



Transactional Boosting:

A Methodology for
Highly Concurrent Transactional Objects

Eric Koskinen
Brown University

Joint work with

Maurice Herlihy
Brown University

Everyone loves STM

Everyone loves STM

- Locking doesn't scale

Everyone loves STM

- Locking doesn't scale
- Replace locking with software transactions

Everyone loves STM

- Locking doesn't scale
- Replace locking with software transactions

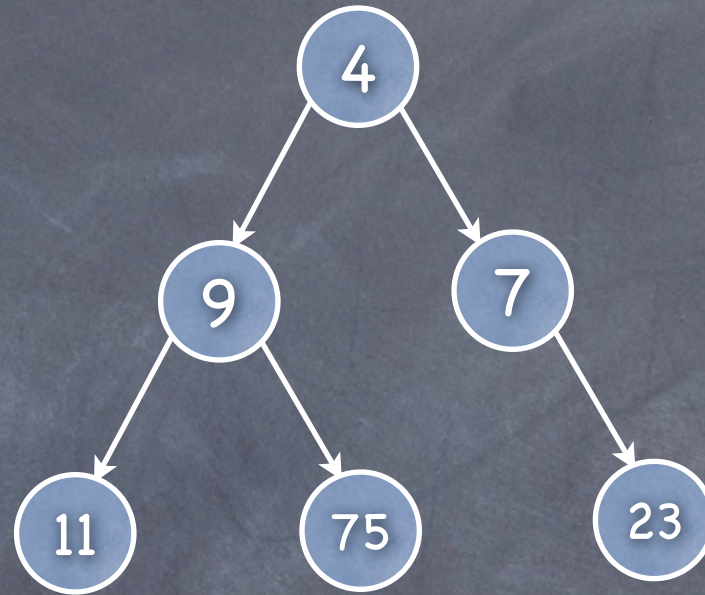
Everyone loves STM

- Locking doesn't scale
- Replace locking with software transactions
- **But** numerous issues remain.

Everyone loves STM

- Locking doesn't scale
- Replace locking with software transactions
- **But** numerous issues remain.
- Here's one ...

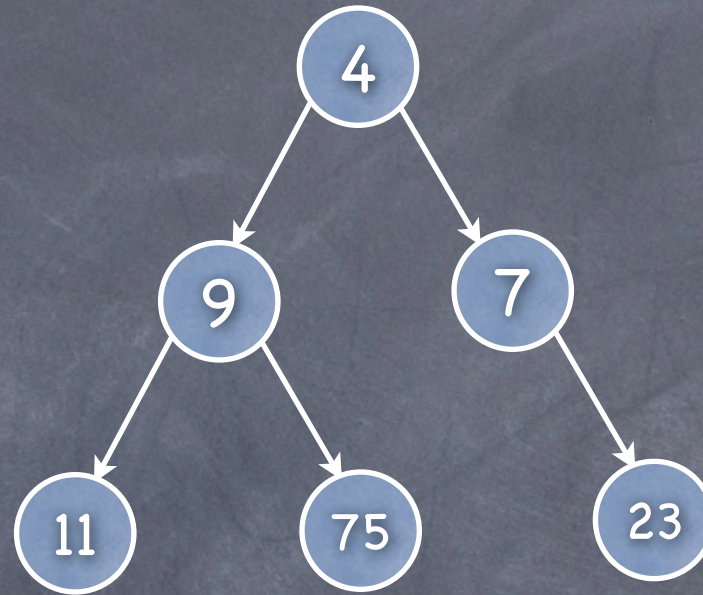
Concurrent Skew Heap



Concurrent Skew Heap

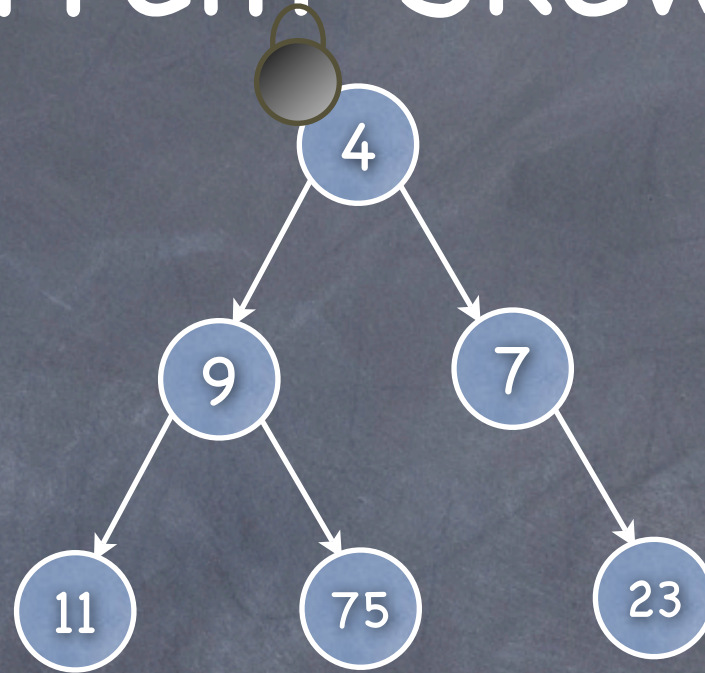
Thread A

insert(91)



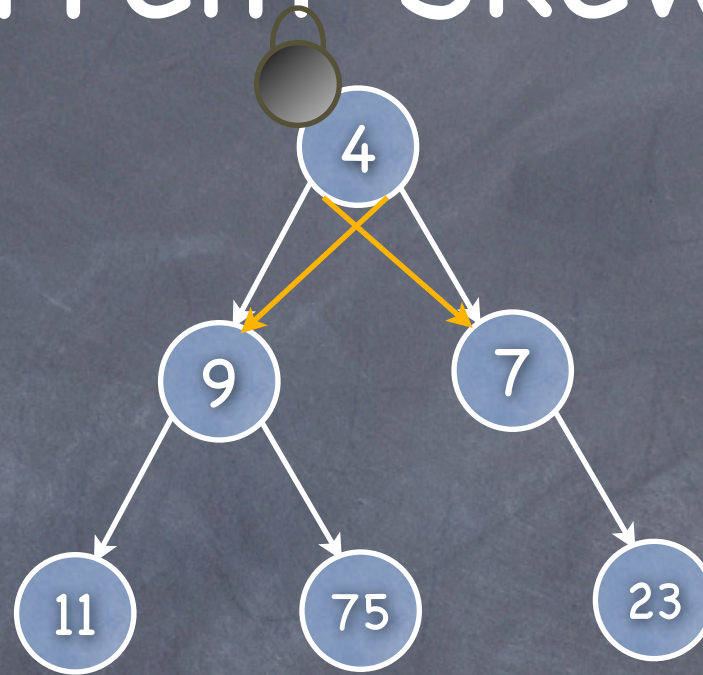
Concurrent Skew Heap

Thread A
insert(91)



Concurrent Skew Heap

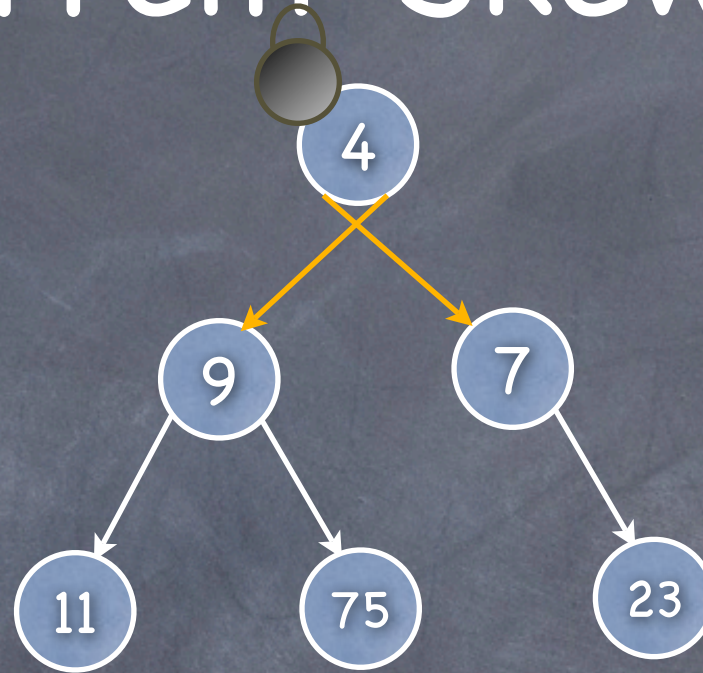
Thread A
insert(91)



Concurrent Skew Heap

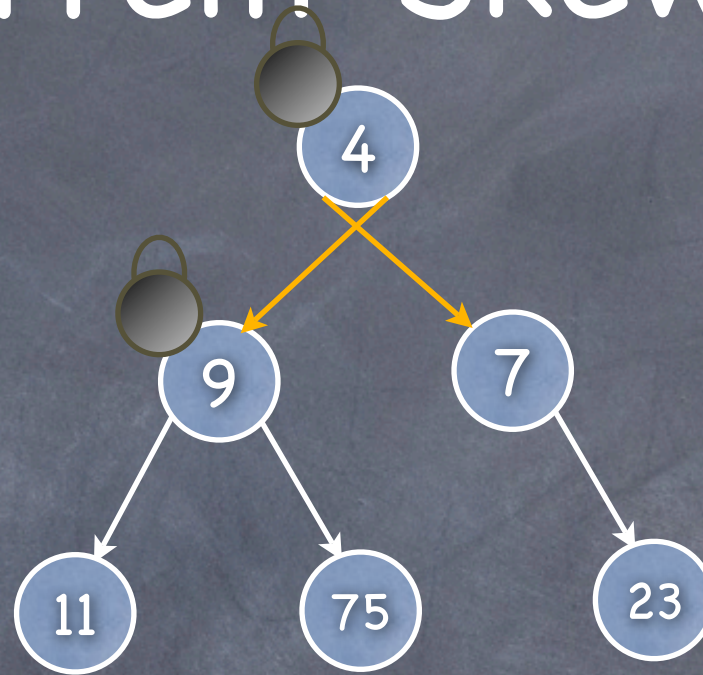
Thread A

insert(91)



Concurrent Skew Heap

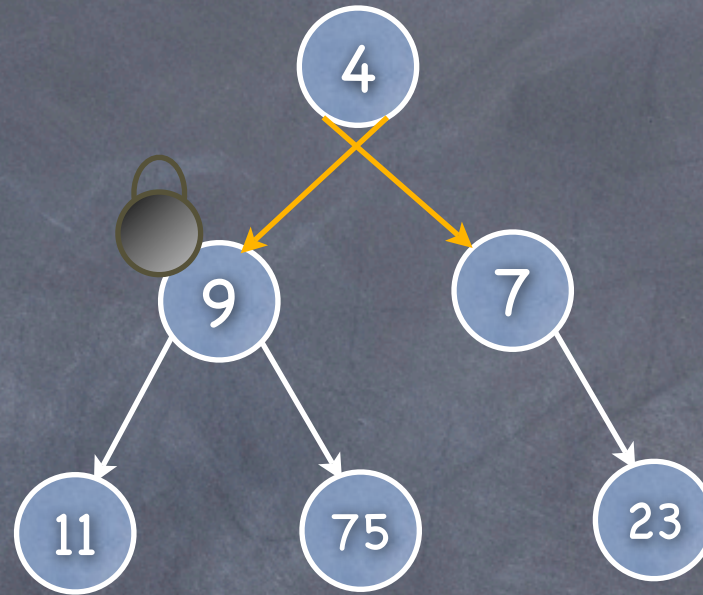
Thread A
insert(91)



