# Lecture 17: Concurrent Objects

COSC 273: Parallel and Distributed Computing

Spring 2023

# Announcements

16+

1. Lab 03 Due Friday
   - Mandelbrot computations using `Vector` operations
   - Make sure your machine supports `Vector` ops **today**:

```
> javac --add-modules jdk.incubator.vector SomeFile.java
> java --add-modules jdk.incubator.vector SomeFile
```

   on HPC cluster, first run

```
> module load amh-java/19.0.1
```

2. Coming weeks
   - final projects (groups up to 3)
   - one more written assignment
   - quizzes, ~weekly starting next week

- sorting
- prime #s
- Choose own adventure

# Concurrent Objects

# So Far

*Synchronized* access to shared objects via locking:

```
public class SharedCounter {
    int count = 0;
    Lock lock = new SomeLockImplementation();
    public void increment() {
        lock.lock()
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

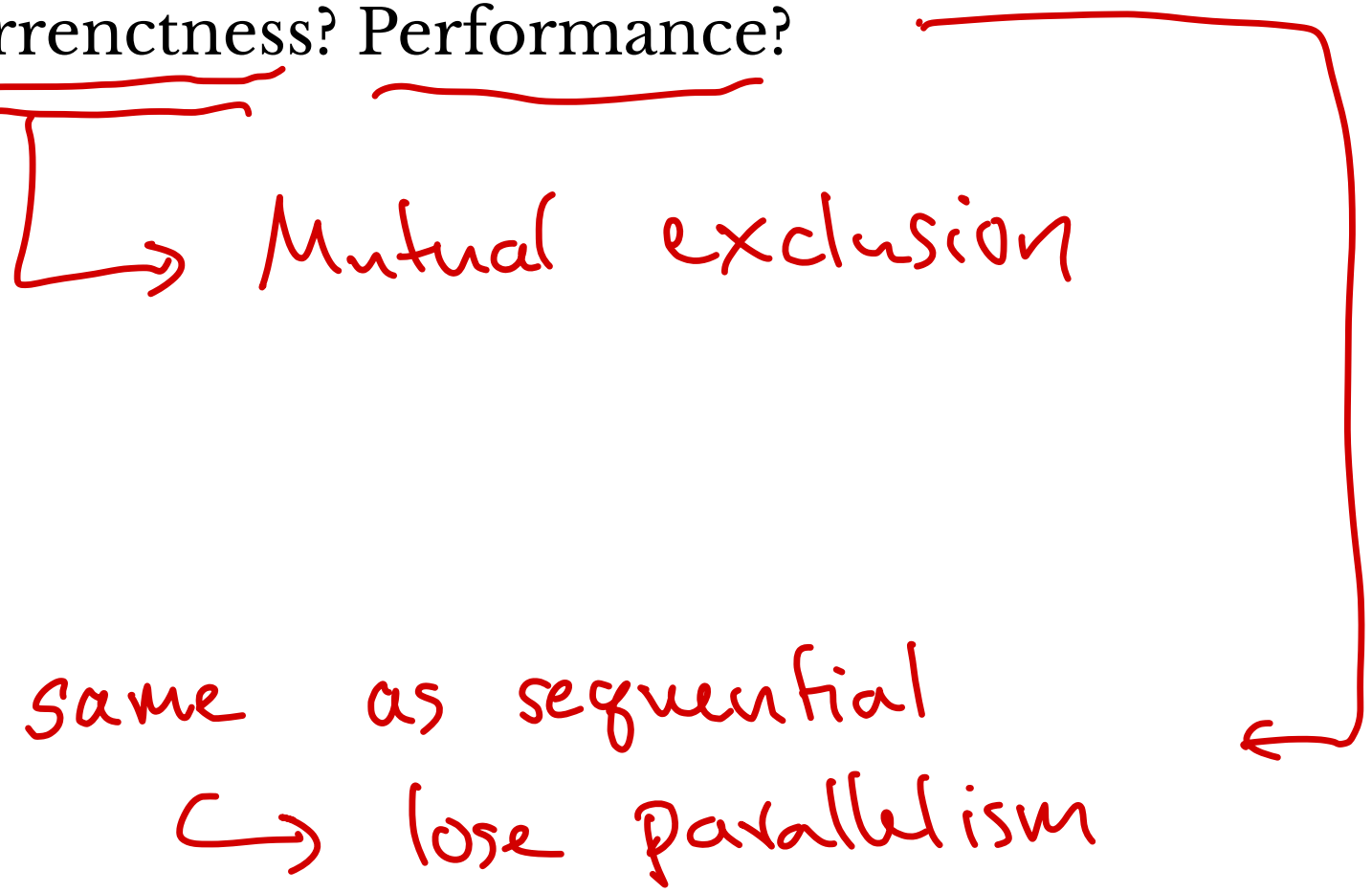
obtain lock

Critical section

release lock

# Locking (Dis)advantages?

Correnctness? Performance?



# Today:

abstract data type (queue, list, stack)

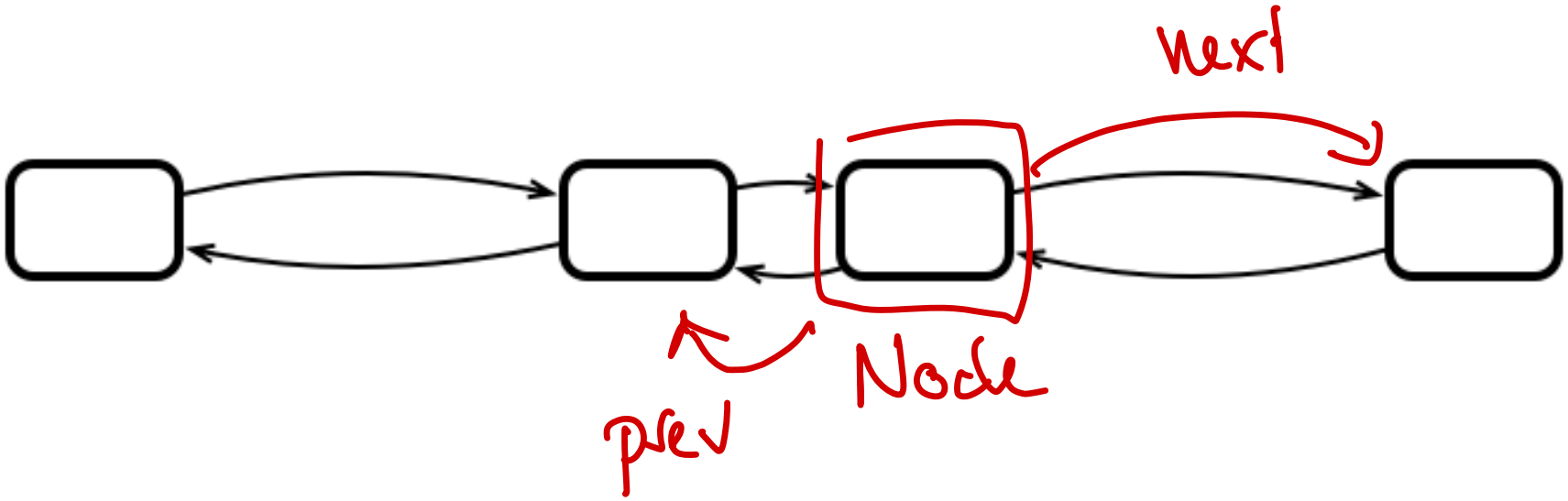
Concurrent ADTs and data structures!

**Main Questions:**

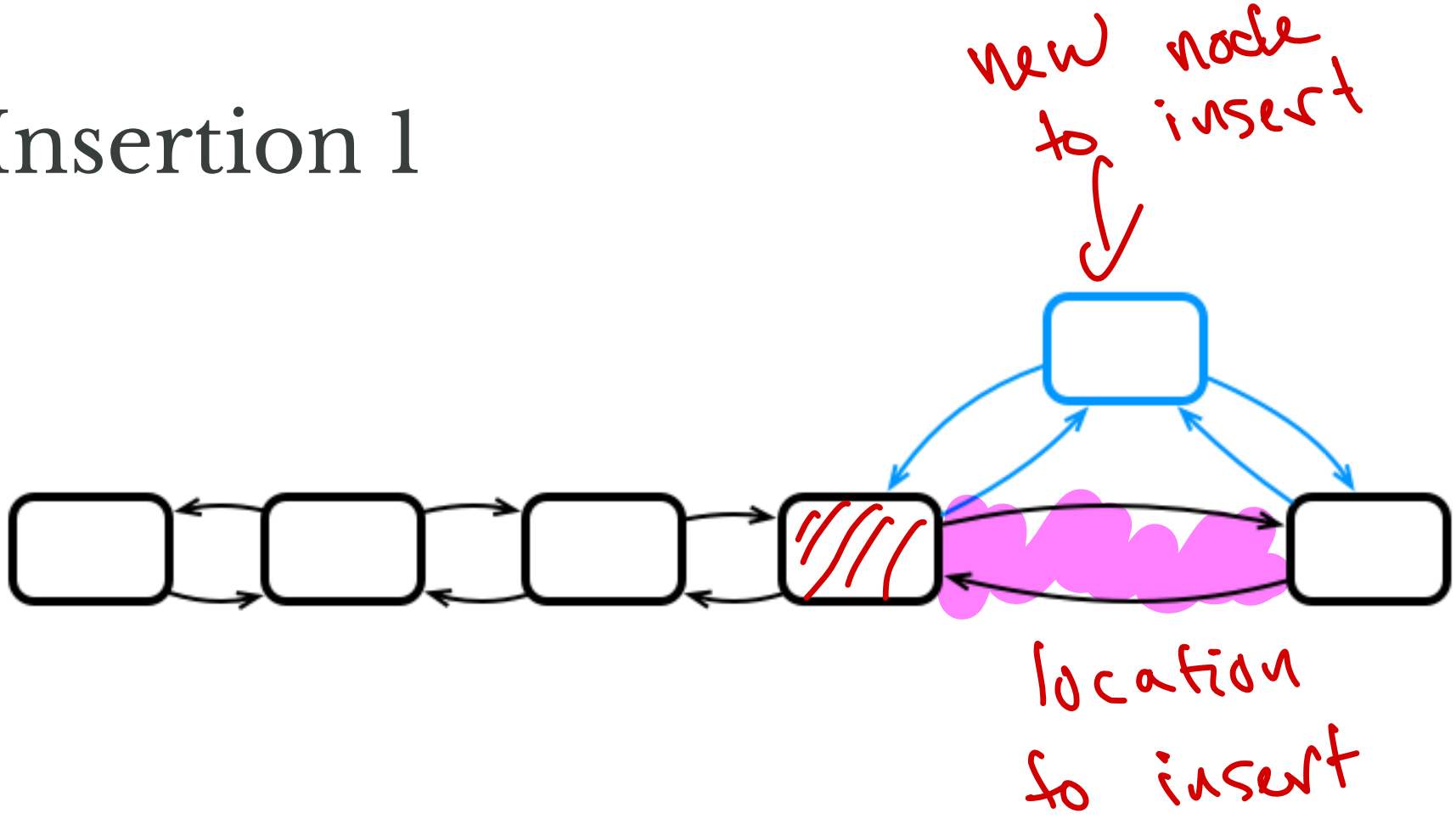
1. How can we guarantee correctness of data structures that allow concurrent accesses? — maybe easy w/ locks?
   - What do we even mean by "correct" for concurrent data structures?
2. How can we achieve the best multithreaded performance?
   - Can we design data structures where parallel architecture gives some speed up?
     - locking seems to *force* sequential executions...

# Linked Lists

Recall: a (doubly) linked list

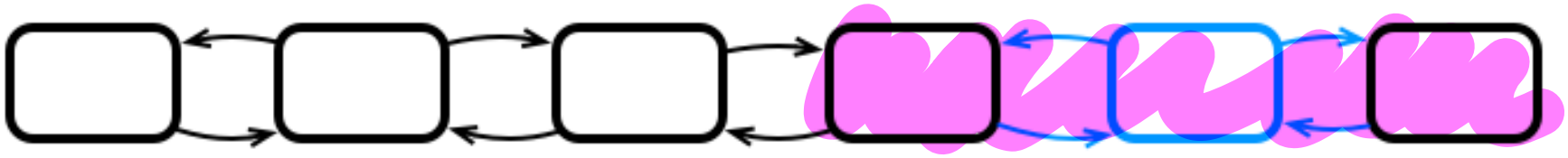


# Insertion 1



# Insertion 2

modify prev/next of these nodes

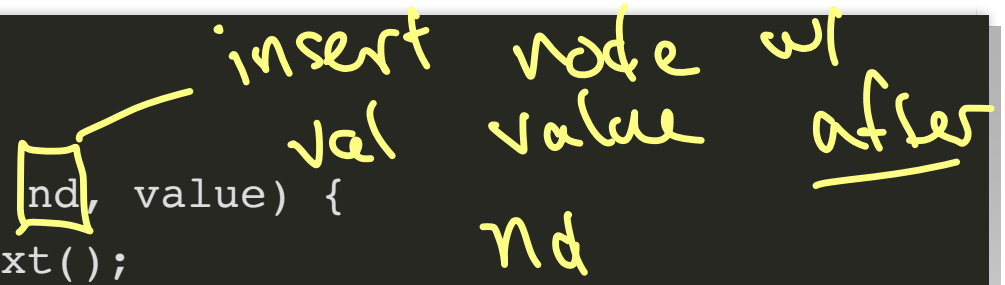


# Linked List in Code: Node

