Chapter 9.1-9.3
Linked Lists: The Role of Locking

Magnus Andersson

Introduction

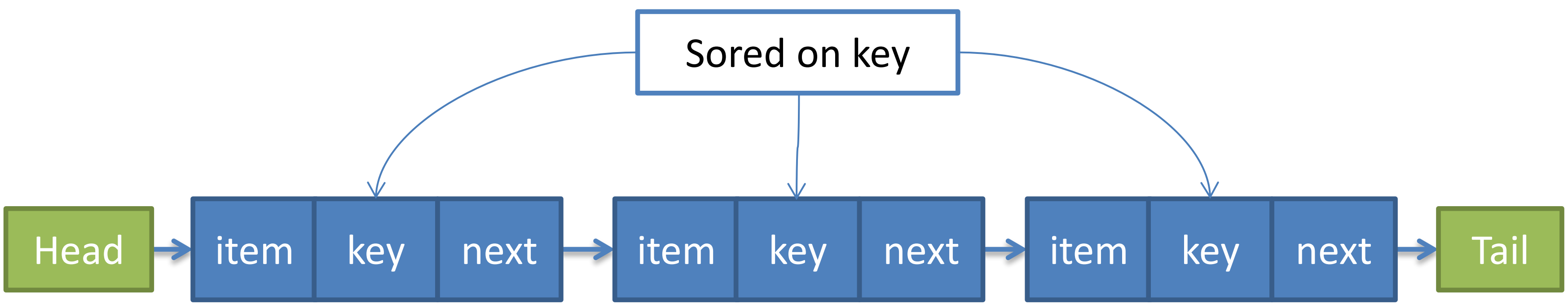
- Fine-grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- Non-blocking Synchronization

Set on top of a *Linked list*

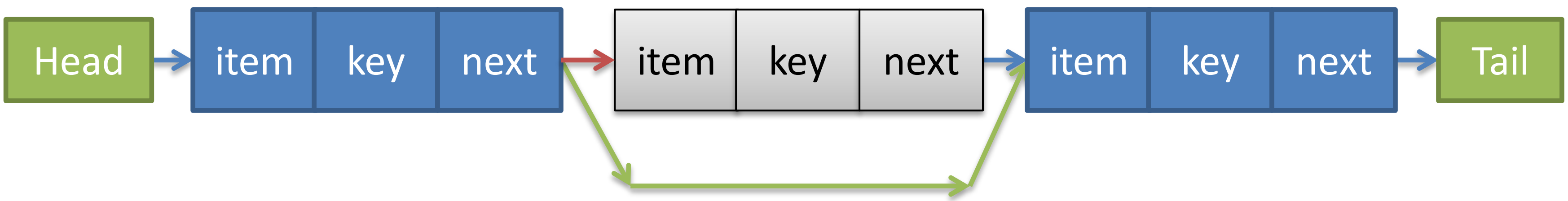
```
public interface Set<T> {
    boolean add(T x);
    boolean remove(T x);
    boolean contains(T x);
}
```

Linked list node

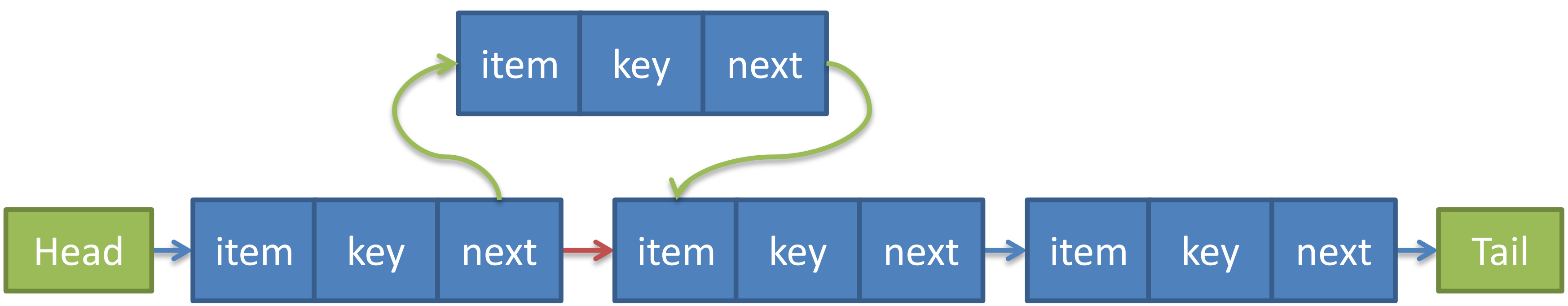
```
private class Node {
      volatile T item;
      volatile int key;
      volatile Node next;
}
```