Concurrent Skew Heap

Thread A

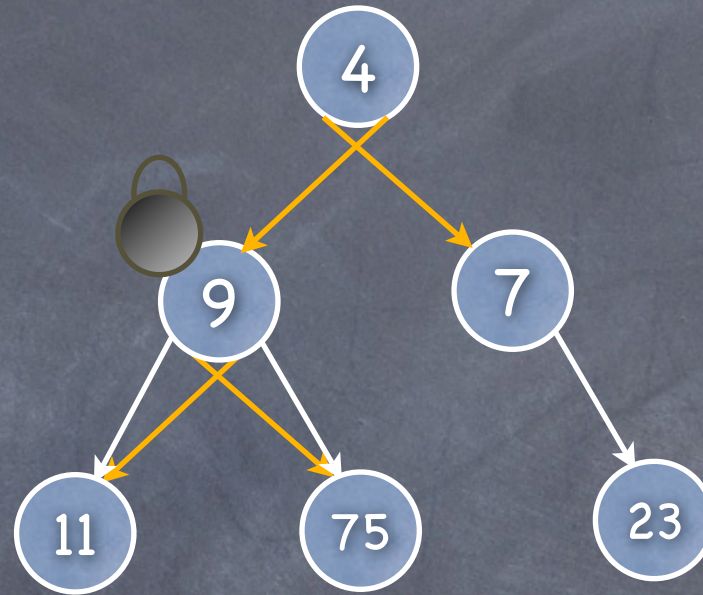
insert(91)



Concurrent Skew Heap

Thread A

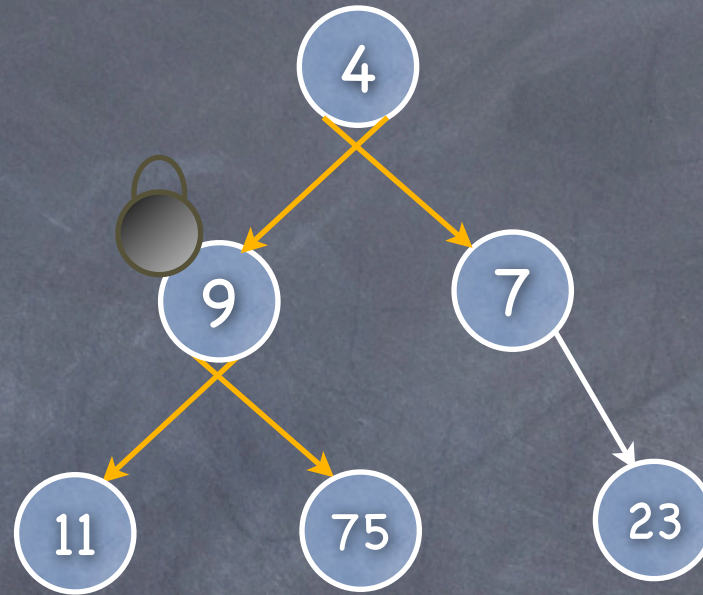
insert(91)



Concurrent Skew Heap

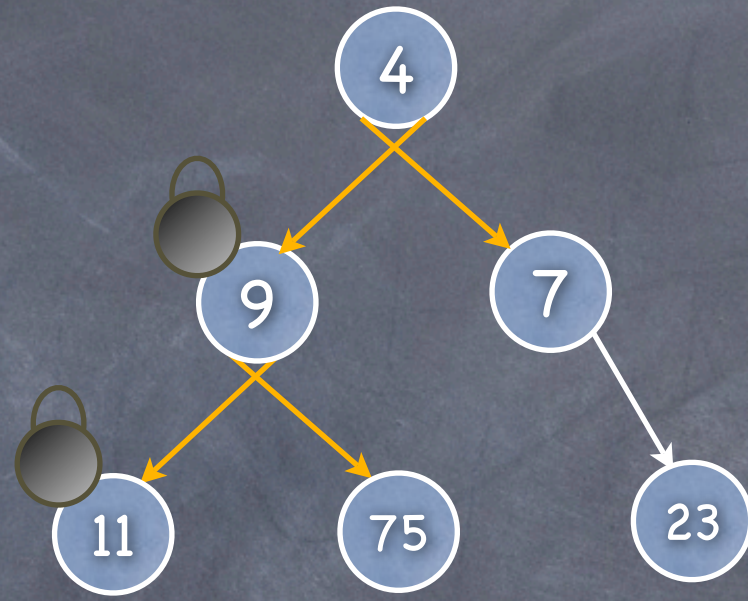
Thread A

insert(91)



Concurrent Skew Heap

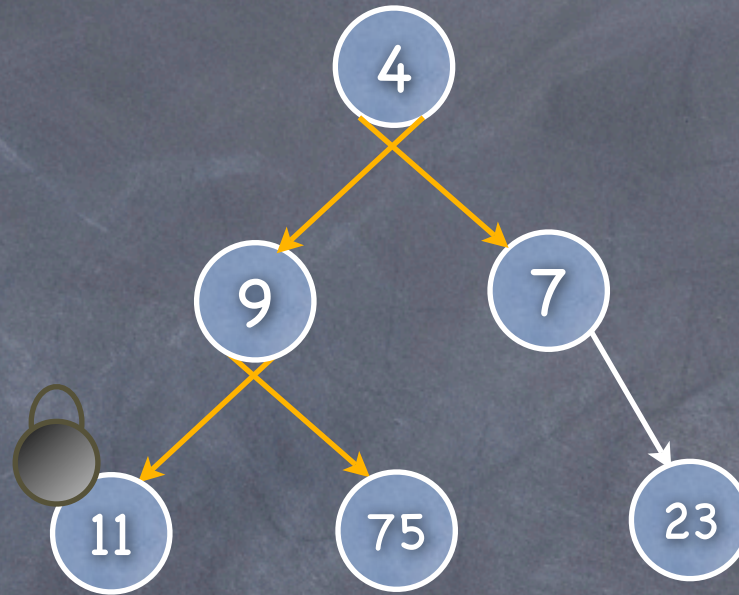
Thread A
insert(91)



Concurrent Skew Heap

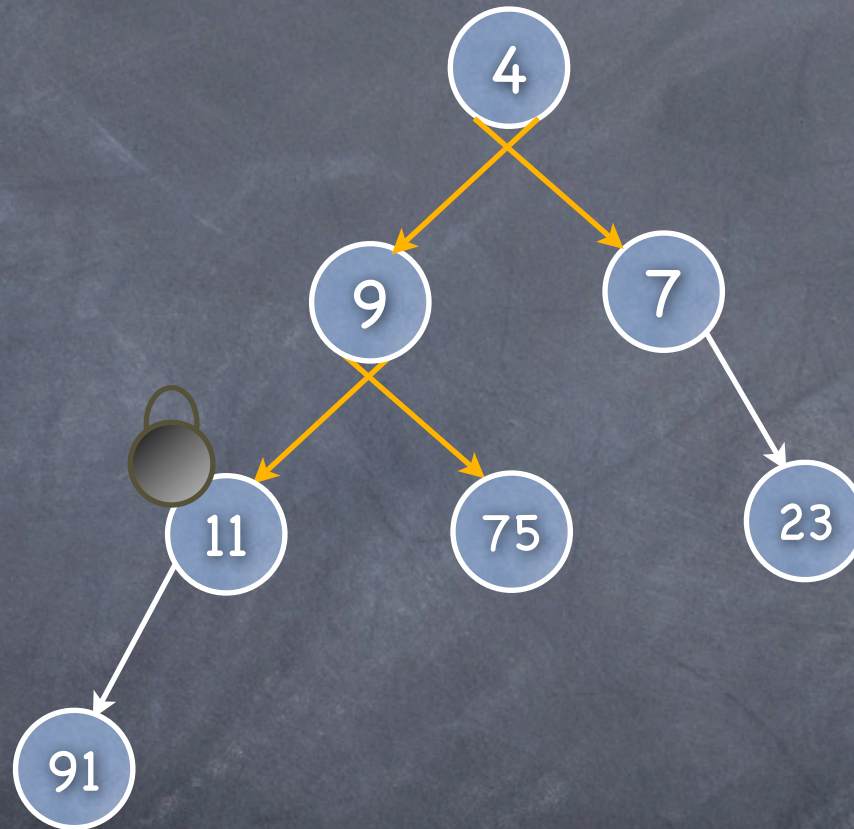
Thread A

insert(91)



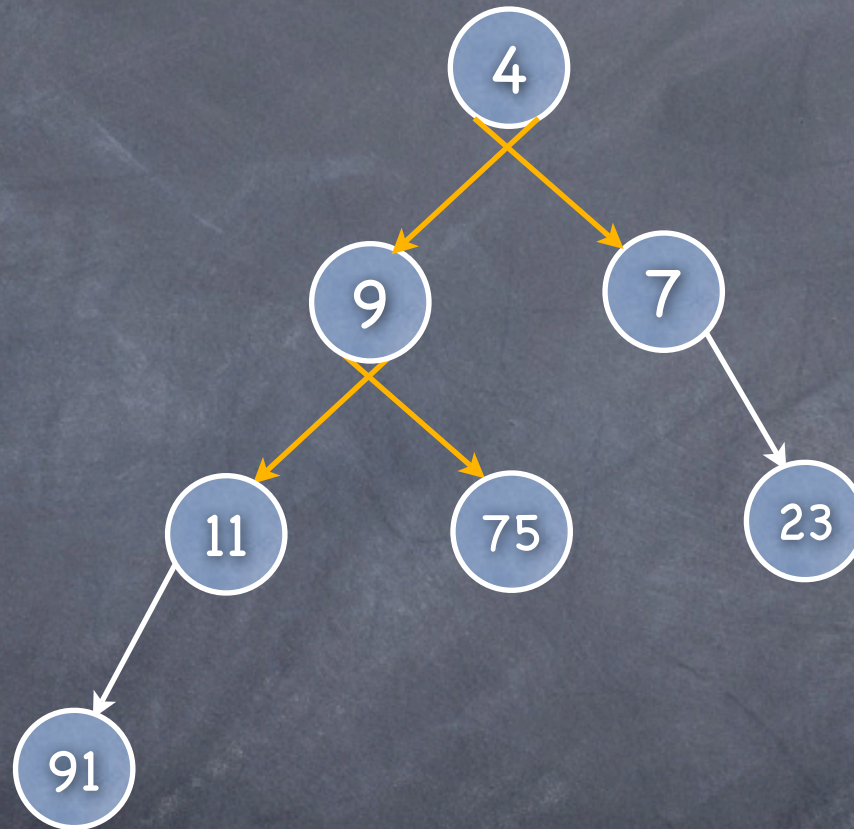
Concurrent Skew Heap

Thread A
insert(91)



