Concurrent List-Based Sets

fine-grained, optimistic and lazy synchronization

INF346, 2015

Implementing a scalable concurrent data structure?

- What *is* a concurrent data structure?
  - ✓Sequential type
  - ✓Wait-free
  - ✓Linearizable
- What is scalable?
  - ✓Throughput: the number of complete operations per time unit
  - ✓Workload:
  - ✓Throughput scales with the growing workload
- Typically, better concurrency translates to better
  - ✓The "number" of accepted concurrent schedules

Example: set type

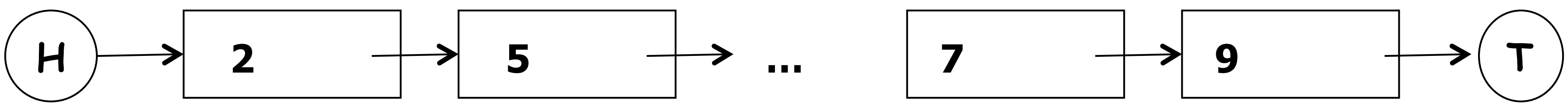
A set abstraction stores a set of integers (no duplicates) and exports operations:

- add(x) – adds x to the set and returns true if and only if x is not in the set
- remove(x) – remove x from the set and returns true if and only if x is in the set
- contains(x) – returns true if and only if x is in the set