```
class Node {
    public Node next;
    public Node prev;
    public int value;
}
```

# Linked List in Code

```
public class MyLinkedList {
    private Node head;
    public void insert (Node nd, value) {
      Node next = nd.getNext();
      Node cur = new Node(value);
      nd.next = cur;
      cur.prev = nd;
      cur.next = next;
      if (next != null) next.prev = cur;
    }
}
```



# Insertion with Multiple Threads

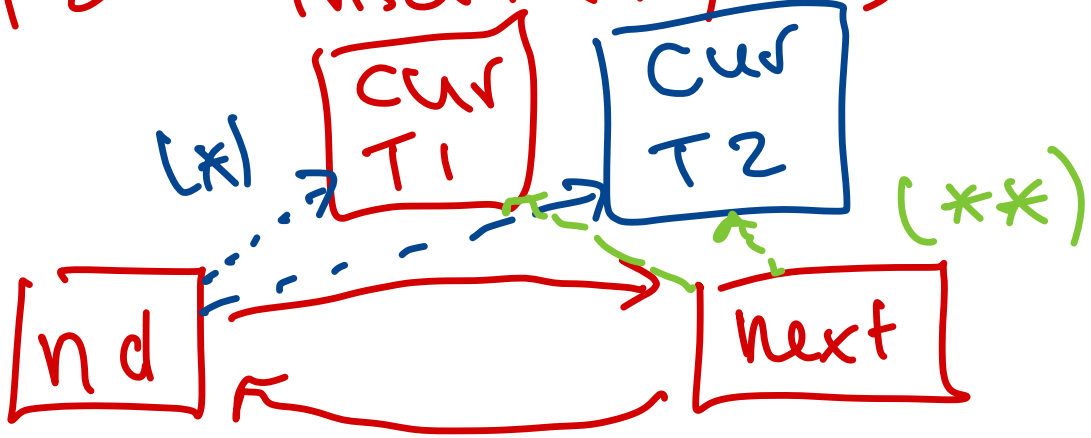
What could go wrong?