Linked list node

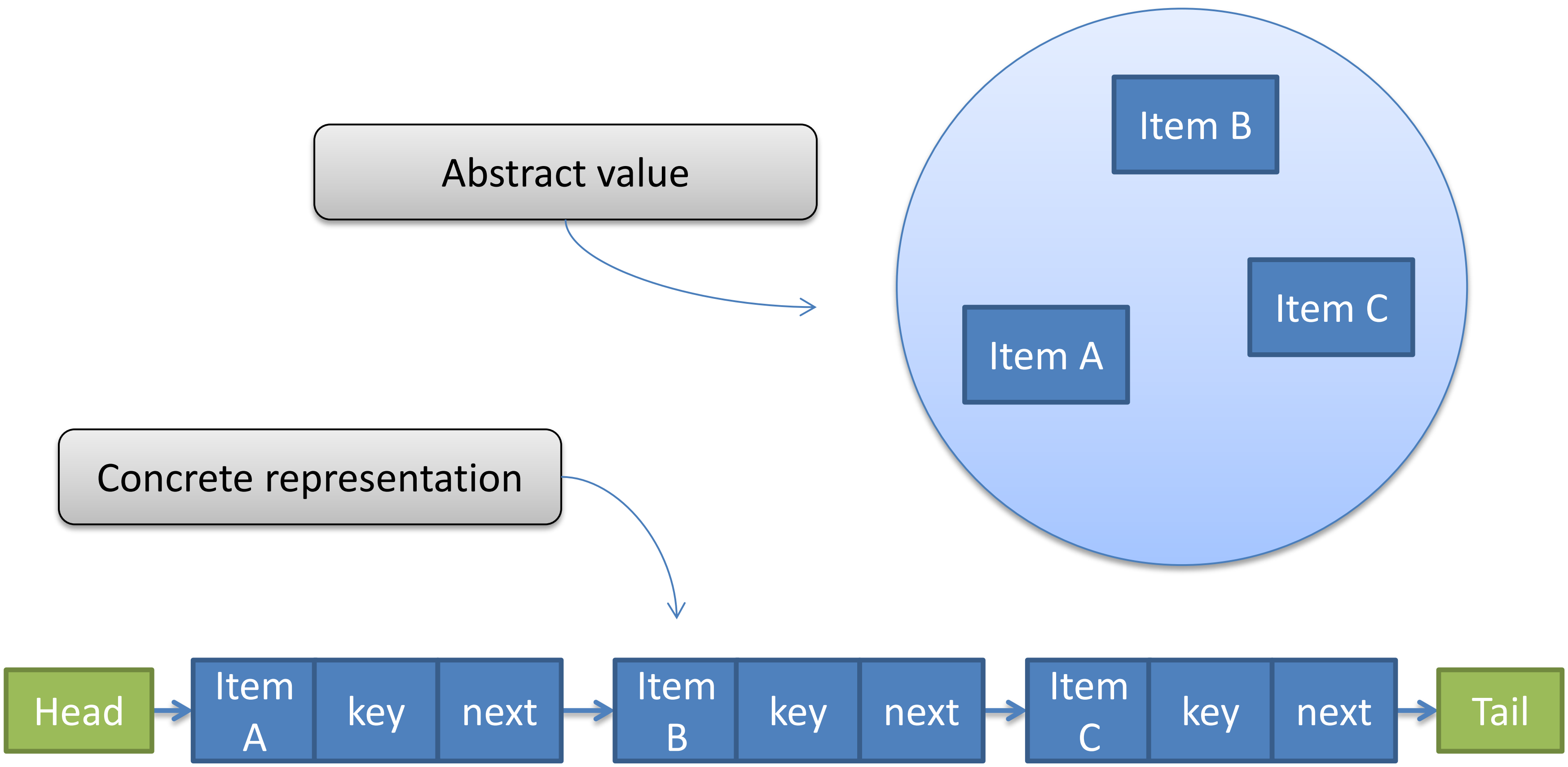


Removing



Adding

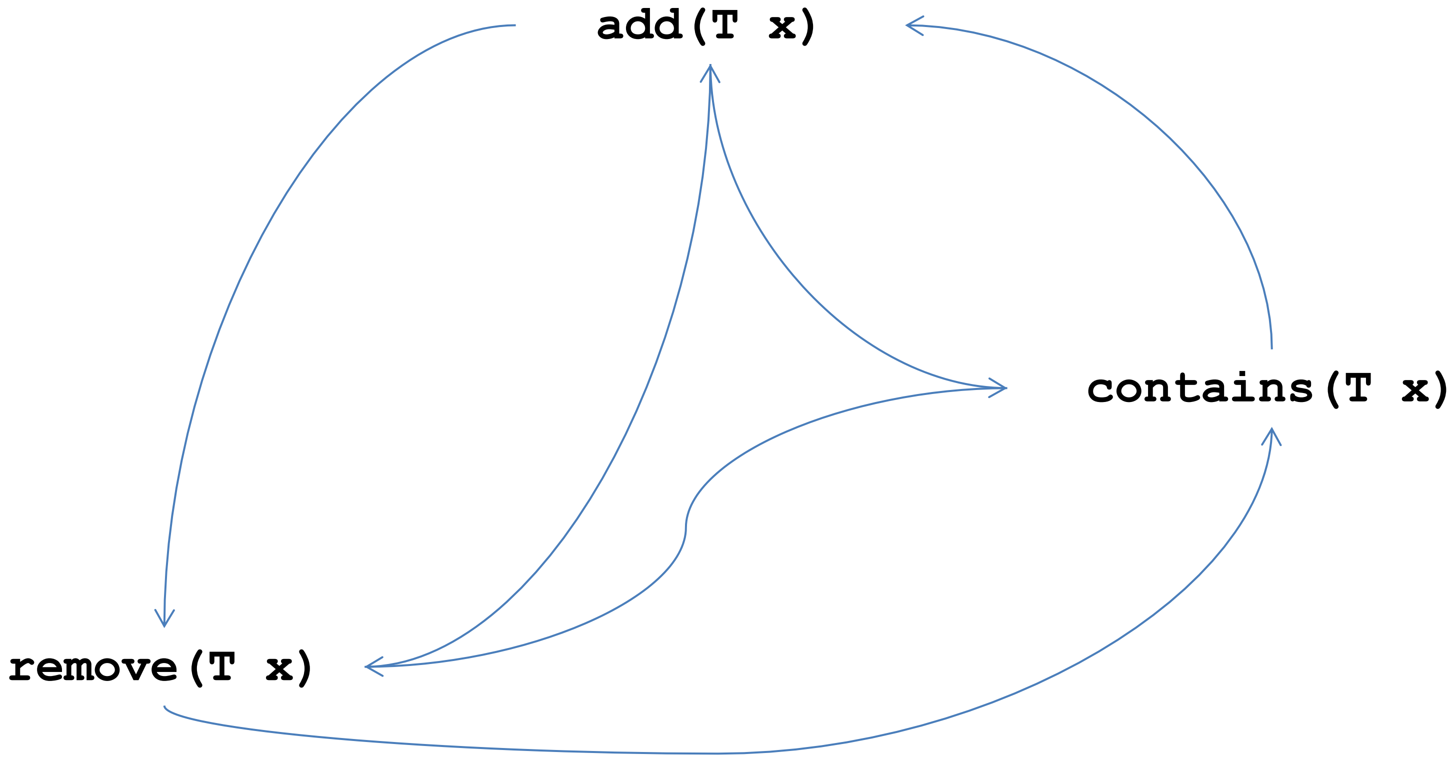
Concurrent Reasoning



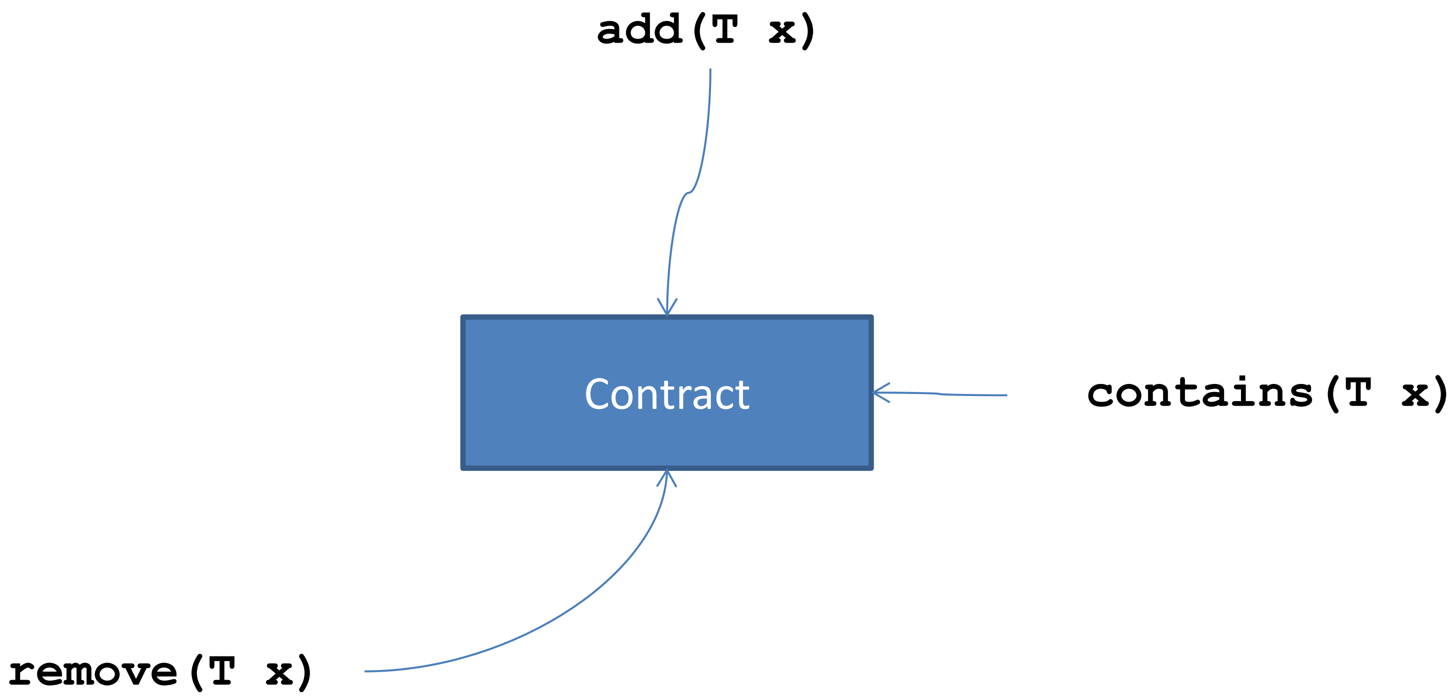
Concurrent Reasoning

- *Invariants*
  - Properties that hold from creation and onward