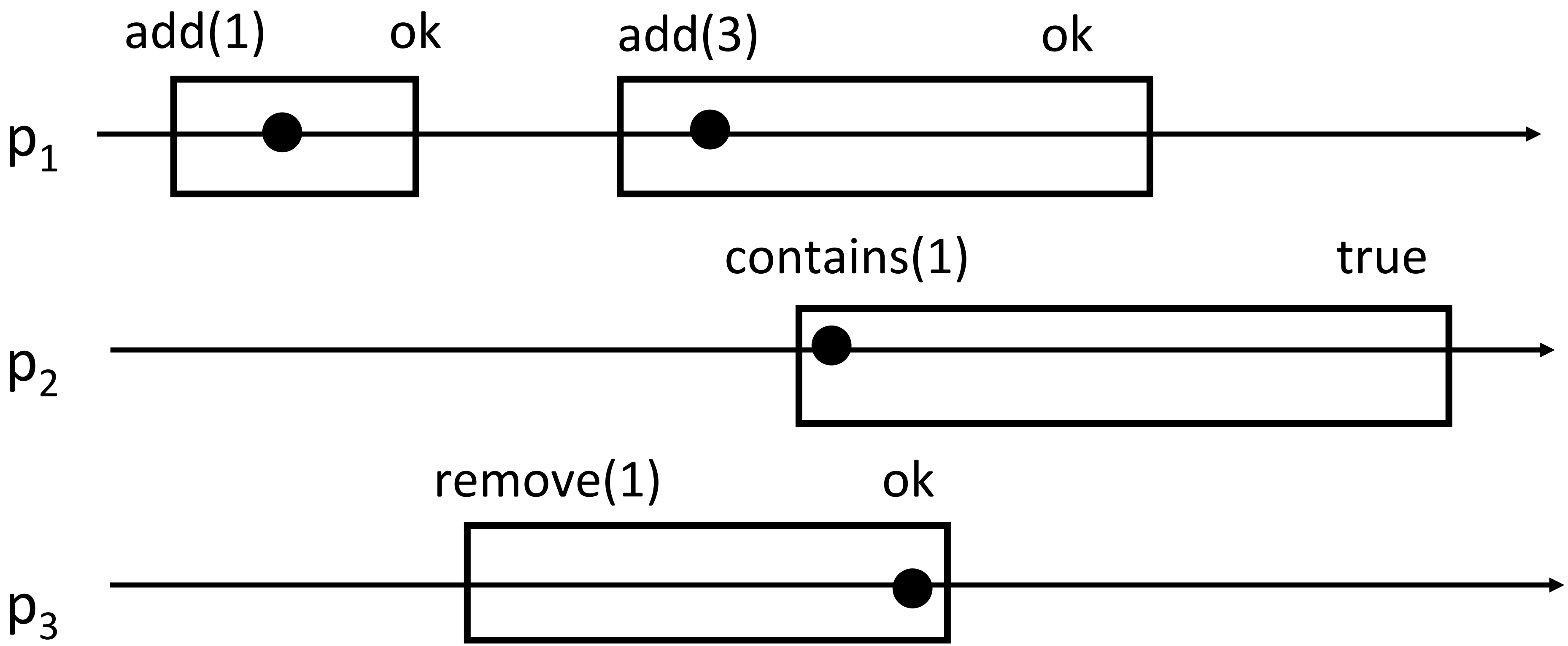
Sequential list-based set

Implementing a set using a sorted linked list:

- To locate x, search starting from the head *curr* points to the first node storing x'≥x, *prev* points to its predecessor
- To add x (if x'=x), point *prev.next* to *curr.next*
- To insert x (if x'>x), set *prev.next* to the new node storing x and pointing to *curr*



Linearizable histories



The history is equivalent to a legal sequential history on a set (real-time order preserved)

Linked-list for Set: sequential implementation

```
/* The node of an integer list. At creation, default pointer
   is null */
public class Node{
    Node(int item){key=item;next=null;}
    public int key;
    public Node next;}


public class SetList{

    private Node head;

    public SetList(){
          head = new Node(Integer.MIN_VALUE);
          head.next = new Node(Integer.MAX_VALUE);
    }
```

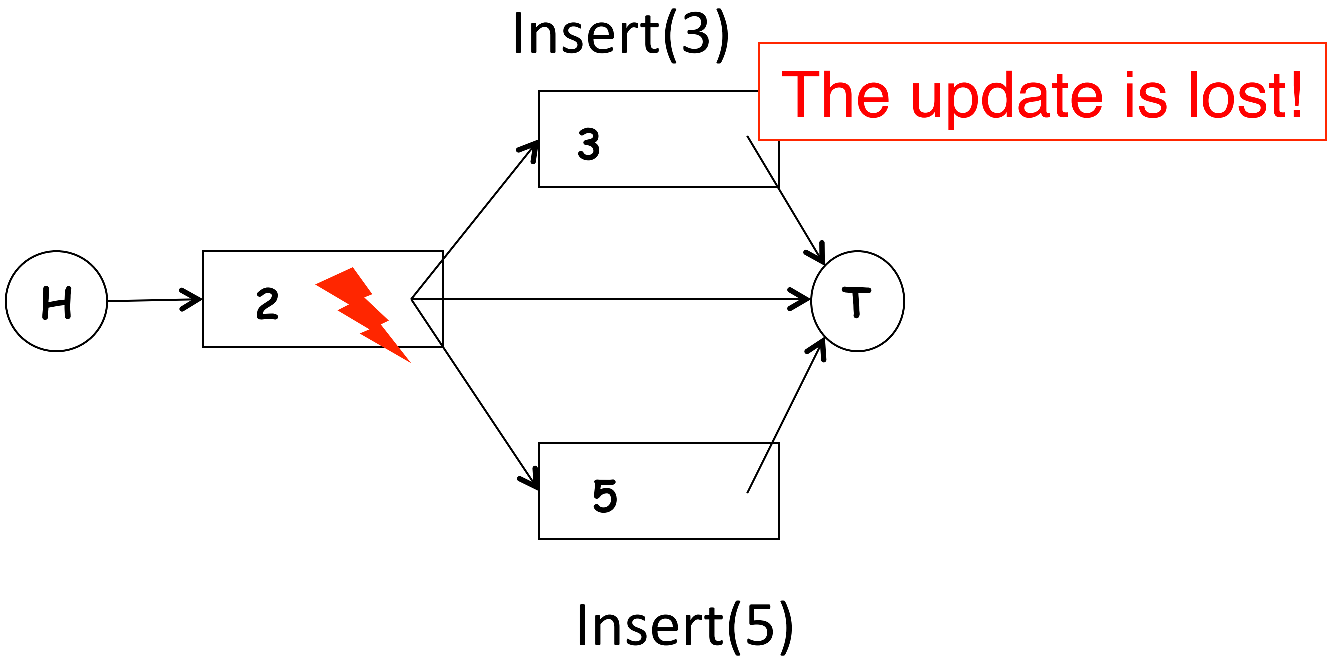
Linked-list for Set: sequential implementation

```
public boolean add(int item){
  Node pred=head;
       Node curr=head.next;
       while (curr.key < item){
      pred = curr;
      curr = pred.next;}
       if (curr.key==item){return
  false;}
       else {Node node = new
  Node(item);
      node.next=curr;
          pred.next=node;
          return true;}}
```

```
public boolean remove(int item){
  Node pred=head;
       Node curr=head.next;
       while (curr.key < item){
      pred = curr;
      curr = pred.next;}
       if (curr.key==item)
  {pred.next=curr.next; return
  true;}
       else {return false;}}
```

