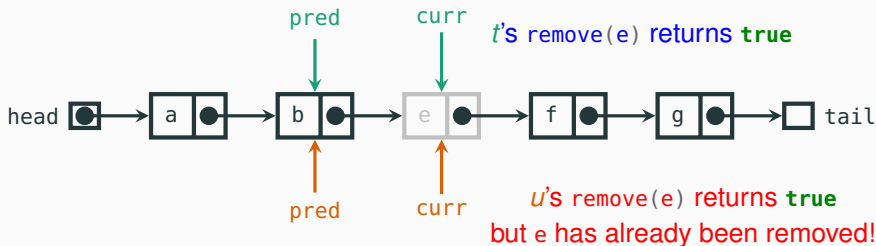


Logical removal: only one thread succeeds

If two threads both try to mark a node invalid, **only one can succeed** – so it is guaranteed that no other thread is touching the node.

If this property was not enforced:

- The same element may be removed twice

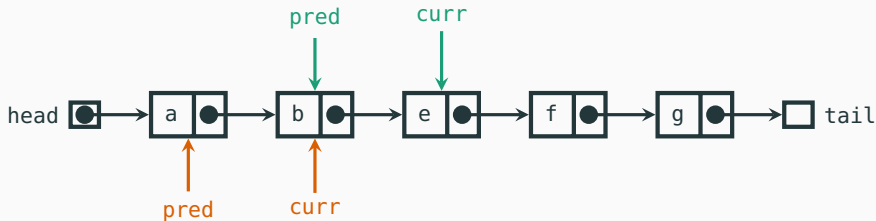


Lock-free set: method add



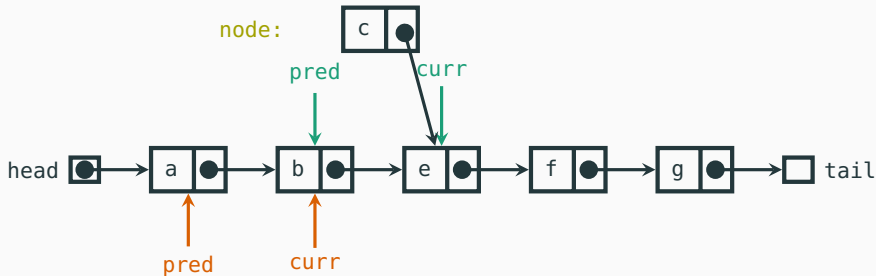
```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



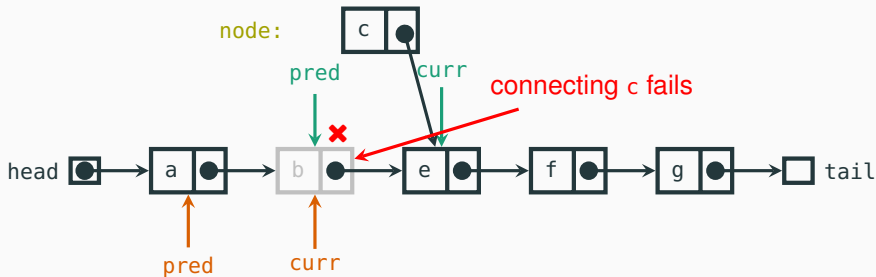
```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



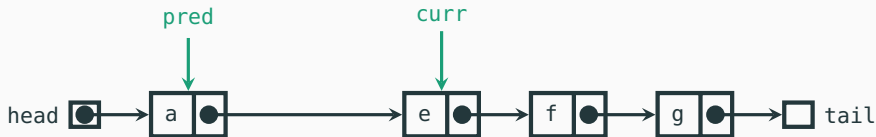
```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true; ← connecting  
    } while (true); // pred changed during add: try again! c fails: retry!  
}
```

Lock-free set: method add



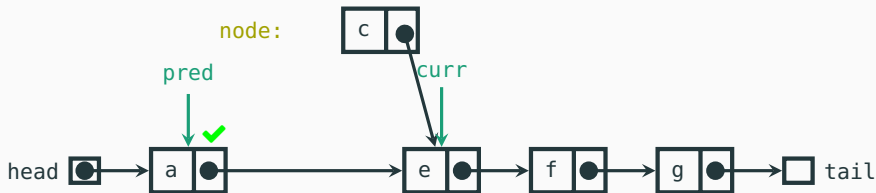
```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



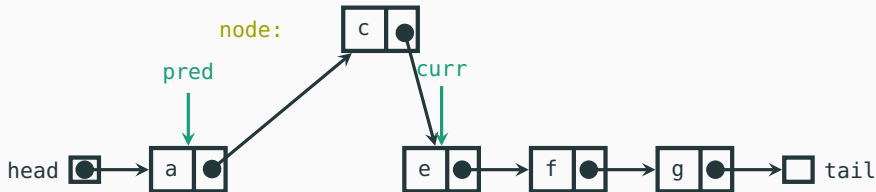
```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```