```
public void insert (Node nd, value) {
    Node next = nd.getNext();
    Node cur = new Node(value);
    nd.next = cur;
    cur.prev = nd;
    cur.next = next;
    if (next != null) next.prev = cur;
}
```

(*) (**)

T1 : insert (nd, 10) {
T2 : insert (nd, 9) }

concurrently...

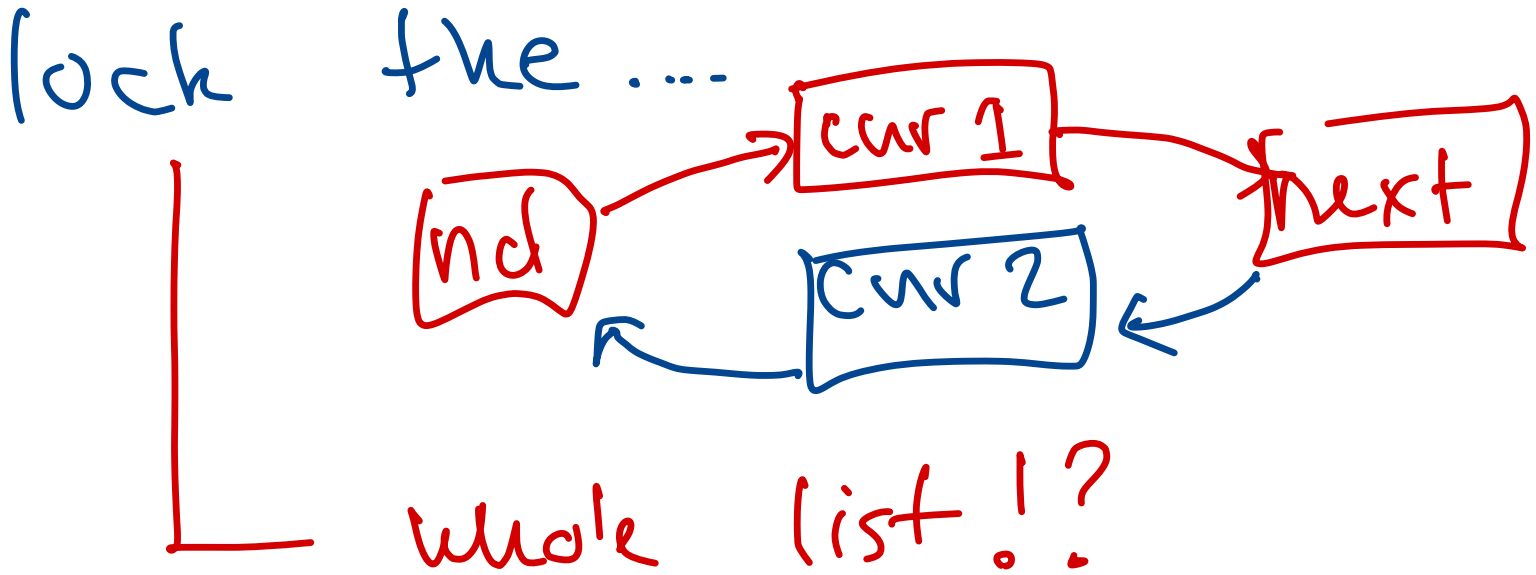



Doesn't guarantee that either is inserted properly

# How to Fix The Problem?

```
public void insert (Node nd, value) {
    Node next = nd.getNext();
    Node cur = new Node(value);
    nd.next = cur;
    cur.prev = nd;
    cur.next = next;
    if (next != null) next.prev = cur;
}
```

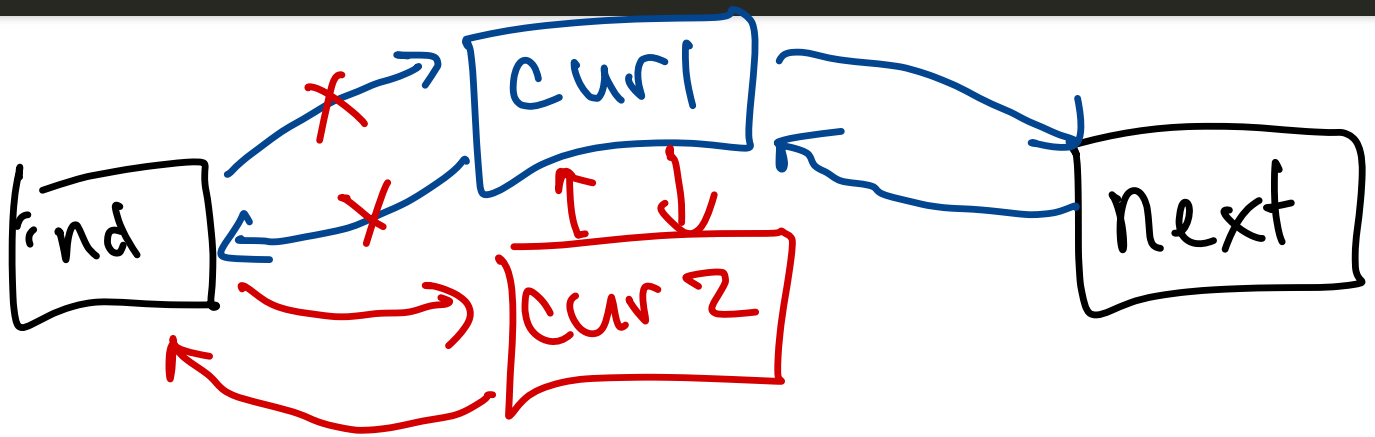
Surround w/ lock ..



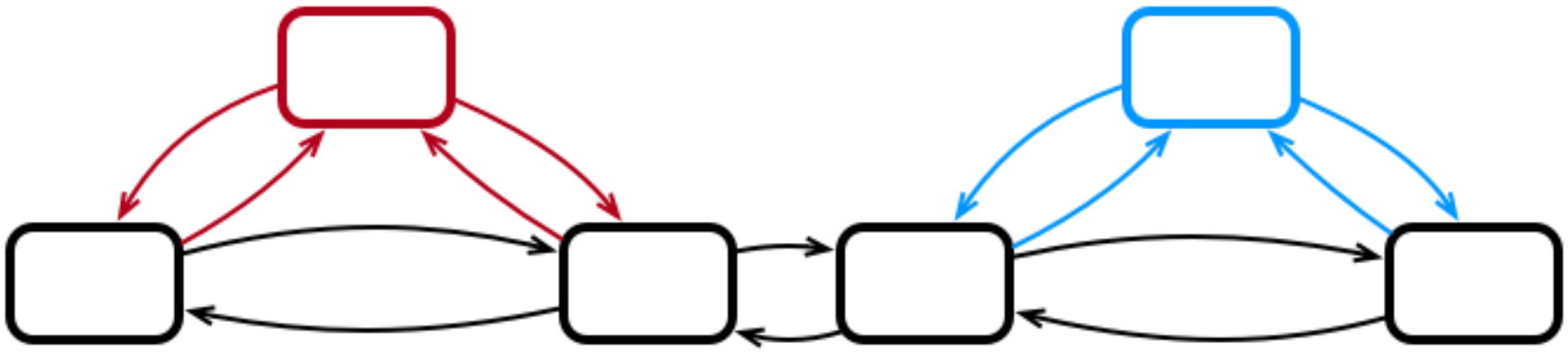
# A Fix: Locking the List