Concurrent Skew Heap

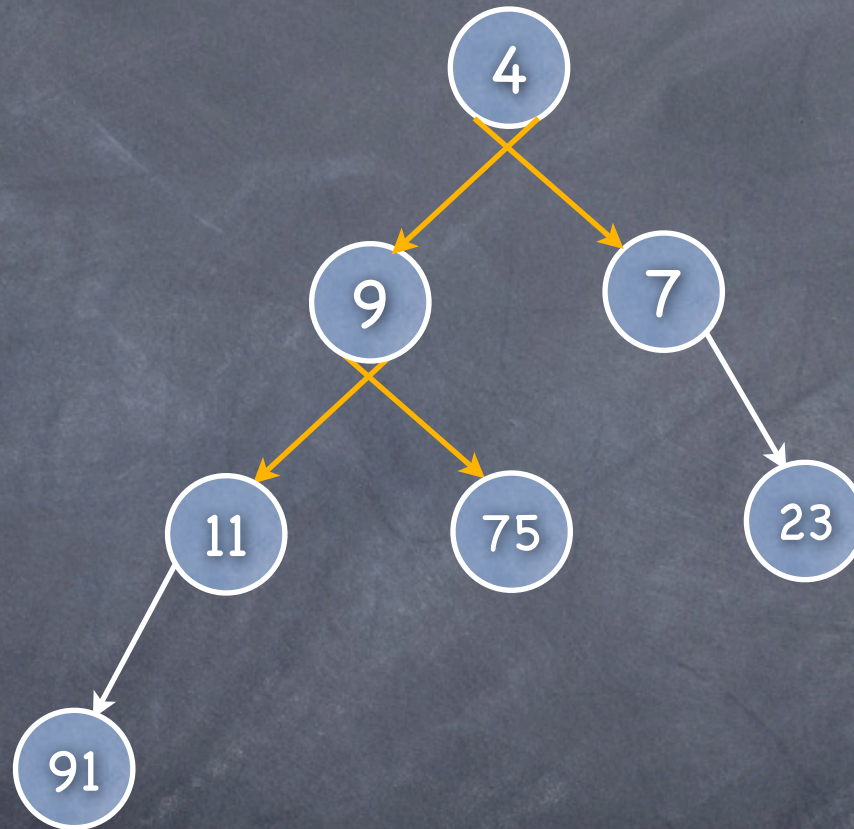
Thread A
insert(91)



Concurrent Skew Heap

Thread A
insert(91)

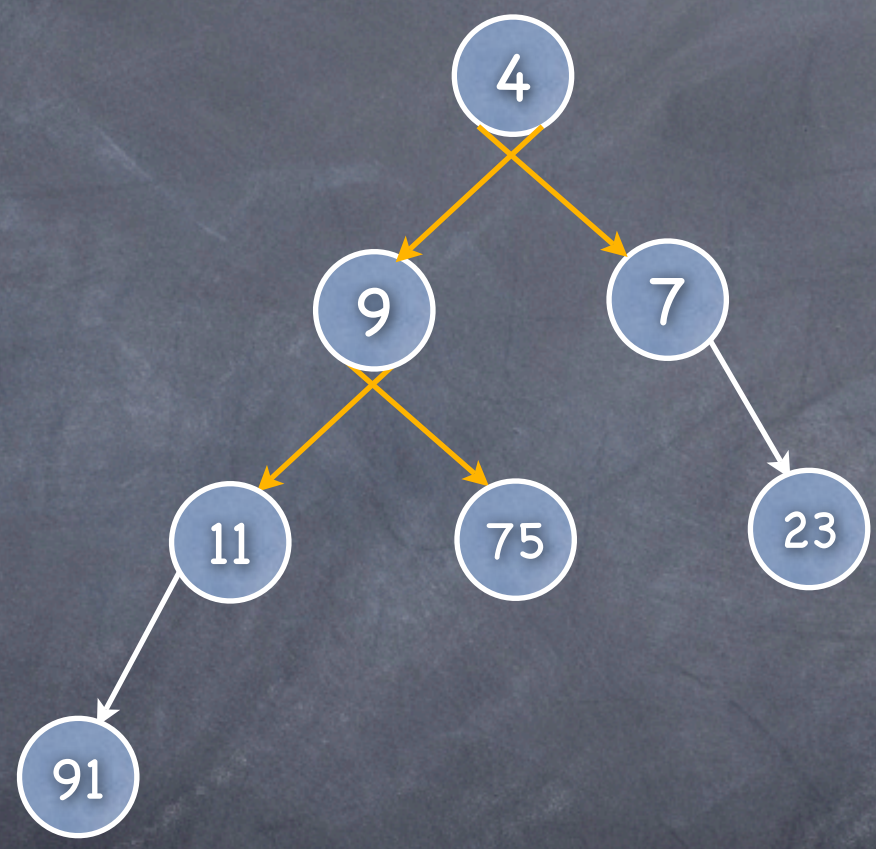
Thread B
removeMin()



Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

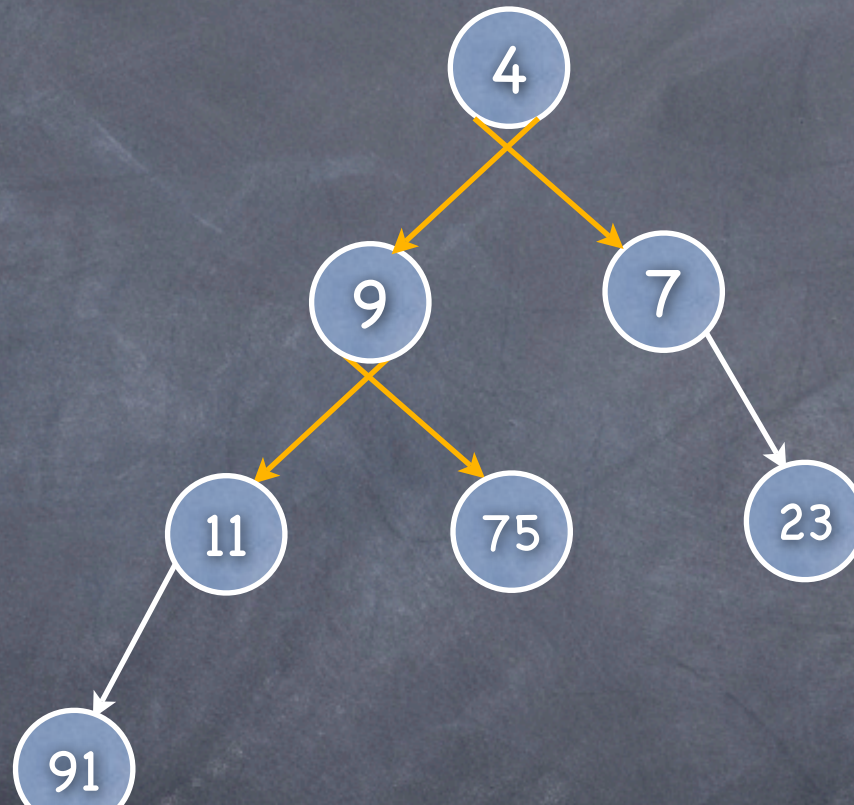


Now, in STM:

Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

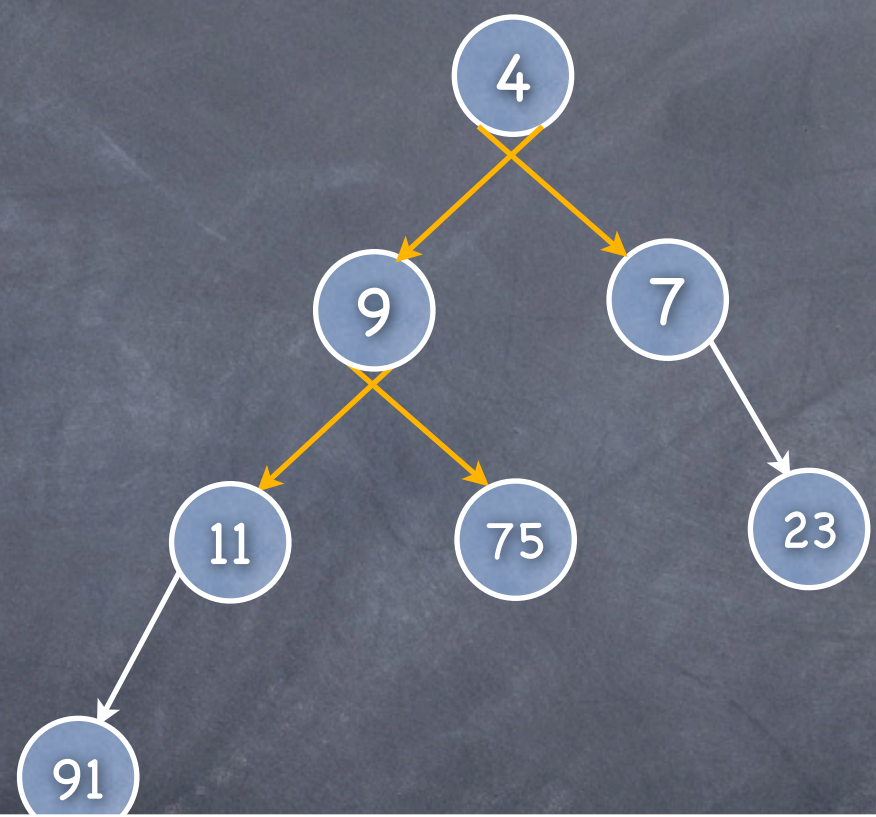


Now, in STM:

Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()



Now, in STM:

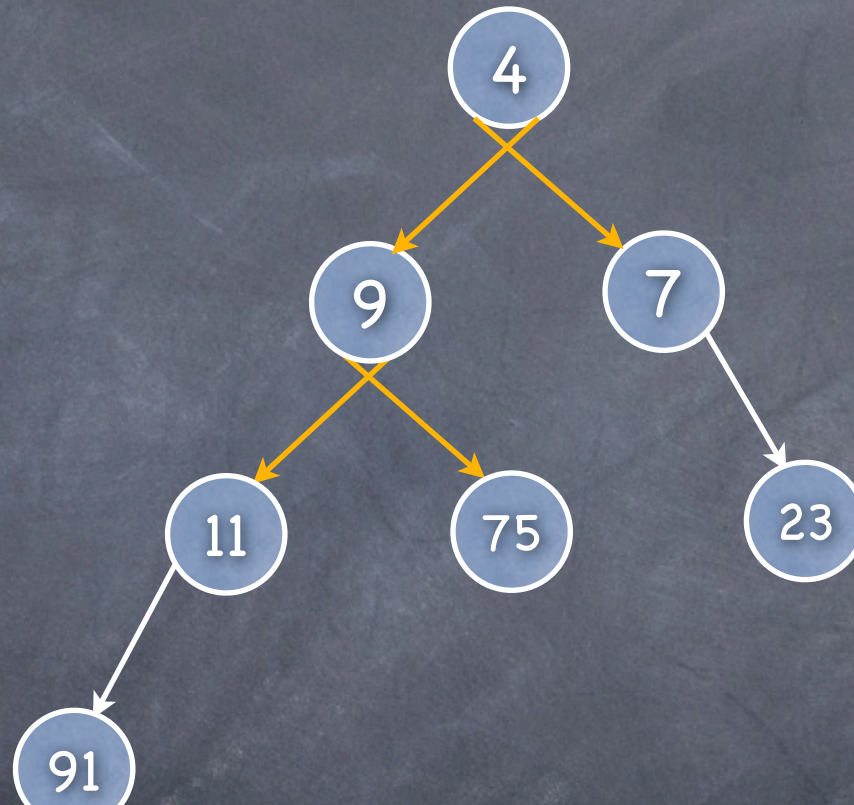
```
atomic {  
  ...  
  tmp = root.left;  
  root.left := root.right;  
  root.right := tmp;  
  ...  
}
```

```
atomic {  
  ...  
  ...  
  a := root.left;  
  b := root.right;  
  ...  
}
```

Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()



Now, in STM:

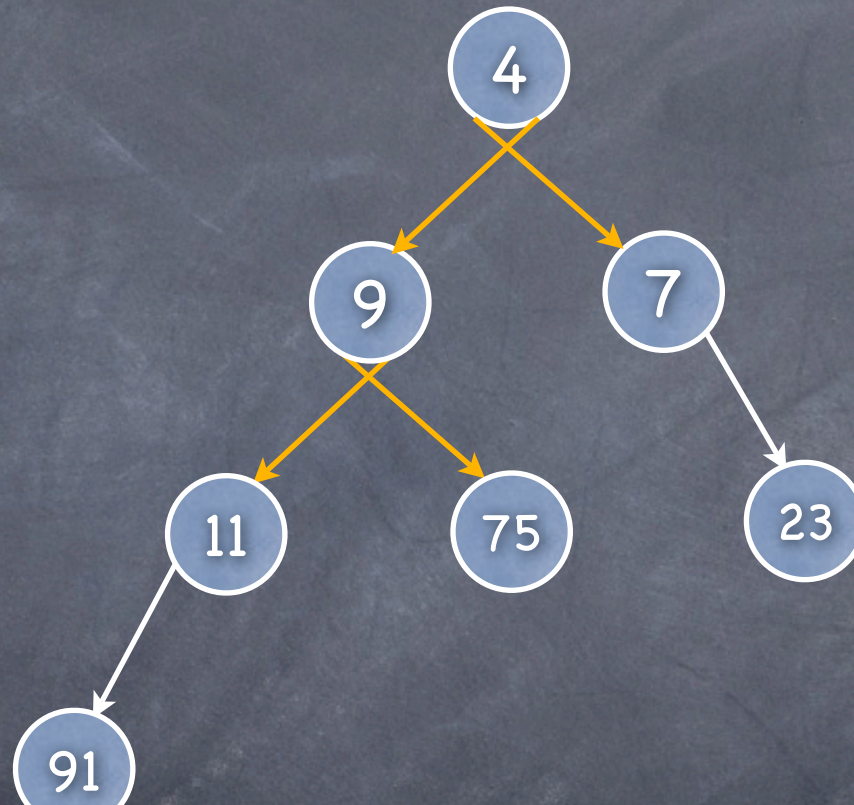
```
atomic {  
  ...  
  tmp = root.left;  
  root.left := root.right;  
  root.right := tmp;  
  ...  
}
```

```
atomic {  
  ...  
  a := root.left;  
  b := root.right;  
  ...  
}
```

Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()



```
atomic {  
  ...  
  tmp = root.left;  
  root.left := root.right;  
  root.right := tmp;  
  ...  
}
```

```
atomic {  
  ...  
  ...  
  root := root.left;  
  root := root.right;  
  ...  
}
```

false conflict!!!

The Problem

- Highly concurrent objects
- Become sequential in a transaction

The Problem

- Highly concurrent objects
- Become sequential in a transaction

The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:

The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:

- False conflicts

The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:

- False conflicts
- No distinction between **thread** synchronization and **transactional** synchronization

Transactional Boosting

- Black-box Linearizable base object
- Given semantics: commutative methods and inverse methods
- Solve **synchronization** and **recovery** without tracking read/write sets
- Reuse highly concurrent **linearizable** objects as highly concurrent **transactional** objects

Transactional Boosting

- Black-box Linearizable b
- Given semantics: commut
- inverse methods
- Solve **synchronization** and **recovery** without tracking read/write sets
- Reuse highly concurrent **linearizable** objects as highly concurrent **transactional** objects

Highly concurrent
transactional objects
for free!

Simple Example: A Set



(Implemented as a linked list)

Methods

add(x)/bool

remove(x)/bool

contains(x)/bool

Simple Example: A Set

Method	Inverse
--------	---------



Simple Example: A Set



Method	Inverse
$\text{add}(x)/\text{false}$	$\text{noop}()$

Simple Example: A Set



Method	Inverse
$\text{add}(x)/\text{false}$	$\text{noop}()$
$\text{add}(x)/\text{true}$	$\text{remove}(x)/\text{true}$

Simple Example: A Set



Method	Inverse
$\text{add}(x)/\text{false}$	$\text{noop}()$
$\text{add}(x)/\text{true}$	$\text{remove}(x)/\text{true}$
$\text{remove}(x)/\text{false}$	$\text{noop}()$

Simple Example: A Set



Method	Inverse
$\text{add}(x)/\text{false}$	$\text{noop}()$
$\text{add}(x)/\text{true}$	$\text{remove}(x)/\text{true}$
$\text{remove}(x)/\text{false}$	$\text{noop}()$
$\text{remove}(x)/\text{true}$	$\text{add}(x)/\text{true}$

Simple Example: A Set



Method	Inverse
$\text{add}(x)/\text{false}$	$\text{noop}()$
$\text{add}(x)/\text{true}$	$\text{remove}(x)/\text{true}$
$\text{remove}(x)/\text{false}$	$\text{noop}()$
$\text{remove}(x)/\text{true}$	$\text{add}(x)/\text{true}$
$\text{contains}(x)/_$	$\text{noop}()$

Simple Example: A Set

Commutativity



Simple Example: A Set



Commutativity

$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

Simple Example: A Set



Commutativity

$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

$$\text{remove}(x)/_ \Leftrightarrow \text{remove}(y)/_ \quad x \neq y$$

Simple Example: A Set



Commutativity

$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

$$\text{remove}(x)/_ \Leftrightarrow \text{remove}(y)/_ \quad x \neq y$$

$$\text{contains}(x)/_ \Leftrightarrow \text{contains}(y)/_ \quad x \neq y$$

Simple Example: A Set



all commute
with each other

Commutativity

$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

$$\text{remove}(x)/_ \Leftrightarrow \text{remove}(y)/_ \quad x \neq y$$

$$\text{contains}(x)/_ \Leftrightarrow \text{contains}(y)/_ \quad x \neq y$$

Simple Example: A Set



all commute
with each other

Commutativity

$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

$$\text{remove}(x)/_ \Leftrightarrow \text{remove}(y)/_ \quad x \neq y$$

$$\text{contains}(x)/_ \Leftrightarrow \text{contains}(y)/_ \quad \cancel{x \neq y}$$

Simple Example: A Set



all commute
with each other

Commutativity

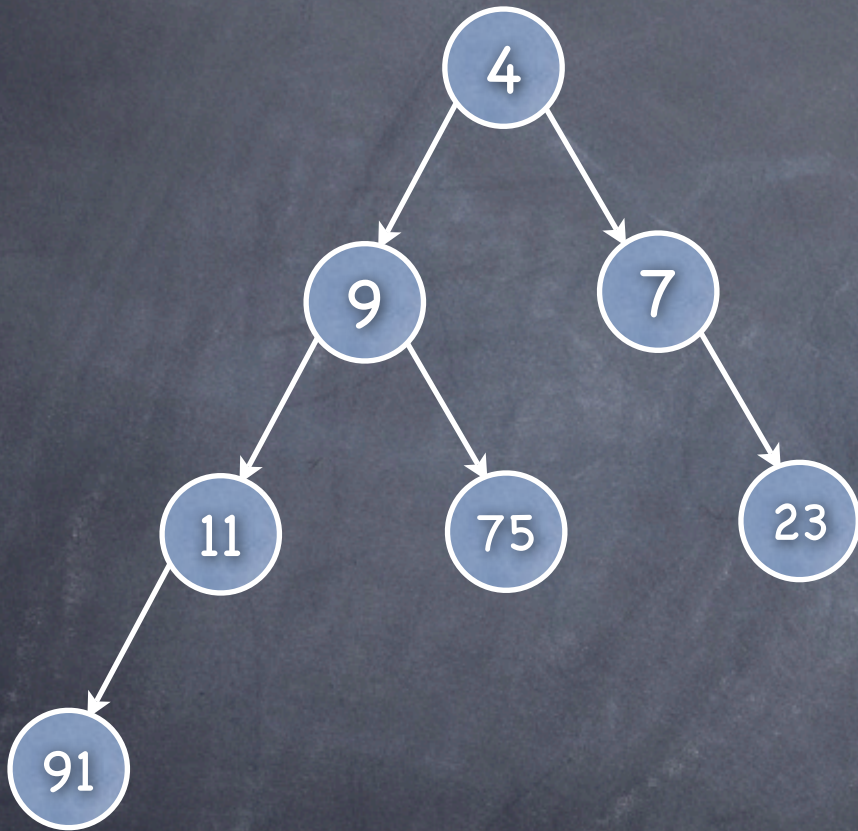
$$\text{add}(x)/_ \Leftrightarrow \text{add}(y)/_ \quad x \neq y$$

$$\text{remove}(x)/_ \Leftrightarrow \text{remove}(y)/_ \quad x \neq y$$

$$\text{contains}(x)/_ \Leftrightarrow \text{contains}(y)/_ \quad \cancel{x \neq y}$$

$$\text{add}(x)/\text{false} \Leftrightarrow \text{remove}(x)/\text{false} \Leftrightarrow \text{contains}(x)/_$$