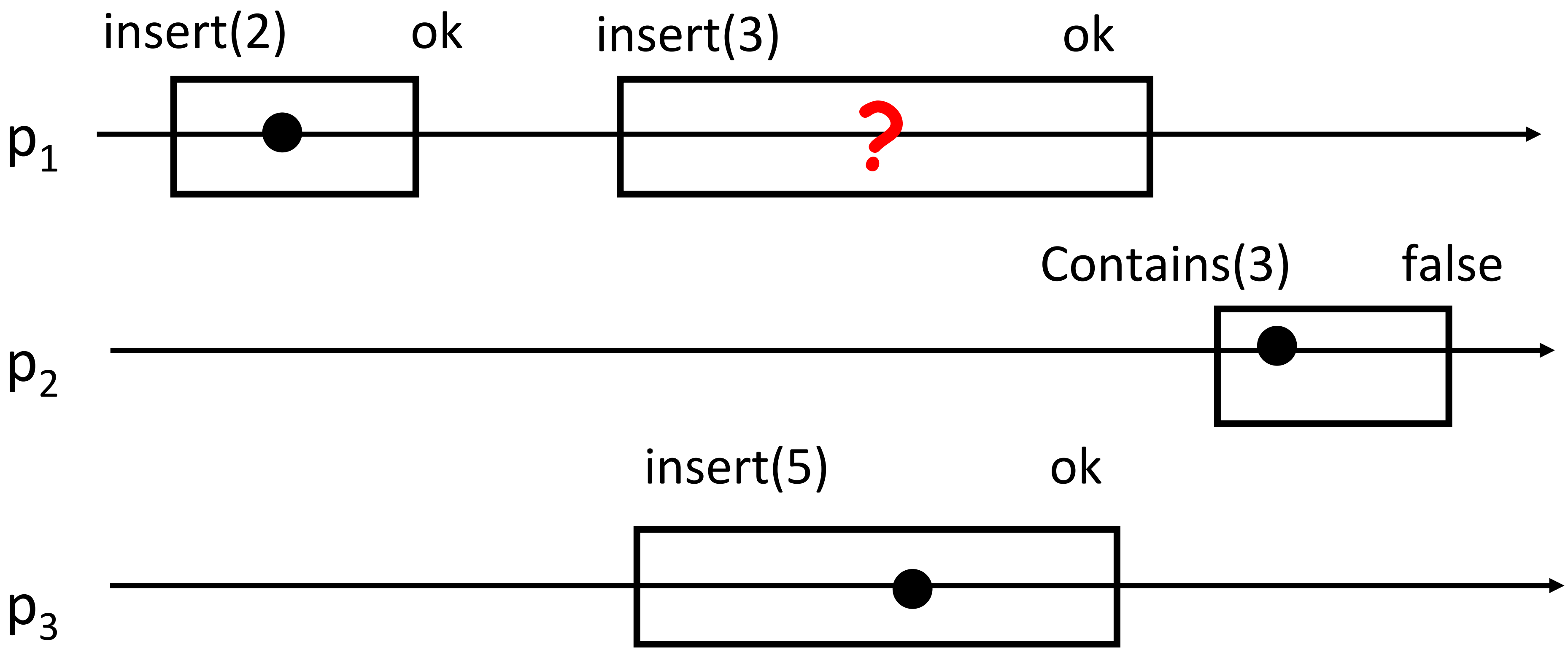
```
public boolean contains(int
   item){
   Node pred=head;
        Node curr=head.next;
        while (curr.key <
   item){
       pred = curr;
       curr = pred.next;}
        if (curr.key==item)
   {return true;}
        else {return
   false;}}
```