```
public class MyLinkedList {
    private Lock lock;
    public void insert (Node nd, value) {
        lock.lock();
        try { // all of this is critical section
            Node next = nd.getNext();
            Node cur = new Node(value);
            nd.next = cur; cur.prev = nd;
            cur.next = next;
            if (next != null) next.prev = cur;
        } finally { lock.unlock(); }
    }}
```

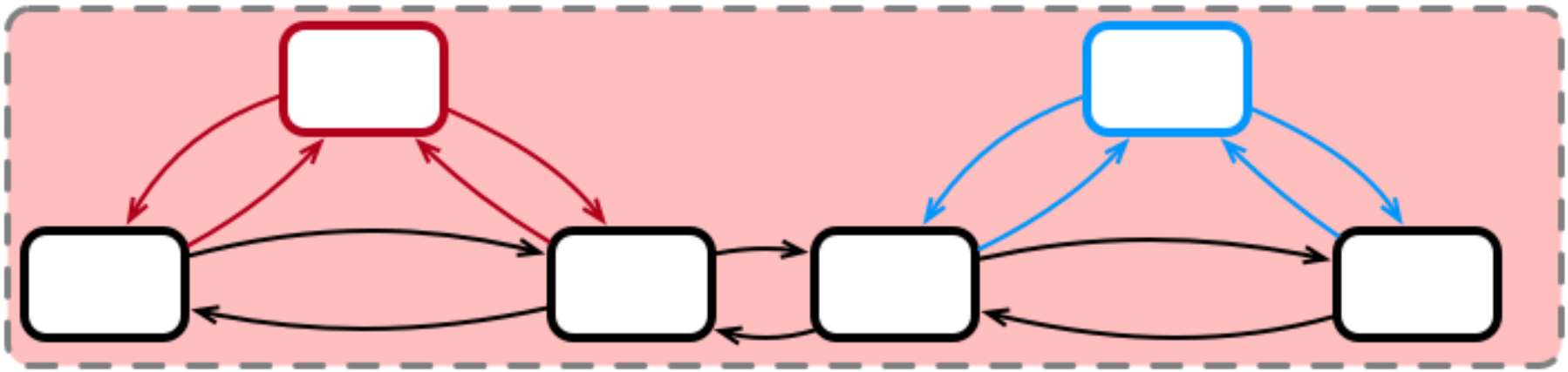
crit. section



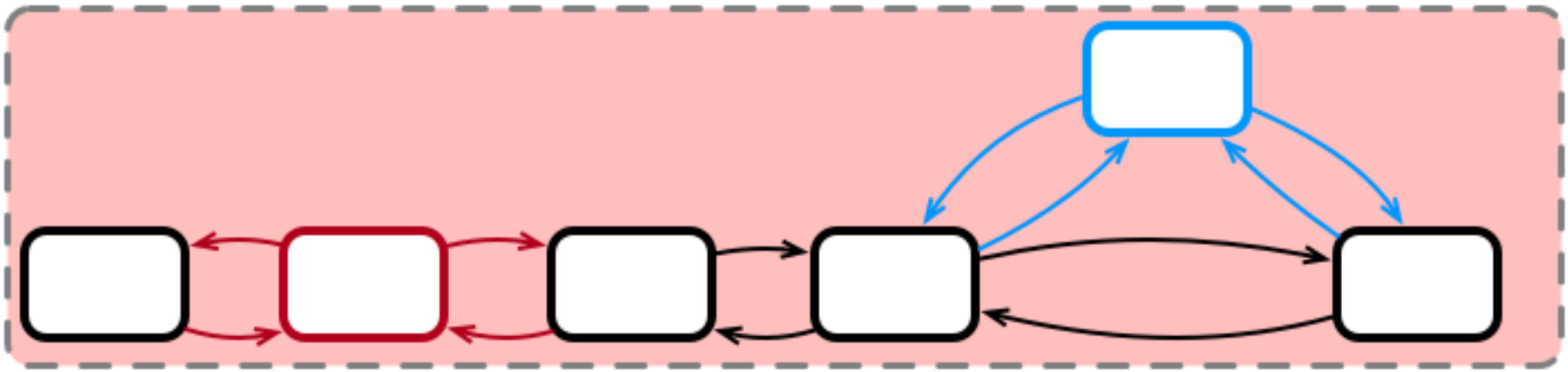
# Illustration of Locked Execution



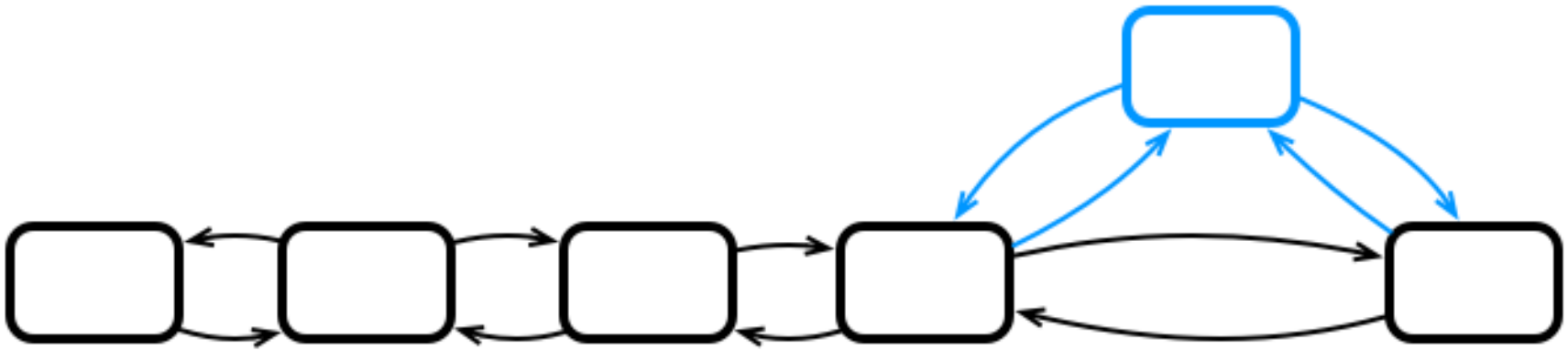
# Red Acquires Lock



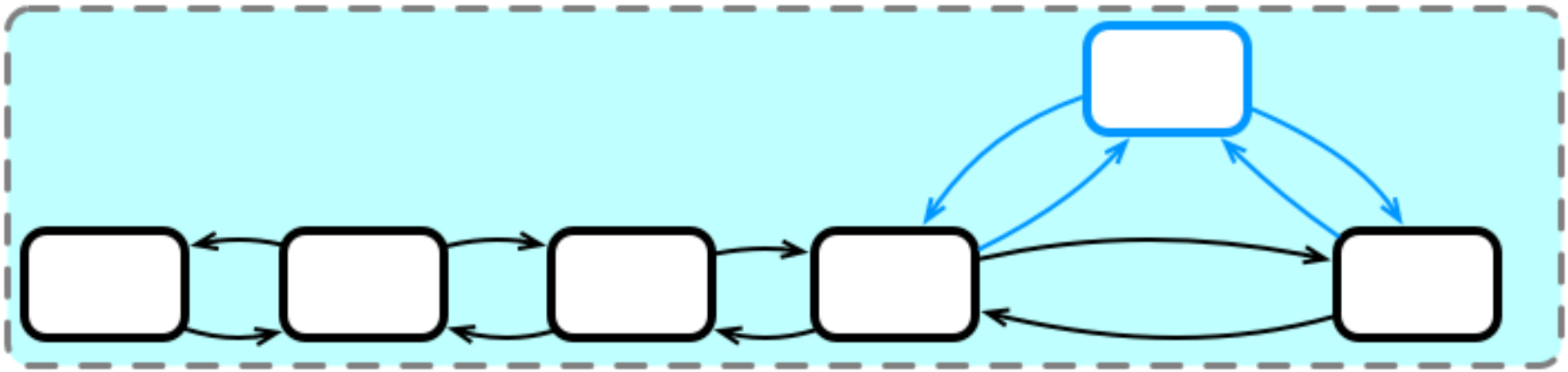