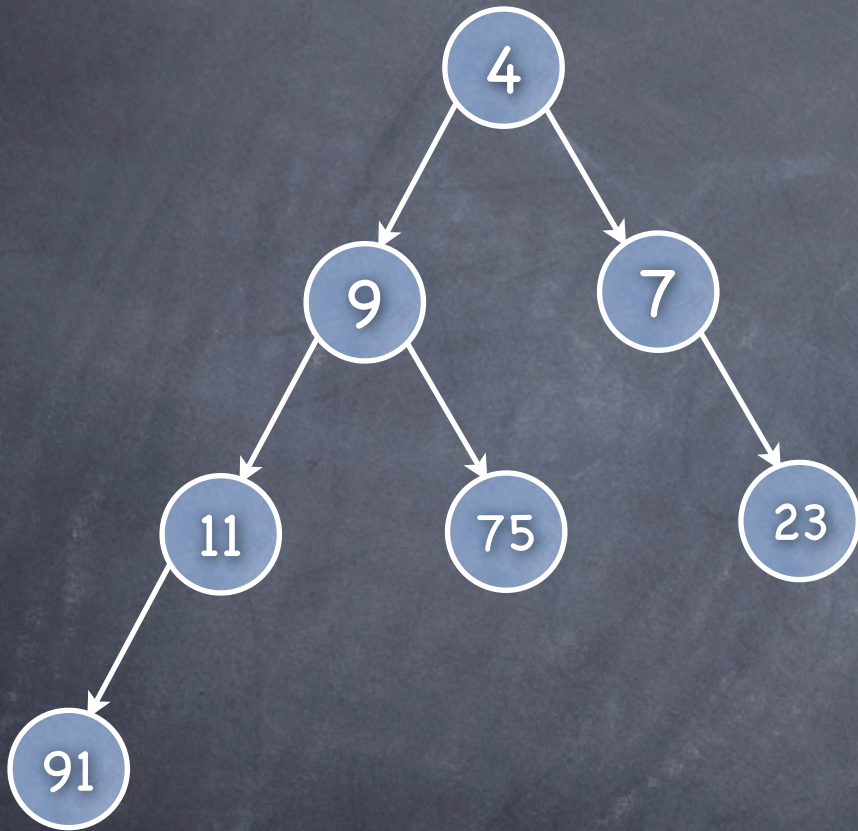
Example 2: Priority Queue



Methods
$\text{removeMin}(x)$
$\text{min}(x)$
$\text{insert}(x)$

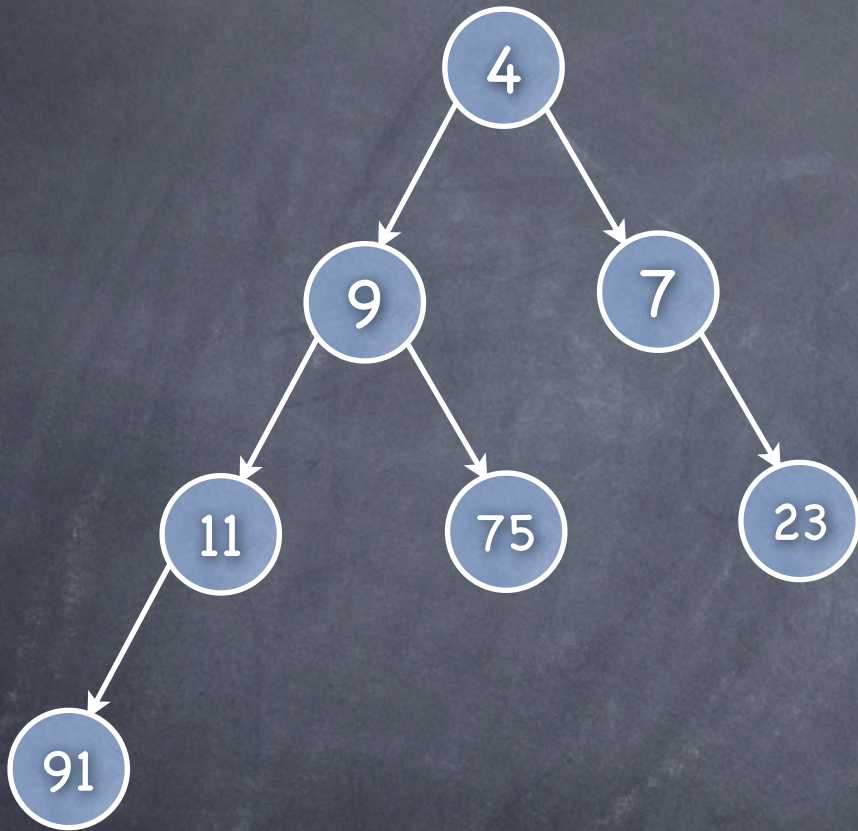
(Implemented as a Skew Heap)

Example 2: Priority Queue



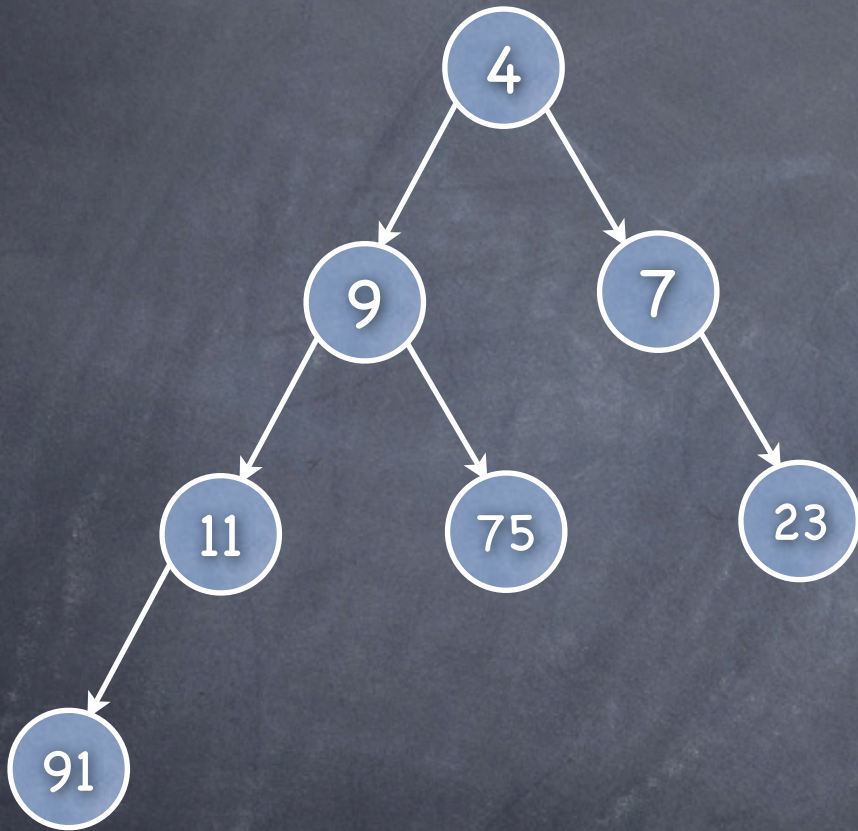
Method	Inverse
<code>removeMin()</code> / <code>x</code>	<code>insert(x)</code>

Example 2: Priority Queue



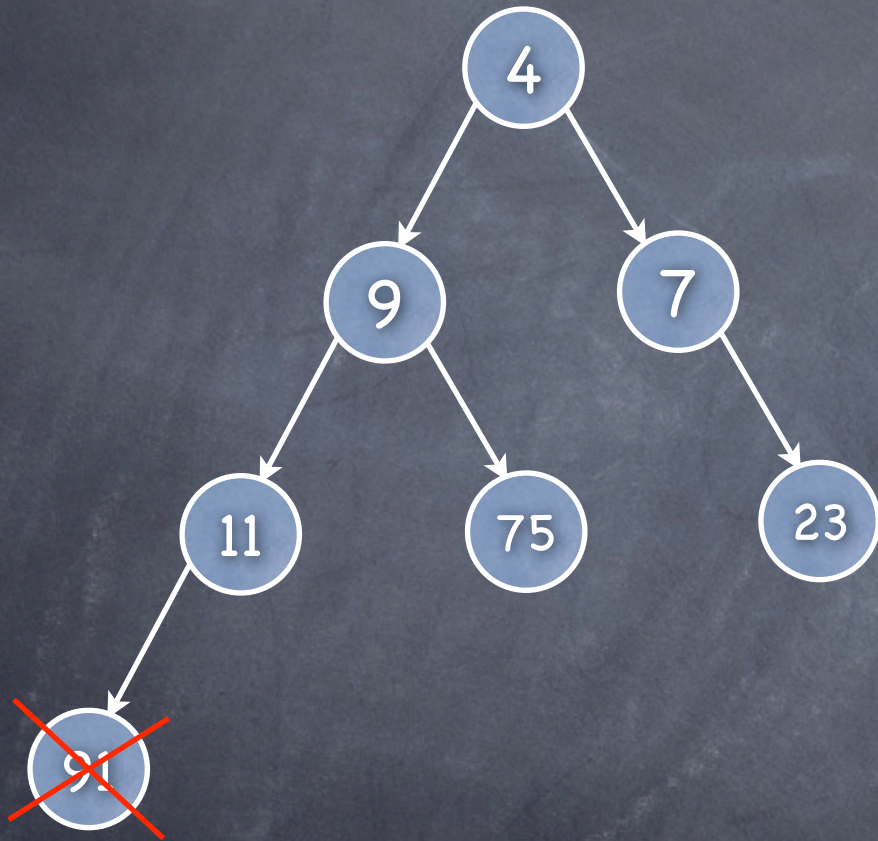
Method	Inverse
removeMin \emptyset/x	insert (x)
min \emptyset/x	noop \emptyset

Example 2: Priority Queue



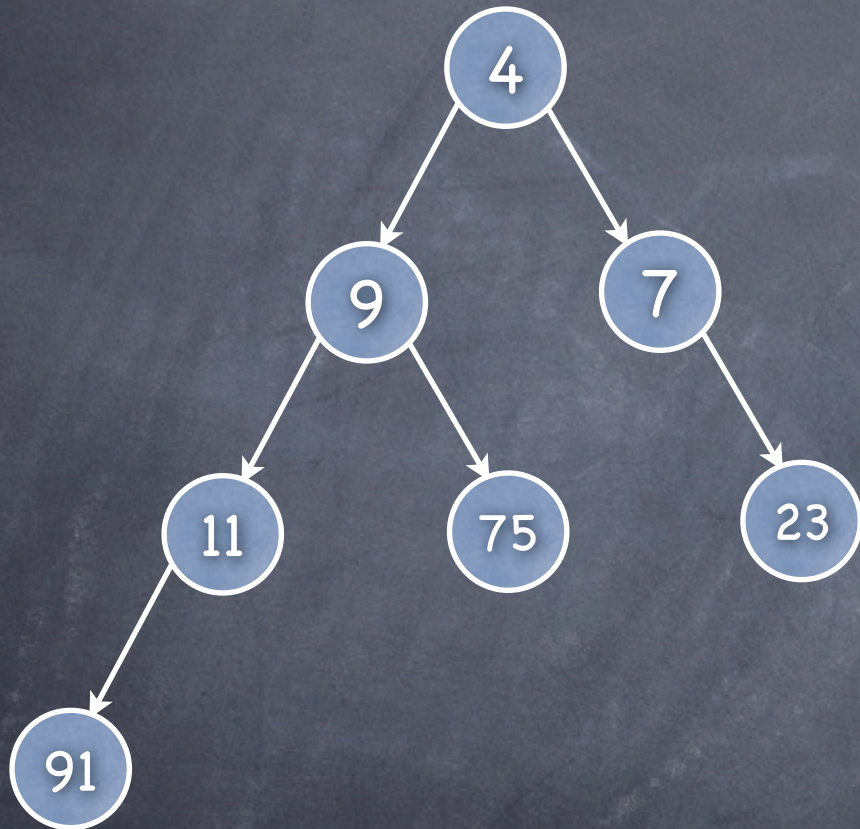
Method	Inverse
<code>removeMin()</code> / <code>x</code>	<code>insert(x)</code>
<code>min()</code> / <code>x</code>	<code>noop()</code>
<code>insert(x)</code>	<code>insertInv(x)</code>