As is?



The extension with contains(3)

Is not linearizable!



Need to protect the list elements:

locks, transactional memory…

Concurrent reasoning?

- How to show that an implementation is correct (linearizable)?
- Invariants: *true* initially, no transition can render it *false*
  - ✓E.g., the object representation "makes sense"
- (Sorted) list-based sets:
  - ✓ *head* and *tail* are sentinels
  - ✓ nodes are sorted and keys are unique
  - ✓(the structure can be produced sequentially)

Progress guarantees?

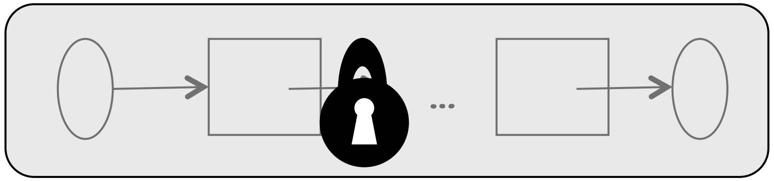
- Locks are used to protect list elements (assuming cooperation):
  - ✓Deadlock-freedom: at least one process makes progress (completes all its operations)
  - ✓Starvation-freedom: every process makes progress
- Nonblocking approaches:
  - ✓Wait-free: every operation completes in a finite number of steps
  - ✓Lock-free: some operation completes in a finite number of steps

Coarse grained solution

```
public class CoarseList{

    private Node head;
    private Lock lock = new ReentrantLock();



    public boolean add(int item){
      lock.lock();
      Node pred=head;
      try {
        Node curr=head.next;
        while (curr.key < item){
            pred = curr;
            curr = pred.next;
        }
        if (curr.key==item){return false;}
        Node node = new Node(item);
        node.next=curr;
        pred.next=node;
        return true;
      } finally{
      lock.unlock();
       }
    }
```

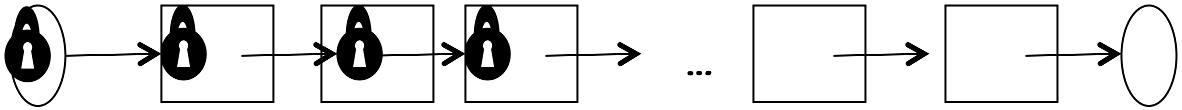
- Same progress guarantees as lock
  - ✓ ReentrantLock – starvation-free
- Good for low contention
- Sub-optimal for moderate to high contention: operations run sequentially