# Red Inserts Element



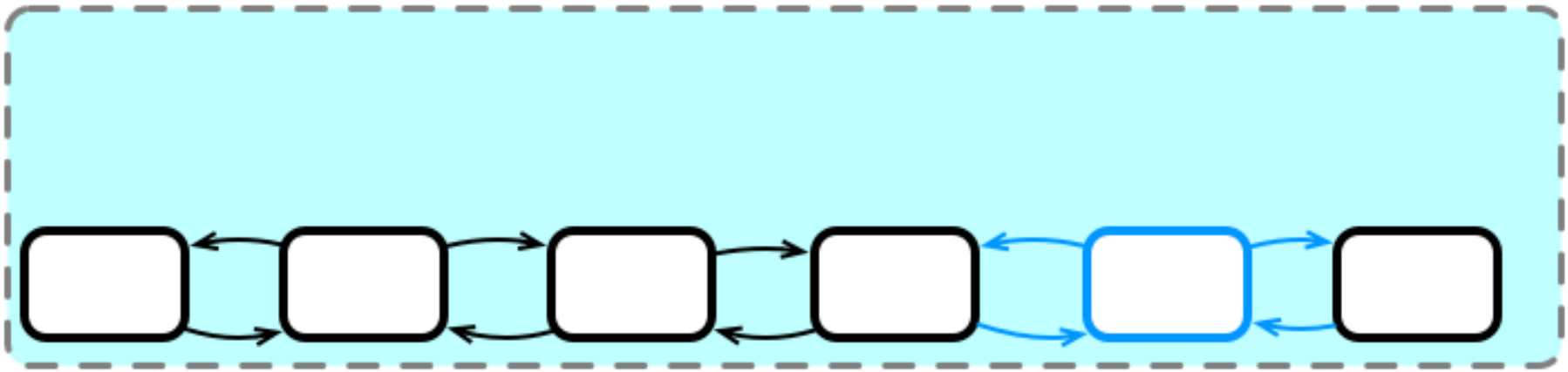
# Red Releases Lock



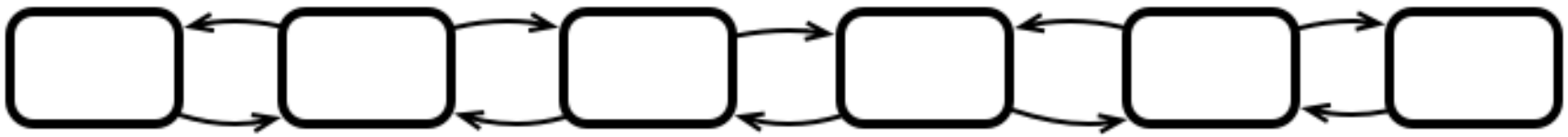
# Blue Acquires Lock



# Blue Inserts Element



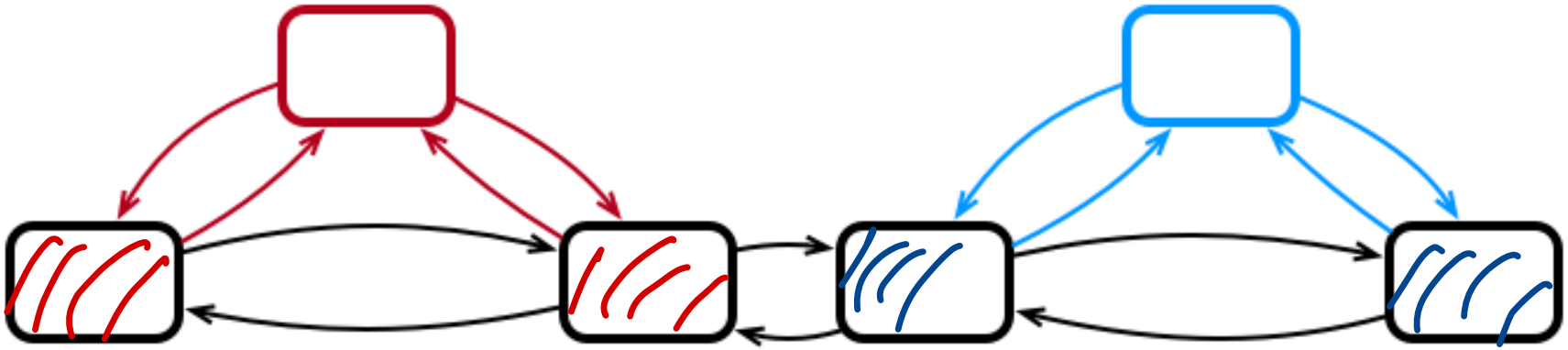
# Blue Releases Lock



Nice...

...but...

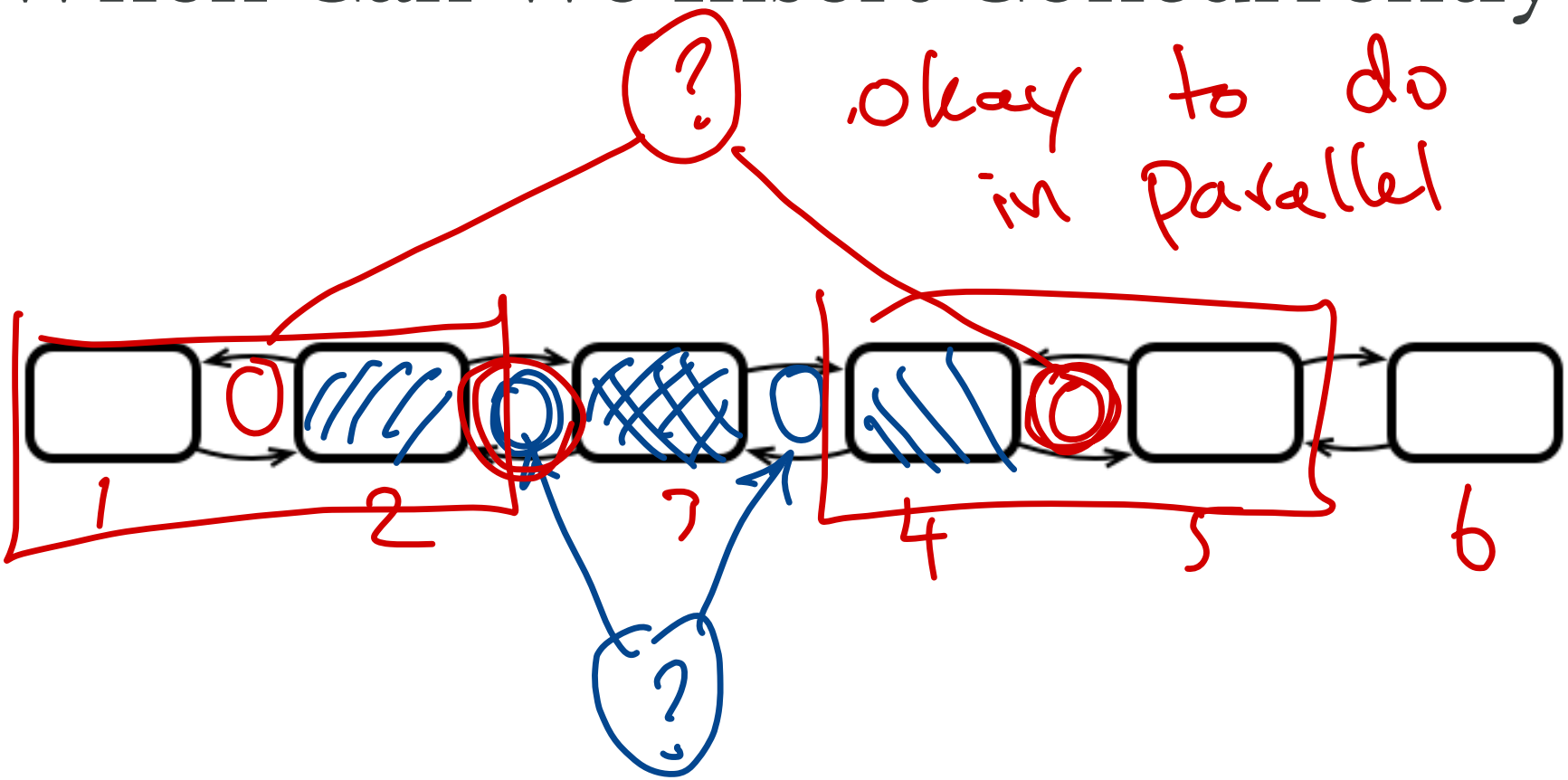
# ...Could we Have Done This Faster?



How?

Idea: Can insert nodes in parallel if no overlapping modifications of nodes

# When Can We Insert Concurrently?

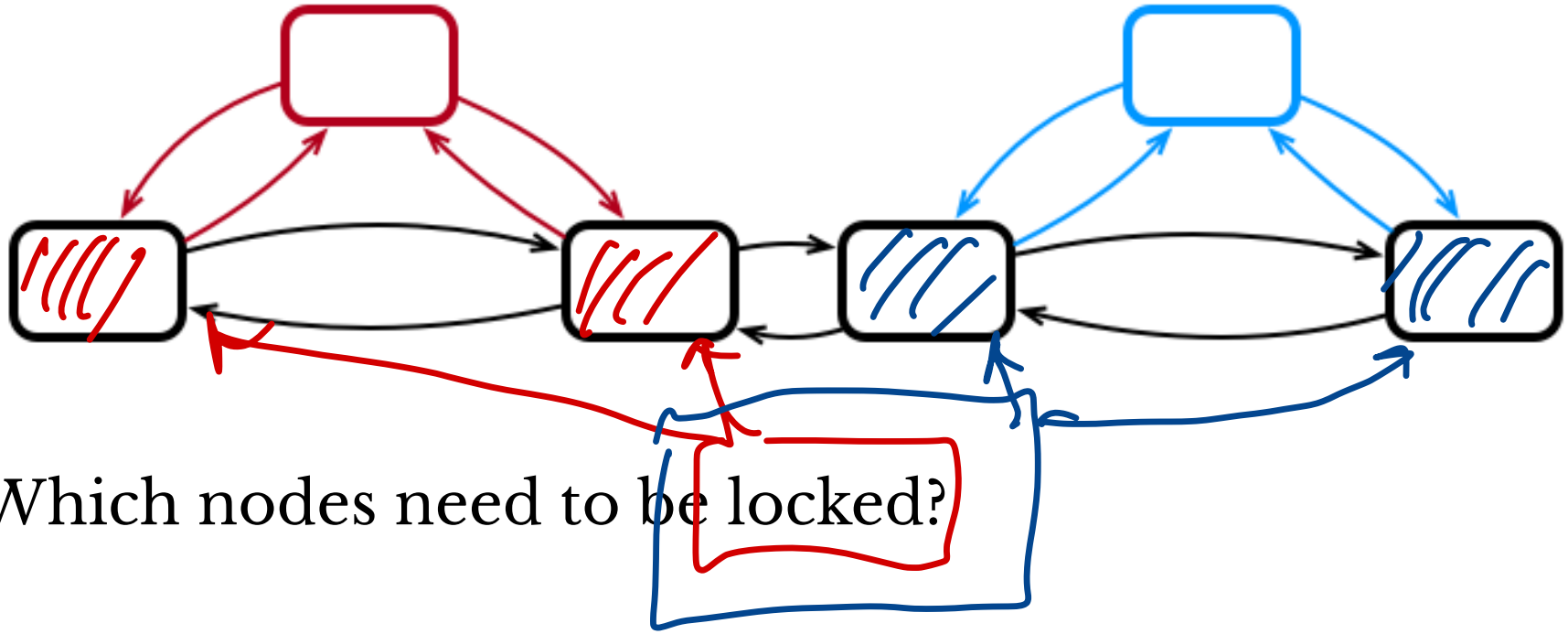


# What Should we Lock?

Not the whole list!

→ lock individual nodes

# Idea: Locking Individual Nodes



Which nodes need to be locked?

# Locking Nodes in Code