Lock-free set: method add



```
public boolean add(T item) {  
    do { Node<T> pred, curr = find(head, item.key()); // already in set  
        if (curr.key() == item.key() && curr.valid()) return false;  
        // new node, pointing to curr  
        Node<T> node = new LockFreeNode<>(item).setNext(curr);  
        // if pred valid and points to curr, make it point to node  
        if (pred.setNextIfValid(curr, node)) return true;  
    } while (true); // pred changed during add: try again!  
}
```


Lock-free set: method has

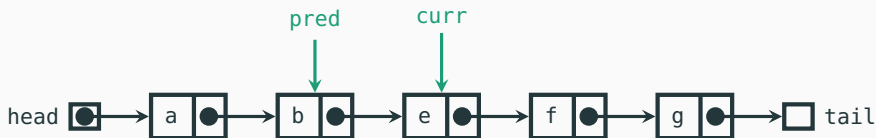
Method `has` works as in `LazySet`: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s. Unlike `add` and `remove` (which use a new version of `find`), `has` traverses both valid and invalid nodes, and makes **no attempt at removing** the latter.



```
public boolean has(T item) {  
    // find position (use plain search in SequentialSet)  
    Node<T> pred, curr = super.find(head, item.key());  
    // check validity and item  
    return curr.valid() && curr.key() == item.key();  
}
```

Lock-free set: method has

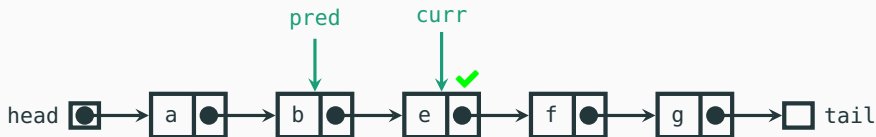
Method `has` works as in `LazySet`: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s. Unlike `add` and `remove` (which use a new version of `find`), `has` traverses both valid and invalid nodes, and makes **no attempt at removing** the latter.



```
public boolean has(T item) {  
    // find position (use plain search in SequentialSet)  
    Node<T> pred, curr = super.find(head, item.key());  
    // check validity and item  
    return curr.valid() && curr.key() == item.key();  
}
```

Lock-free set: method has

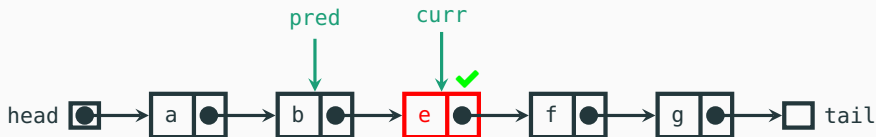
Method `has` works as in `LazySet`: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s. Unlike `add` and `remove` (which use a new version of `find`), `has` traverses both valid and invalid nodes, and makes **no attempt at removing** the latter.



```
public boolean has(T item) {  
    // find position (use plain search in SequentialSet)  
    Node<T> pred, curr = super.find(head, item.key());  
    // check validity and item  
    return curr.valid() && curr.key() == item.key();  
}
```

Lock-free set: method has

Method `has` works as in `LazySet`: it finds the position (`pred`, `curr`), validates `curr`, and checks whether `curr`'s key is equal to `item`'s. Unlike `add` and `remove` (which use a new version of `find`), `has` traverses both valid and invalid nodes, and makes **no attempt at removing** the latter.



```
public boolean has(T item) {  
    // find position (use plain search in SequentialSet)  
    Node<T> pred, curr = super.find(head, item.key());  
    // check validity and item  
    return curr.valid() && curr.key() == item.key();  
}
```

When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure.

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes **encountered by `find`**. This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes.

For example, the logical removal of nodes `f` and `g` requires **thread `t`** to physically remove `f` before it can physically remove `g`:

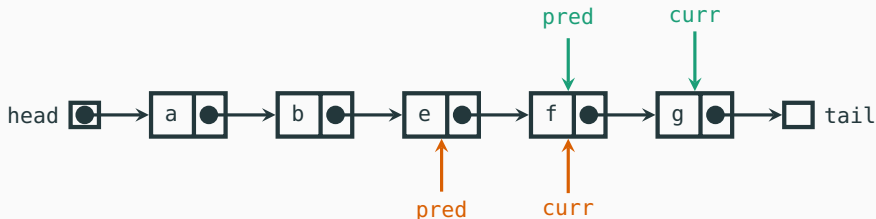


When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure.

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes **encountered by `find`**. This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes.