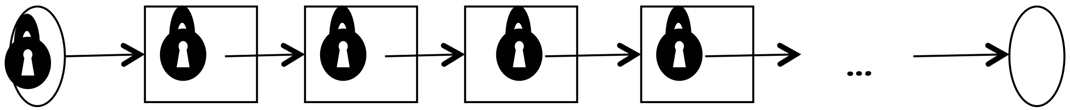
Locking schemes for a linked-list



Coarse-grained locking

2-phase locking





Hand-over-hand locking

Fine-grained solution: hand-over-hand

```
 public boolean add(int item){
   head.lock();
   Node pred=head;
   try {
     Node curr=head.next;
     curr.lock();
     try {
       while (curr.key < item){
             pred.unlock();
             pred = curr;
             curr = pred.next;
             curr.lock()
         }
        if (curr.key==item){
              return false;}
        Node node = new Node(item);
         node.next=curr;
        pred.next=node;
        return true;
     } finally{
          curr.unlock();
     }
  finally{
    pred.unlock();
  }
}
```

```
 public boolean remove(int item){
   head.lock();
   try {
     Node pred=head;
     Node curr=pred.next;
      curr.lock();
     try {
       while (curr.key < item){
             pred.unlock();
             pred = curr;
             curr = pred.next;
             curr.lock()
         }
        if (curr.key==item){
              pred.next=curr.next;
              return true;}
        return false;
     } finally{
          curr.unlock();
     }
  finally{
    pred.unlock();
  }
}
```

Hand-over-hand: correctness?

- Linearizability:
  - ✓An item is in the set **if and only if** it its node is reachable
  - ✓A successful add(item) linearizes when the highest node was locked (the last `curr.lock()`)
  - ✓A successful add(item) linearizes when the highest predecessor node was locked (the last `pred.lock()`)
- Progress (starvation-freedom)
  - ✓All operations acquire the locks in the order of growing items
  - ✓No deadlock possible

Hand-over-hand: concurrency limitations

```
 public boolean add(int item){
   head.lock();
   Node pred=head;
   try {
     Node curr=head.next;
     curr.lock();
     try {
       while (curr.key < item){
             pred.unlock();
             pred = curr;
             curr = pred.next;
             curr.lock()
         }
        if (curr.key==item){
              return false;}
        Node node = new Node(item);
         node.next=curr;
        pred.next=node;
        return true;
      } finally{
           curr.unlock();
      }
  finally{
     pred.unlock();
  }
}
```

- More concurrency:
  - ✓ An operation working on a "high" node does not obstruct ones working on "low" nodes
- But! Operations concerning disjoint nodes may obstruct each other
  - ✓ E.g. add(3) and add (9) applied to {1,5,7}
- Optimistic algorithm?
  - ✓ No locks on the traverse path

Quiz 1: hand-over-hand

- How to linearize unsuccessful `add` and `remove`?
- How to linearize `contains`?
- Prove starvation-freedom (assuming starvation-free locks)

Optimistic solution: no locks on traversal and validation

```
private boolean validate(Node pred, Node
   curr) {
    Node node=head;
    while (node.key <= pred.key){
          if (node==pred){
             return pred.next==curr;}
          node=node.next;
    }
    return false;
  }
```

```
public boolean remove(int item)
   while (true){
      Node pred=head;
      Node curr=pred.next;
      while (curr.key<item){
         pred=curr;
         curr=curr.next;
      }
      pred.lock(); curr.lock();
      try {
        if (validate(pred,curr)){
          if (curr.key==item) {

           pred.next=curr.next;
               return true;
           }
          return false; }
      } finally{
         pred.unlock();
         curr.unlock();
      }
}
```

Is validation necessary for updates?

Is it necessary for contains?

Optimistic solution: no locks on traversal and validation

```
public boolean add(int item){
   while (true){
      Node pred=head;
      Node curr=pred.next;
      while (curr.key<item){
         pred=curr;
         curr=curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr)){
           if (curr.key==item) {
             return false;
            }
           Node node = new Node(item);
           node.next=curr;
           pred.next=node;
           return true; }
      } finally{
         pred.unlock();
         curr.unlock();}
   }
}
```

```
public boolean contains(int item) {
   while (true){
      Node pred=head;
      Node curr=pred.next;
      while (curr.key<item){
         pred=curr;
         curr=curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr)){
            return (curr.key==item);
            }
      } finally{
         pred.unlock();
         curr.unlock();}
      }
   }
}
```

- contains grabs locks
- updates re-traverse even if no contention.