Example 2: Priority Queue



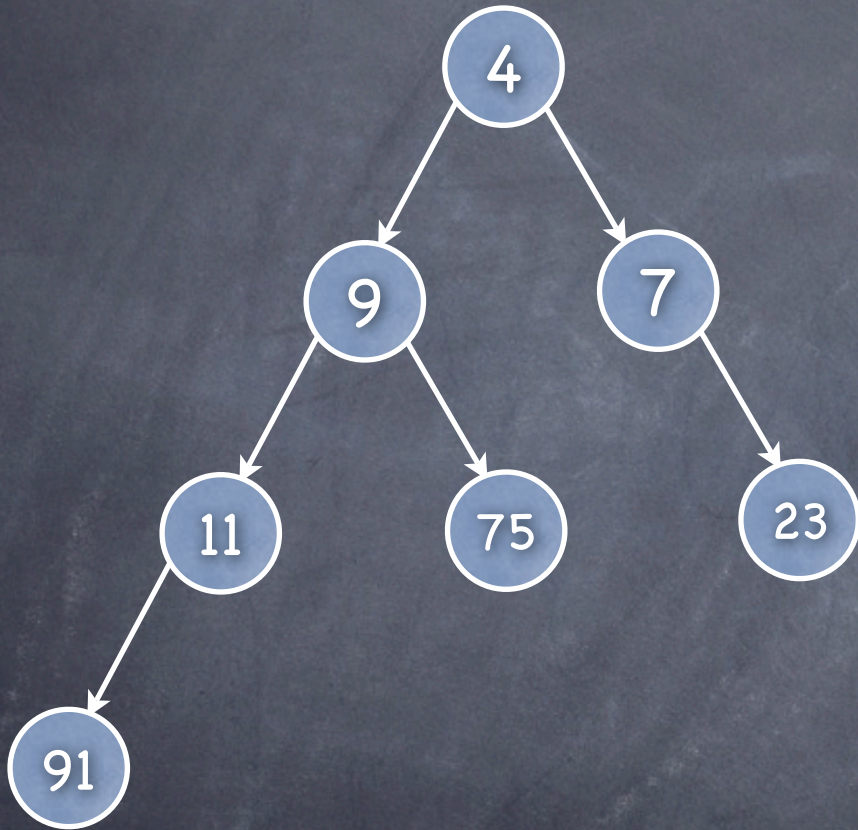
Method	Inverse
<code>removeMin()</code> / x	<code>insert(x)</code>
<code>min()</code> / x	<code>noop()</code>
<code>insert(x)</code>	<code>insertInv(x)</code>

Example 2: Priority Queue



Commutativity

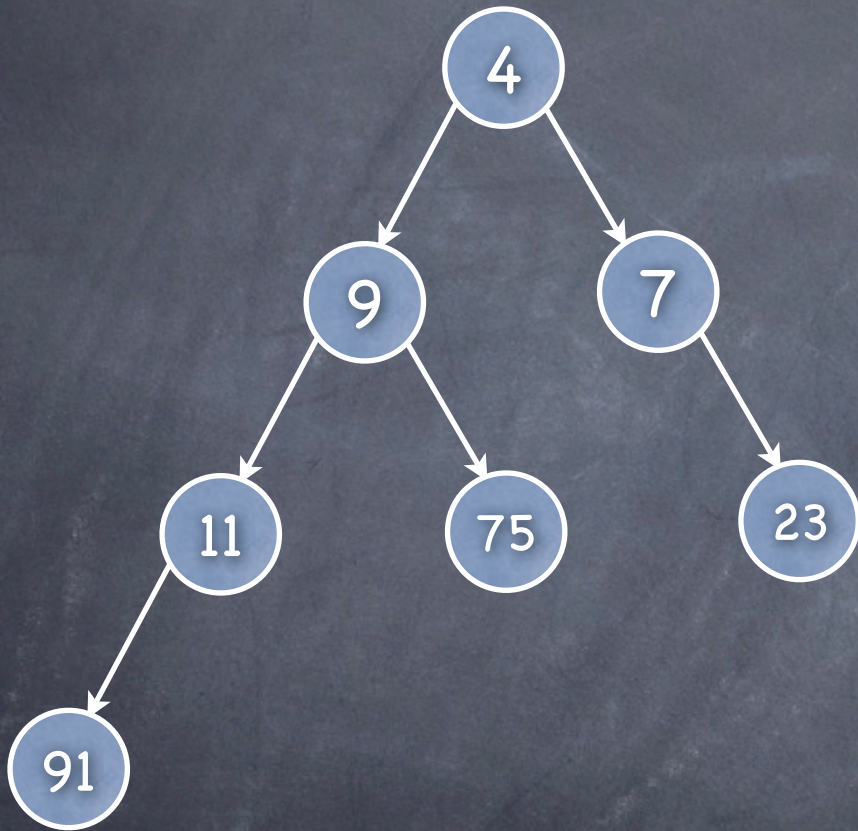
Example 2: Priority Queue



Commutativity

$\text{insert}(x) \Leftrightarrow \text{insert}(y)$

Example 2: Priority Queue

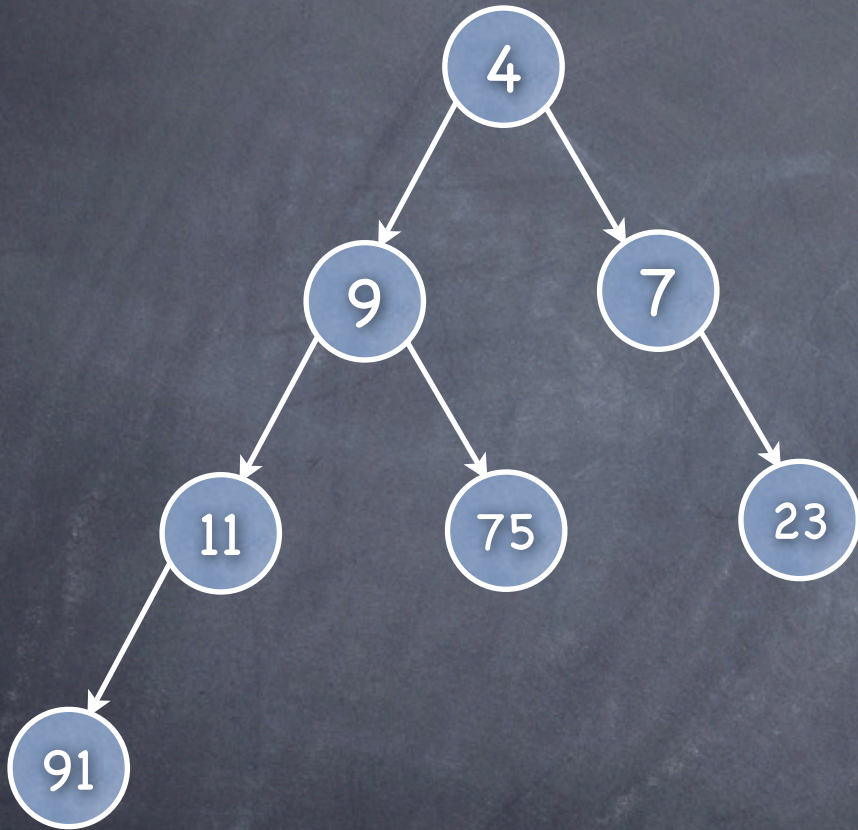


Commutativity

$\text{insert}(x) \Leftrightarrow \text{insert}(y)$

$\text{removeMin}() \Leftrightarrow \text{insert}(y) \quad x \leq y$

Example 2: Priority Queue



Commutativity

$$\text{insert}(x) \Leftrightarrow \text{insert}(y)$$

$$\text{removeMin}() / x \Leftrightarrow \text{insert}(y) \quad x \leq y$$

$$\text{min}() / x \Leftrightarrow \text{min}() / x$$

Outline

Outline

- ✓ Black-box linearizable base object

Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Now, where's my **free** transactional object?

Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Now, where's my **free** transactional object?

Synchronization

Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Now, where's my **free** transactional object?