For example, the logical removal of nodes `f` and `g` requires **thread `t`** to physically remove `f` before it can physically remove `g`:

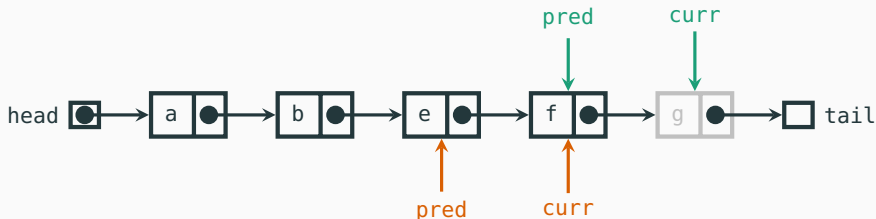


When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure.

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes **encountered by `find`**. This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes.

For example, the logical removal of nodes `f` and `g` requires **thread `t`** to physically remove `f` before it can physically remove `g`:

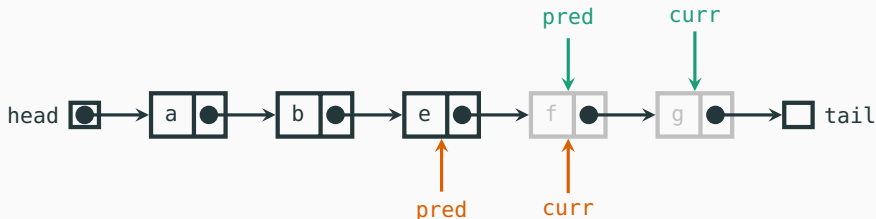


When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure.

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes **encountered by `find`**. This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes.

For example, the logical removal of nodes `f` and `g` requires **thread `t`** to physically remove `f` before it can physically remove `g`:

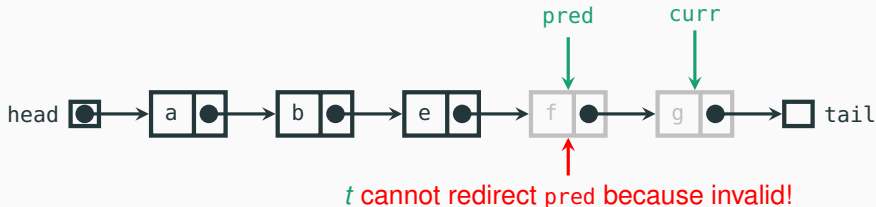


When to physically remove nodes?

Method `has` does not modify the set, so it can safely traverse valid and invalid nodes without changing the node structure.

In contrast, methods `add` and `remove` **physically remove** all logically removed nodes **encountered by `find`**. This is a convenient time to perform physical removal, because it avoids the buildup of long chains of invalid nodes.

For example, the logical removal of nodes `f` and `g` requires **thread `t`** to physically remove `f` before it can physically remove `g`:



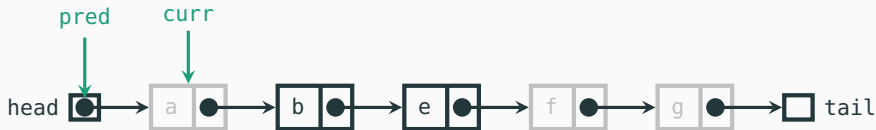
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



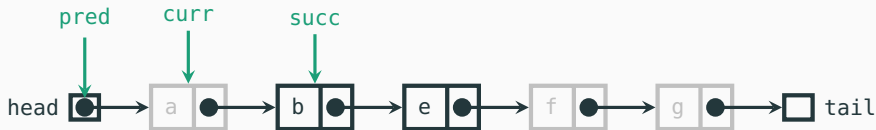
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



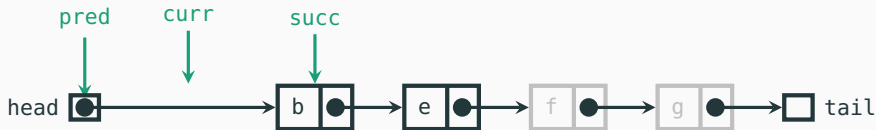
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



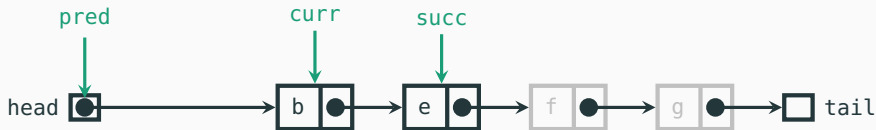
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