```
class Node {
    private Lock lock;
    public Node next;
    public Node prev;
    public int value;

    public void lock() { lock.lock(); }
    public void unlock() { lock.unlock(); }
}
```

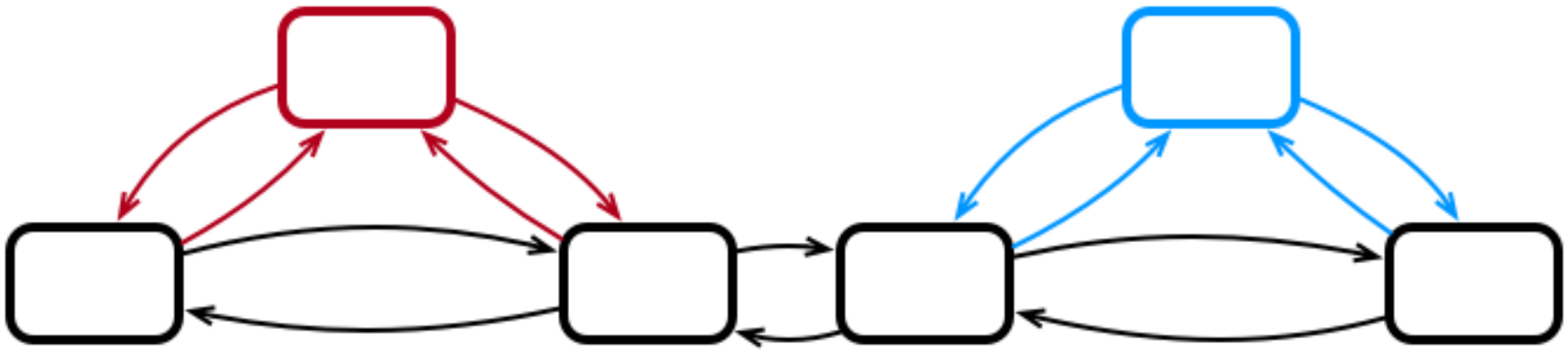
# Insertion with Locked Nodes