Quiz 2: optimistic

- Show that validation is necessary
  - ✓Hint: consider an algorithm without validation and show that an update can get lost because of a series of concurrent removes
- Is validation necessary for contains?
- Show that the algorithm is not starvation-free (even if all locks are)

Lazy synchronization: logical removals and wait-free contains

```
private boolean validate(Node pred, Node
    curr) {

  return !pred.marked && !curr.marked &&
             pred.next==curr;
}
```

- remove first marks the node for deletion and then physically removes it
- contains returns true iff the node is reachable and not marked
- A node is in the set iff it is an unmarked reachable node

```
public boolean remove(int item)
   while (true){
      Node pred=head;
      Node curr=pred.next;
       while (curr.key<item){
         pred=curr;
          curr=curr.next;
       }
      pred.lock();
       try {
          curr.lock();
          try {
           if (validate(pred,curr)){
               return false;}
           curr.marked=true;
           pred.next=curr.next;
           return true;
          } finally{
               curr.unlock(); }
      } finally{
         pred.unlock();}
   }
}
```

Lazy synchronization: wait-free contains

```
public boolean add(int item){
   while (true){
      Node pred=head;
      Node curr=pred.next;
       while (curr.key<item){
         pred=curr;
          curr=curr.next;
       }
      pred.lock();
       try {
          curr.lock();
          try {
           if (validate(pred,curr)){
            if (curr.key==item) {
               return false;
              }
           Node node = new Node(item);
           node.next=curr;
           pred.next=node;
           return true;
          } finally{
             curr.unlock(); }
       } finally{
         pred.unlock();}
     } }
}
```

```
public boolean contains(int item){

   Node curr=head;
    while (curr.key<item){
       curr=curr.next;
    }
    return (curr.key==item)&& !curr.marked ;
}
```

Quiz 3: lazy

- Show that both conditions in the validation check are necessary

  Hint: consider concurrent removes on two consecutive nodes, or a remove concurrent to an add of a preceding node

- Determine linearization points for all operations:
  - ✓ add (successful or not)
  - ✓ remove (successful or not)
  - ✓ contains (successful or not)

  Hint: for an unsuccessful contains(x), linearization point may vary depending on the presence of a concurrent add(x)

From locks to nonblocking

- Lazy [Heller et al.]: best of the class?
  - ✓contains wait-free
  - ✓add and remove are only deadlock-free
- Can we turn the methods to lock-free?
  - ✓Wait-free for contains
- Replace read and update of curr.next with CAS?
  - ✓Not that easy: may need to atomically update both the marked field and the reference
  - ✓AtomicMarkableReference in java…
- Herlihy and Shavit, Chapter 9.8

Conventional synchronization

- Locks are hard to use efficiently
- Nonblocking implementations with CAS have inherent (hardware) limitations
- Multiple operations cannot be easily composed

What can we do about it?

Transactions?

```
public class TxnList{

    private Node head;


    public boolean add(int item){
     atomic {
      Node pred=head;
      Node curr=head.next;
      while (curr.key < item){
            pred = curr;
            curr = pred.next;
      }
      if (curr.key==item){return false;}
      Node node = new Node(item);
      node.next=curr;
      pred.next=node;
      return true;
      }
     }
```

Transactional memory

- A transaction atomic {…} commits or aborts
- Committed transactions serialize:
  - ✓ Constitute a sequential execution
- Aborted transactions "never happened"
  - ✓ Can affect other aborted ones?
- A correct sequential program implies a correct concurrent one
- Composition is easy:

```
atomic{
      x=q0.deq();
      q1.enq(x);
}
```

So what is better?

It depends

on:

- the data structure (some are more concurrency-friendly than others)
- workload (high update-rate vs. read-dominated)
- Programming skills ☺
- TM inherent costs (more to come on this)

- Next time: list-based sets in java
  - ✓What is better on what workload
  - ✓SynchroBench: https://github.com/gramoli/synchrobench
- The list of papers (with pdfs) and the link to a form to submit your choice:
  - ✓http://perso.telecom-paristech.fr/~kuznetso/INF346-2015/
  - ✓By March 27, 2015