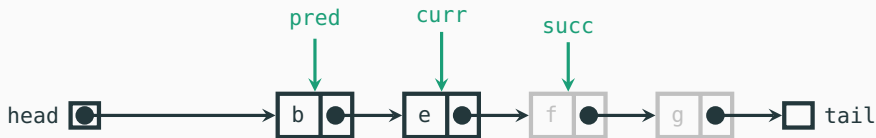
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



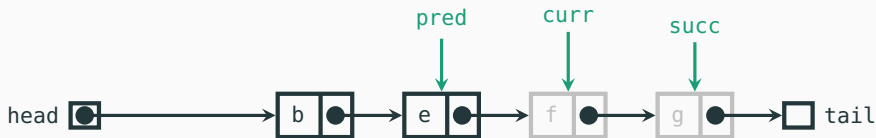
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



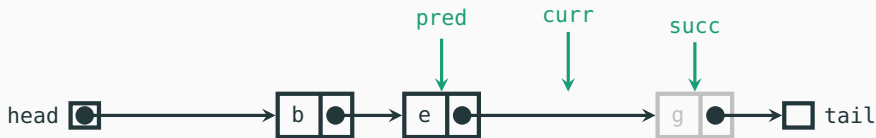
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



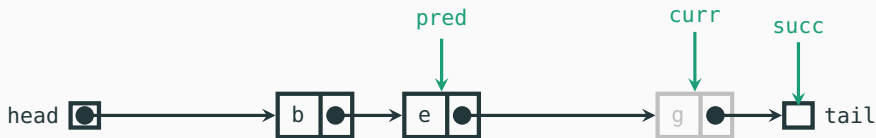
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



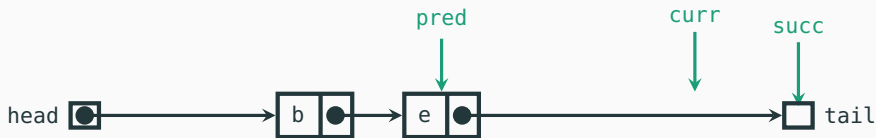
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



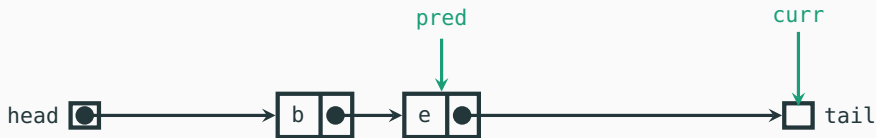
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



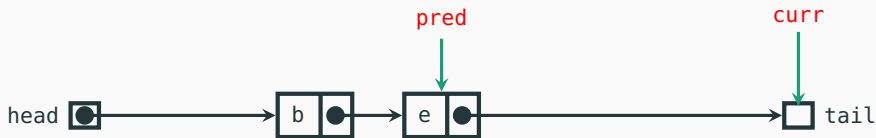
Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



Lock-free set: how find works

A run of `find(k)` that also physically removes three invalid nodes.



Threads may interfere with `find`, requiring to restart it; in the worst case, **starvation** may occur with a thread continuously restarting `find` while others make progress modifying the list.

Lock-free set: method find

```
protected Node<T>, Node<T> find(Node<T> start, int key) {
    boolean valid;           // is curr valid?
    Node<T> pred, curr, succ; // consecutive nodes in iteration
    retry: do {
        pred = start; curr = start.next(); // from start node
        do { // succ is curr's successor; valid is curr's validity
            succ, valid = curr.nextValid();
            while (!valid) { // while curr is not valid, try to remove it
                // if pred is modified while trying to redirect it, retry
                if (!pred.setNextIfValid(curr, succ)) continue retry;
                // curr has been physically removed: move to next node
                curr = succ; succ, valid = curr.nextValid();
            } // now curr is valid (and so is pred)
            if (curr.key() >= key) return (pred, curr);
            pred = curr; curr = succ; // continue search
        } while (true);
    } while (true);
}
```

Lock-free set: pros and cons

Pros:

- all operations do not require locking: maximum potential for parallelism
- membership checking does not require any locking – it's even **wait free** (it traverses the list once without locking)

Cons:

- the implementation needs test-and-set-like synchronization primitives, which have to be supported and come with their own performance costs
- operations `add` and `remove` are lock free but not wait free: they may have to repeat operations, and they may be delayed while they physically remove invalid nodes, with the risk of introducing contention on nodes that have been already previously logically deleted

To lock or not to lock?

Each of the different implementations of concurrent set is the **best choice** for certain **applications** and not for others:

- CoarseSet works well with low contention
- FineSet works well when threads tend to access the list orderly
- OptimisticSet works well to let threads operate on disjoint portions of the list
- LazySet works well when batching invalid node removal is convenient
- LockFreeSet works well when locking is quite expensive

These slides' license

© 2016–2018 Carlo A. Furia



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.