Synchronization

Recovery

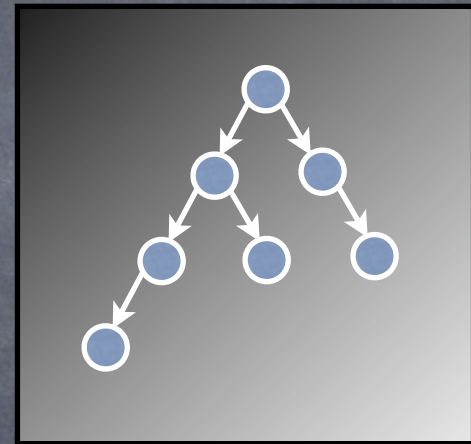
Txn Synchronization

Thread A

Thread B

Thread C

Black-box Concurrent
Data Structure



Txn Synchronization

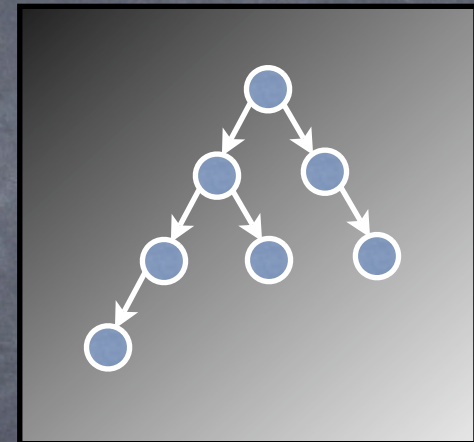
Thread A

Thread B

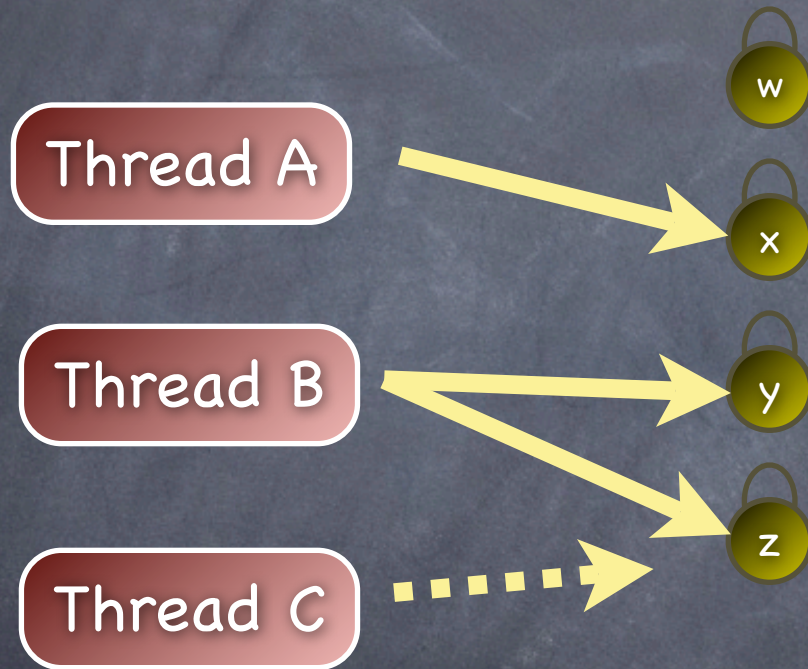
Thread C



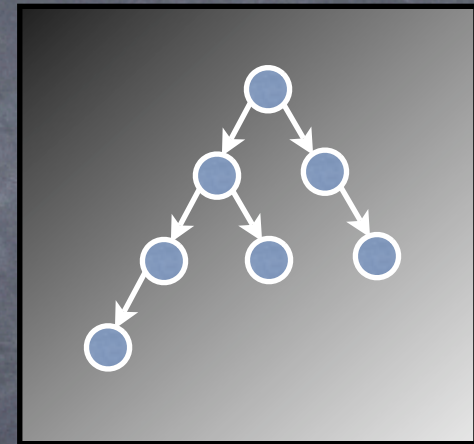
Black-box Concurrent
Data Structure



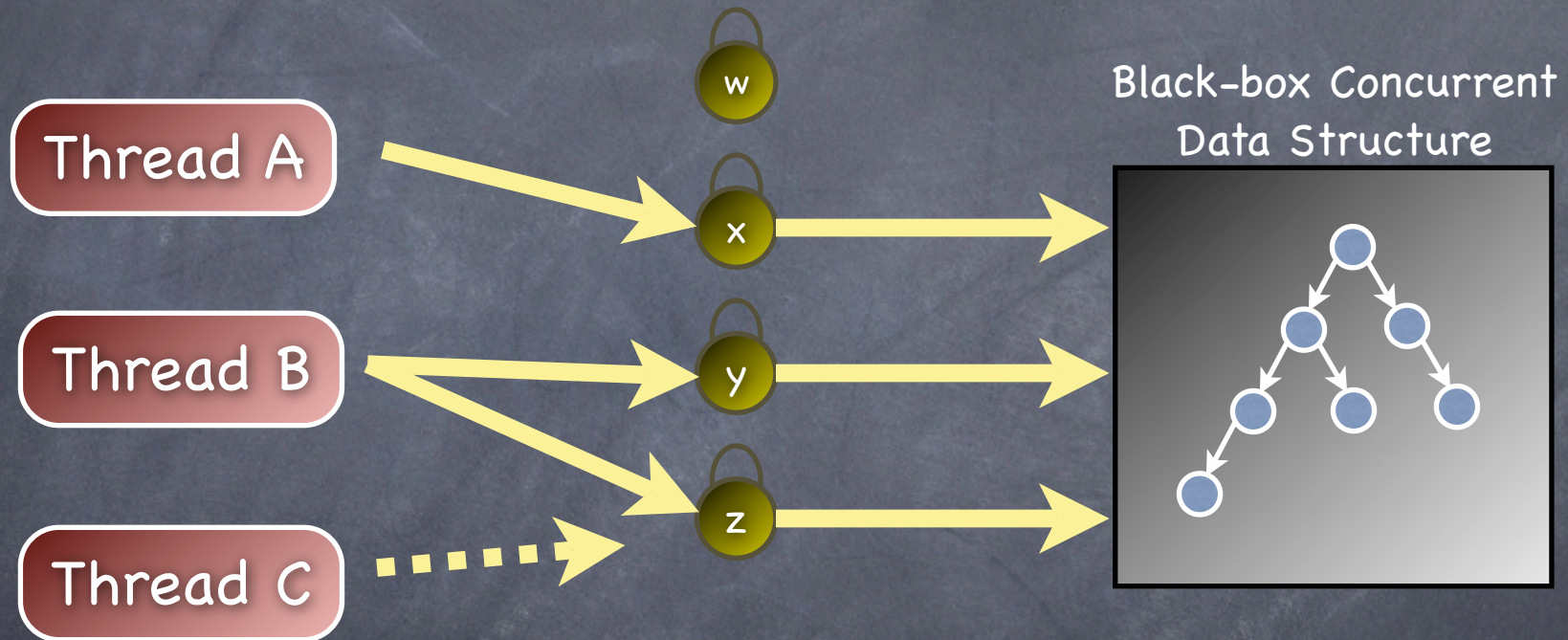
Txn Synchronization



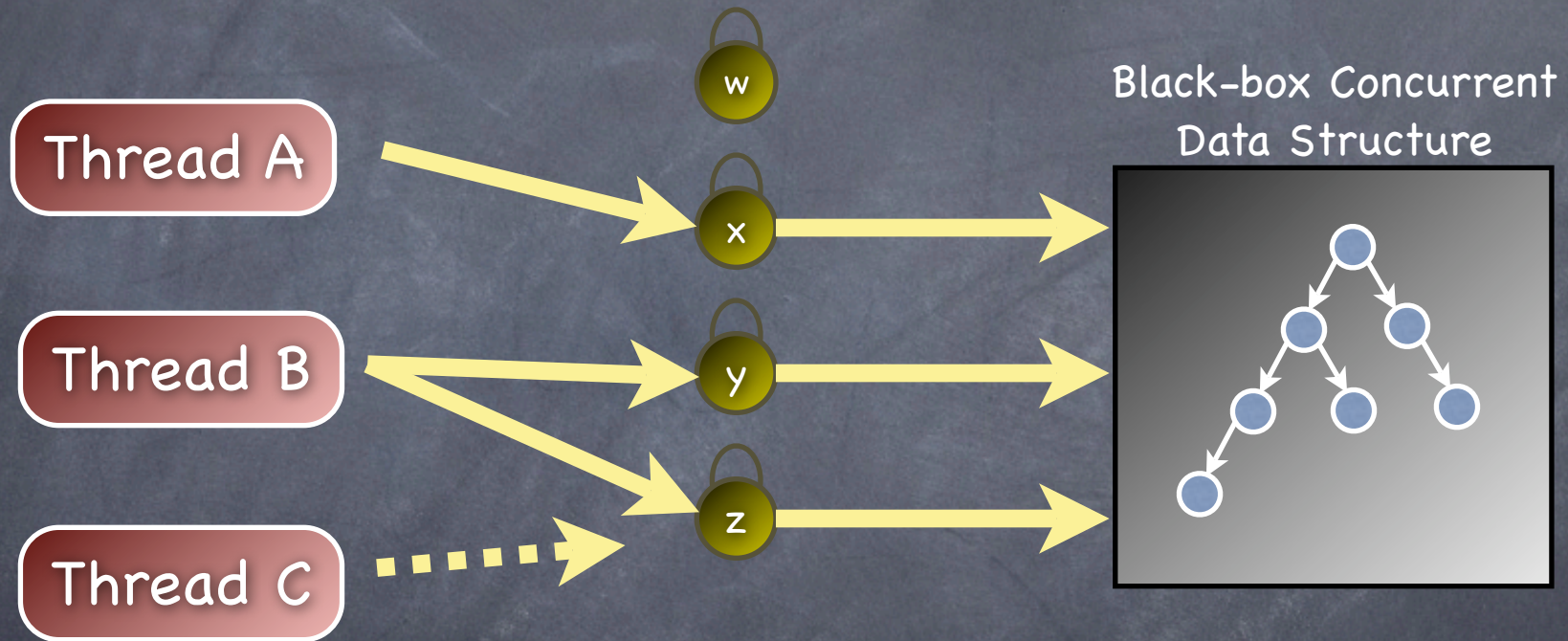
Black-box Concurrent Data Structure



Txn Synchronization

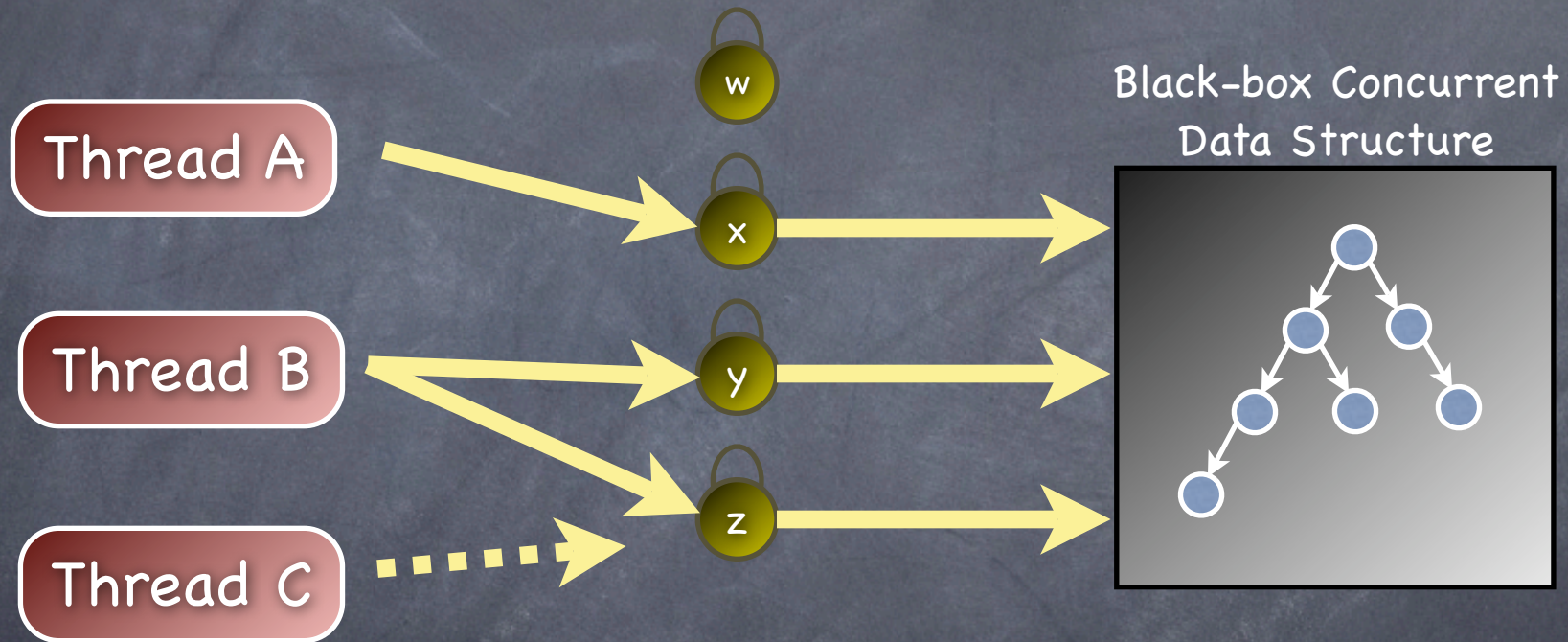


Txn Synchronization



*Transactions synchronize
with abstract locks*

Txn Synchronization



*Transactions synchronize
with abstract locks*

*Base object handles
thread synchronization
(e.g. lock-free)*

Txn Synchronization

- Acquire **abstract locks** before method invocation
- A lock **conflict** means methods don't commute
- Thus, abstract locks ensure that only commuting method calls execute concurrently

Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Now, where's my **free** transactional object?

- ✓ Synchronization: Abstract Locks

Recovery

Recovery

- Transaction invokes various methods
- Each successful method call, **log the inverse** method
- On abort **replay** the log:
 - Invoke inverses
 - Release locks

Thread A



Outline

- ✓ Black-box linearizable base object
- ✓ Commutativity and inverses

Now, where's my **free** transactional object?