```
public void insert (Node nd, value) {
    Node cur = new Node(value);
    nd.lock();
    try {
      Node next = nd.getNext();
      if (next != null) next.lock();
        nd.next = cur; cur.prev = nd; cur.next = next;
        if (next != null) next.prev = cur;
    } finally {
        if (next != null) next.unlock();
        nd.unlock();
    }}
```

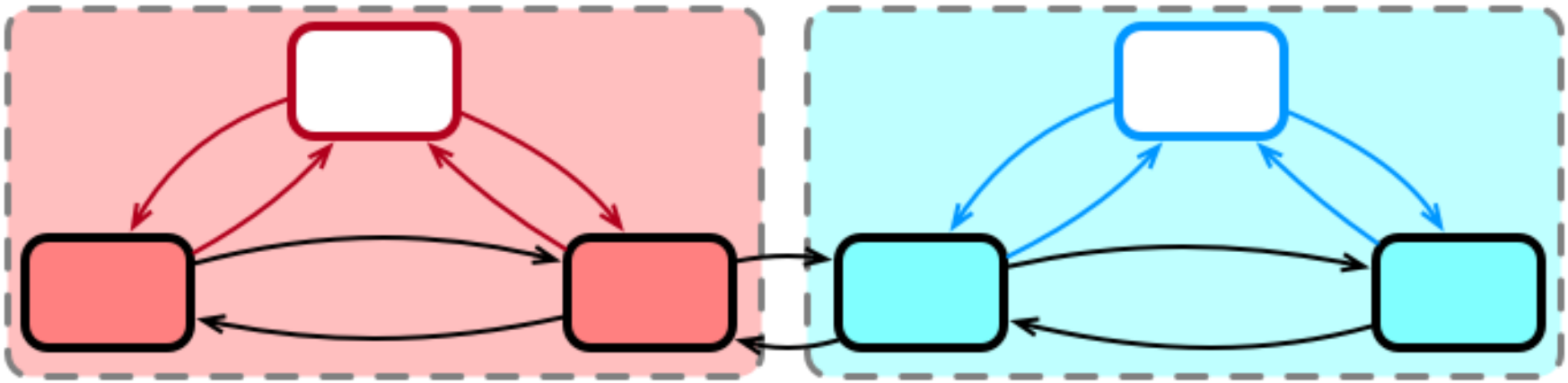
obtain lock for nd

obtain lock for next nod