  - Always preserved by methods
- In this case **`add`**, **`remove`** and **`contains`** could potentially break invariants
  - Why only these?
    - Private
      - *Freedom from interference*

Concurrent Reasoning



Concurrent Reasoning



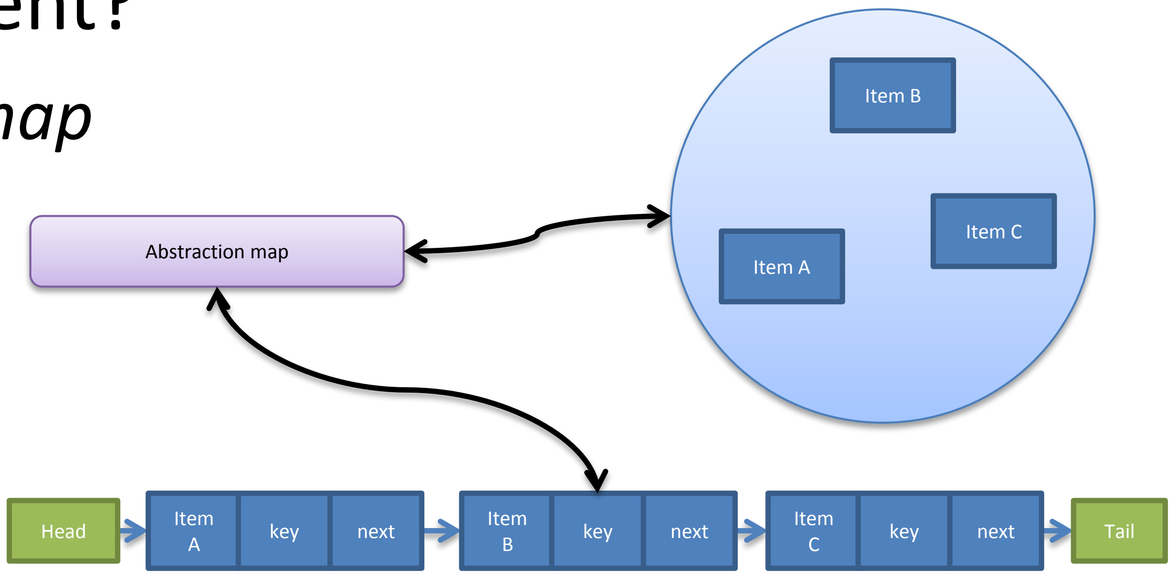
Concurrent Reasoning

- *Representation invariant*
  - Contract
  - Pre- and Post-conditions that everyone agrees on
    - Concrete representation must make sense for the given abstract values