- ✓ Synchronization: Abstract locks
- ✓ Recovery: Log inverses

Boosted Concurrent Set

```
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if ( result ) {
            Thread.onAbort(new Runnable() {
                public void run() { baseSet.remove(v); }
            });
        }
        return result;
    }
}
```

Boosted Concurrent Set

```
public class BoostedSet<Integer> {  
    ConcurrentSet<Integer> baseSet;  
    LockKey abstractLock;  
    ...  
    public boolean add(int v) {  
        abstractLock.lock(v);  
        boolean result = baseSet.add(v);  
        if ( result ) {  
            Thread.onAbort(new Runnable() {  
                public void run() { baseSet.remove(v); }  
            });  
        }  
        return result;  
    }  
}
```

Boosted Concurrent Set

```
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if ( result ) {
            Thread.onAbort(new Runnable() {
                public void run() { baseSet.remove(v); }
            });
        }
        return result;
    }
}
```

Boosted Concurrent Set

```
public class BoostedSet<Integer> {  
    ConcurrentSet<Integer> baseSet;  
    LockKey abstractLock;  
    ...  
    public boolean add(int v) {  
        abstractLock.lock(v);  
        boolean result = baseSet.add(v);  
        if ( result ) {  
            Thread.onAbort(new Runnable() {  
                public void run() { baseSet.remove(v); }  
            });  
        }  
        return result;  
    }  
}
```

acquire abstract lock to
ensure comm. isolation

Boosted Concurrent Set

```
public class BoostedSet<Integer> {  
    ConcurrentSet<Integer> baseSet;  
    LockKey abstractLock;  
    ...  
    public boolean add(int v) {  
        abstractLock.lock(v);  
        boolean result = baseSet.add(v);  
        if ( result ) {  
            Thread.onAbort(new Runnable() {  
                public void run() { baseSet.remove(v); }  
            });  
        }  
        return result;  
    }  
}
```

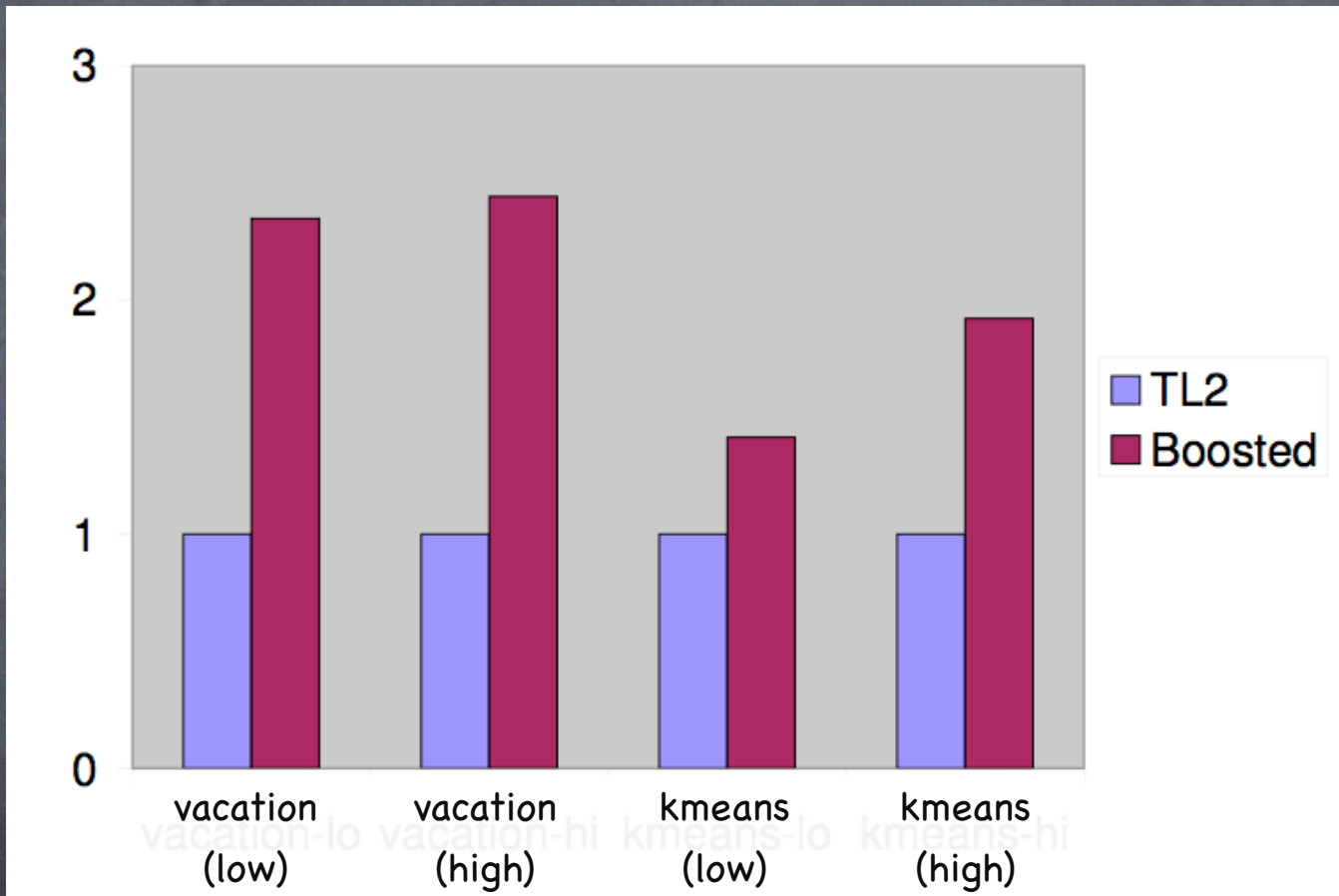
acquire abstract lock to ensure comm. isolation

log the inverse operation for recovery

Experiments

- Boosting implementation in C
- Based on Transactional Locking 2 (TL2)
- Modified STAMP benchmarks to use Boosting
 - vacation -- A travel reservation system
 - kmeans -- A clustering technique
- Ran on 8-way 2.0 GHz Xeon processor

Experiments



Normalized throughput for two STAMP benchmarks

Summary

- Concurrent objects are sequential in a txn
- Use semantics for synchronization, recovery
- No read/write sets → avoid false conflicts
- Highly concurrent transactional objects
... for free!

Open Nested Txns

Open Nested Txns

- ONT addresses same problem
by releasing effects of nested txns to parent

Open Nested Txns

- ONT addresses same problem
by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks

Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology

Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)

Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
- Lock-coupling doesn't correspond to nesting

Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
- Lock-coupling doesn't correspond to nesting
- Requires re-implementing base object

Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
- Lock-coupling doesn't correspond to nesting
- Requires re-implementing base object
- No r/w sets with Boosting: base object perf

Related Work

No existing STM mechanism approaches lock-free algorithms

- Ni et al. Open Nesting in Software Trans'l Memory. PPOPP 2007.
- Carlstrom et al. Transactional Collection Classes. PPOPP 2007.
- Dice et al. Transactional Locking II. DISC 2006.

This Paper:

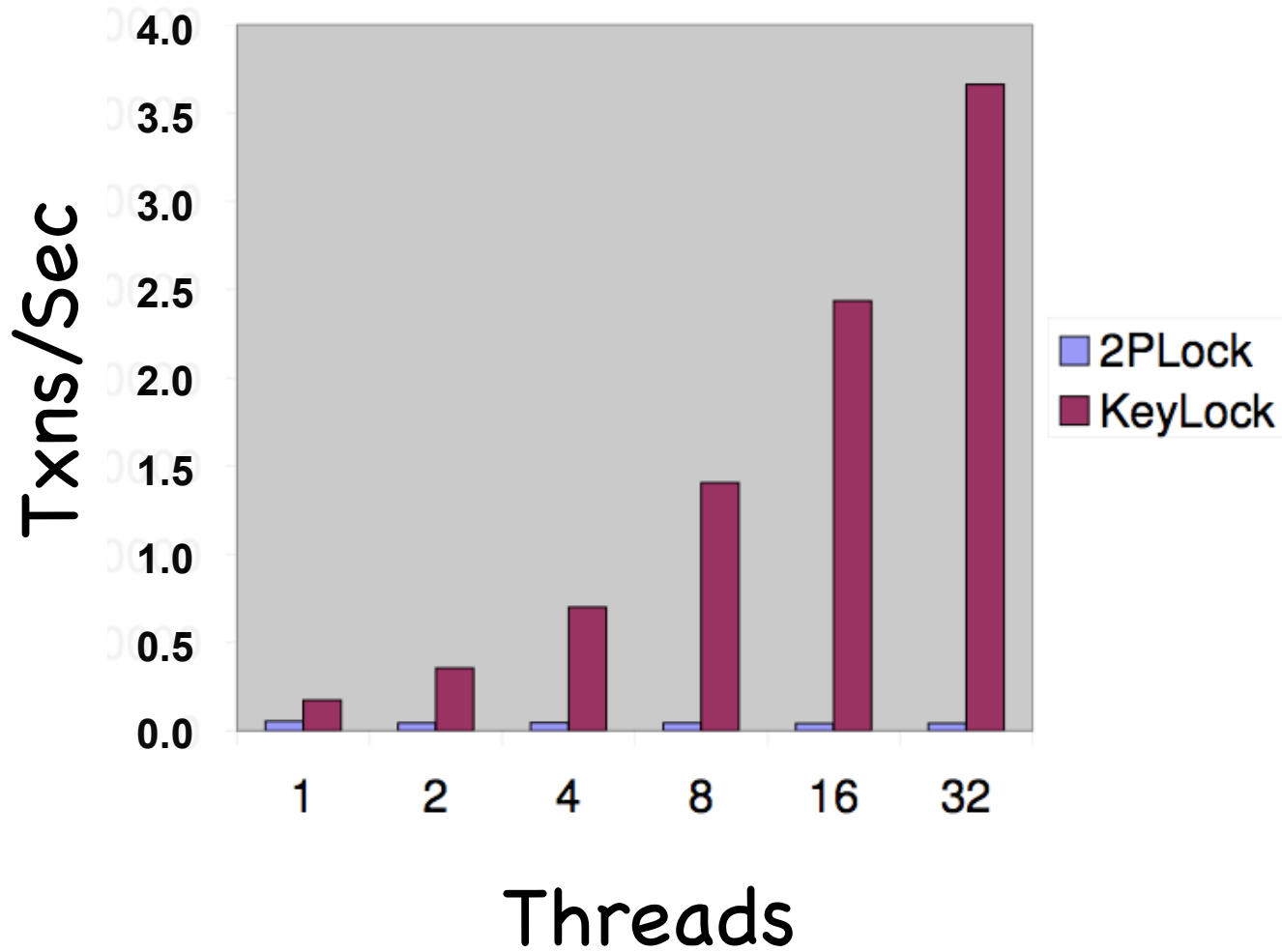
- Herlihy, Koskinen. Transactional Boosting: A Methodology for Highly Concurrent Transactional Objects. PPOPP 2008.
- Proofs in Brown Technical Report CS-07-08

Thanks!

Questions?



Lock Comparison



BOOSTING

BOOSTING

atomic { }

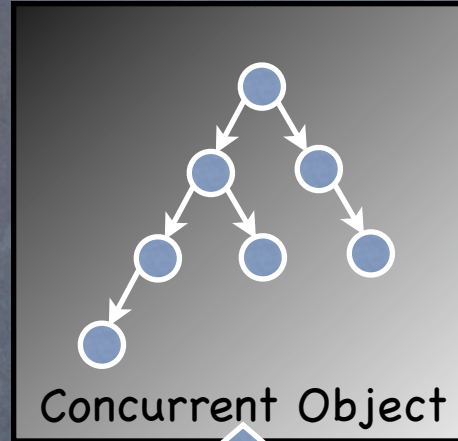
BOOSTING

atomic { }



BOOSTING

atomic {



}

semantics!