- In this case:
  - Sentinels may not be added or removed
  - Sorted by unique keys

Concurrent Reasoning

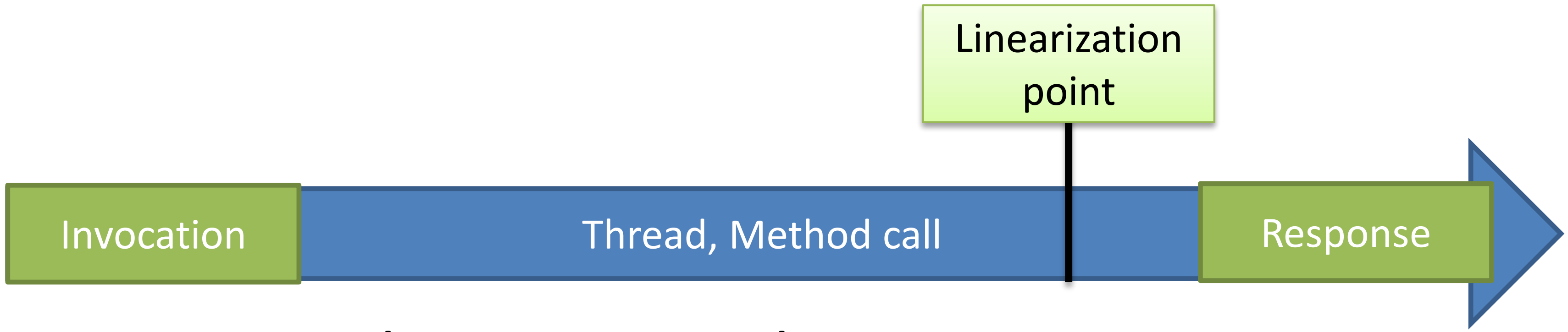
- Given a concrete representation, which set does it represent?
  - *Abstraction map*



- In this case:
  - All values reachable from the head

Concurrent Reasoning

- Consistency model
  - Linearizability
    - Single “operation” (atomic, critical section)



  - Stricter than sequential consistency
- Some examples can be found at
  - http://kisalay.com/2011/04/26/linearizability-3/

Concurrent Reasoning

- Serializable
  - Parallel execution must appear to be executed serially
  - May reorganize tasks between threads “out-of-order”
- Linearizablity
  - Stricter ordering between method calls on an object
    - Response -> Invocation order must be maintained
- Chapter 3 in the book…

Concurrent Reasoning

- Non-blocking properties
  - lock-free (*“Some calls* finishes after a number of steps”)
    - Non-blocking
    - Allows for concurrent access without corruption
      - Though not without restrictions (read the friendly manual)
    - Even if one thread is suspended, others must be able to access uncorrupted data
    - Must retry on failure (unbounded)
  - wait-free (*“All calls* finishes after a number of steps”)
    - Bounded number of steps
    - Always make progress

Concurrent Reasoning

	Non-Blocking	Blocking
All make progress	Wait-free	Starvation-free
Some make progress	Lock-free	Deadlock-free

Exercises

1. Give two examples of scenarios (not the ones from the chapter) where you think it would be practical and/or performant to use…
   – *a)* a wait-free method
   – *b)* a lock-free method

2. Where it makes sense to do so:

From the assignments given by the other students this lecture – categorize each method implemented to one of the four blocking categories. Motivate your conclusions.

	Non-Blocking	Blocking
All make progress	Wait-free	Starvation-free
Some make progress	Lock-free	Deadlock-free

Fin

Concurrent Reasoning [BACKUP]

- Liveness and safety considerations
  - Safety property: "Linearizability"
    - "Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions."
      - M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. TOPLAS, 12:463–492, 1990.

Concurrent Reasoning [BACKUP]

- Linearizability
  - Wikipedia has a fairly comprehensible explanation
  - http://en.wikipedia.org/wiki/Linearizability
- Non-blocking properties
  - A good description can be found here:
  - http://www.justsoftwaresolutions.co.uk/threading/non_blocking_lock_free_and_wait_free.html