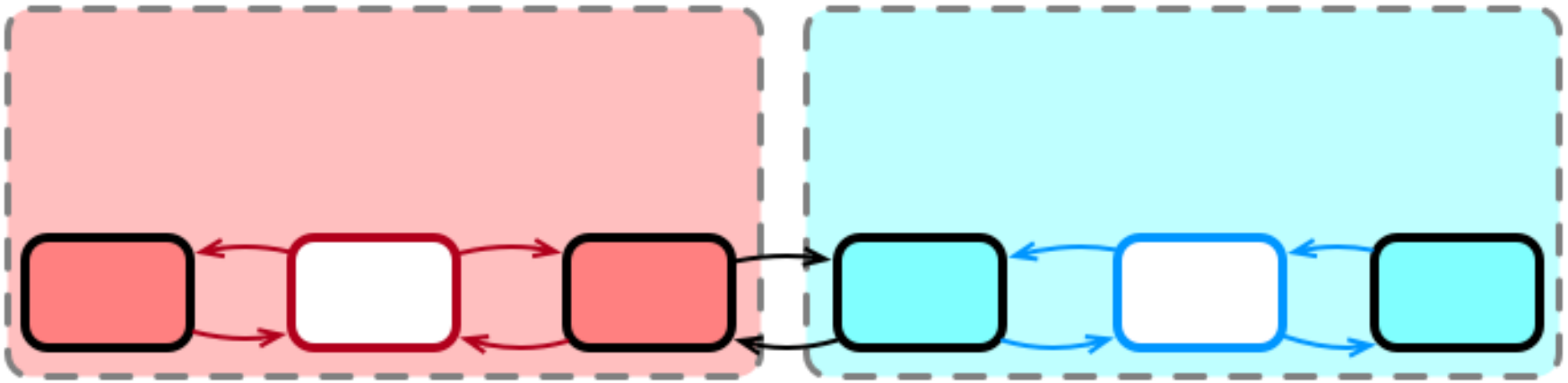
# Concurrent Insertions



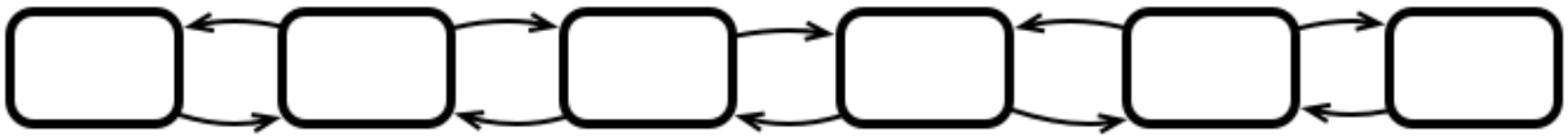
# Acquiring Locks



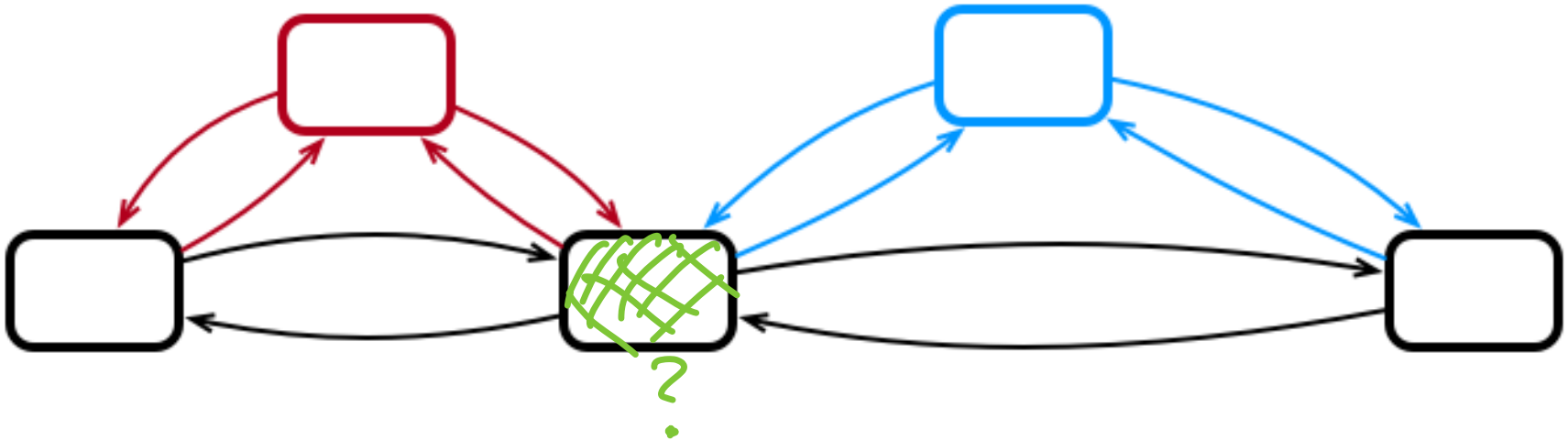
# Both Insert



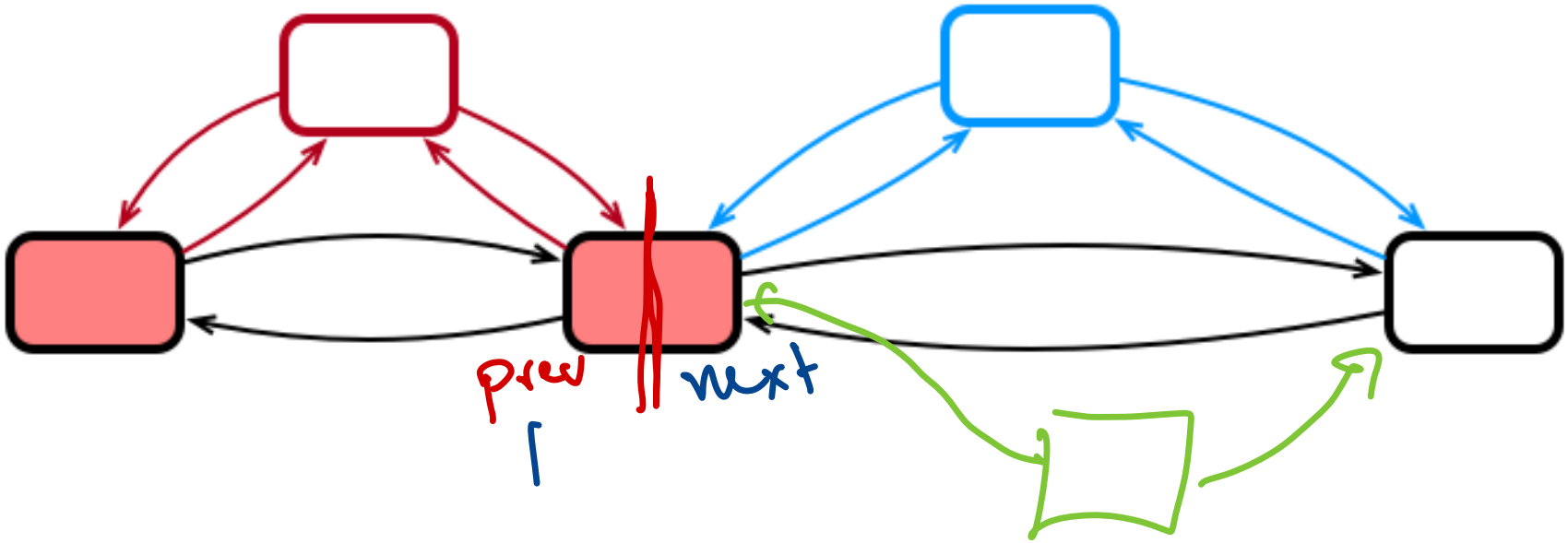
# Both Release



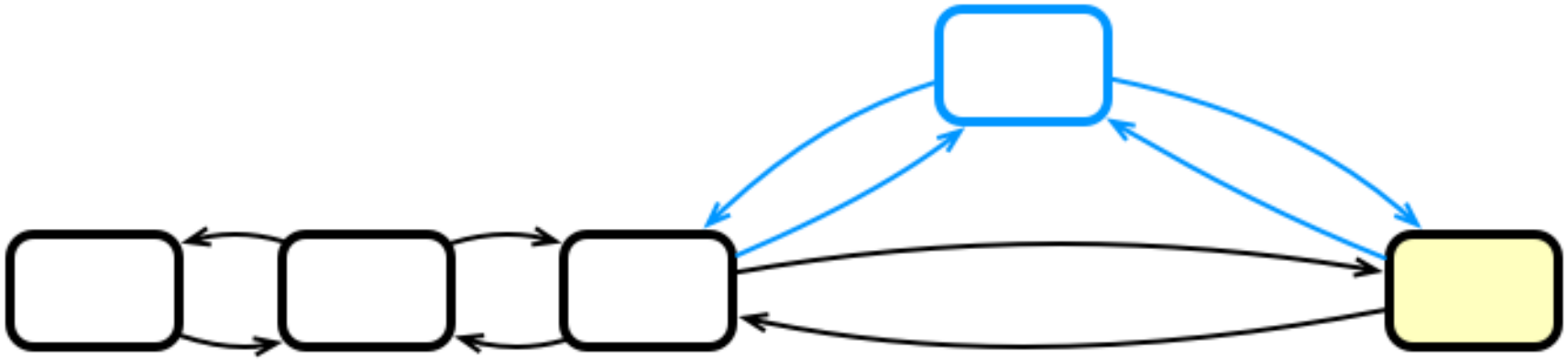
# What Happens with Contention?



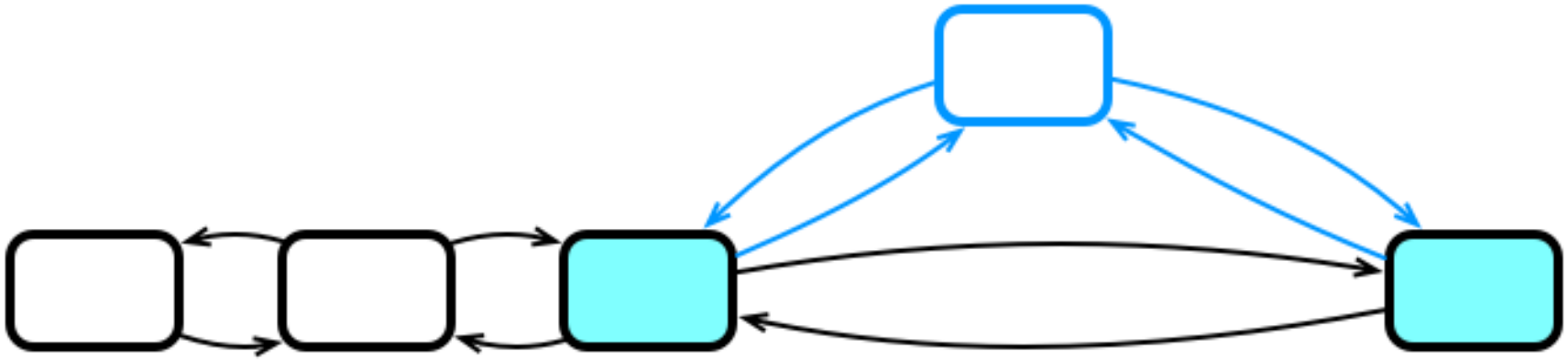
# Red Acquires Locks (Blue Waits)



# Red Inserts & Releases Locks



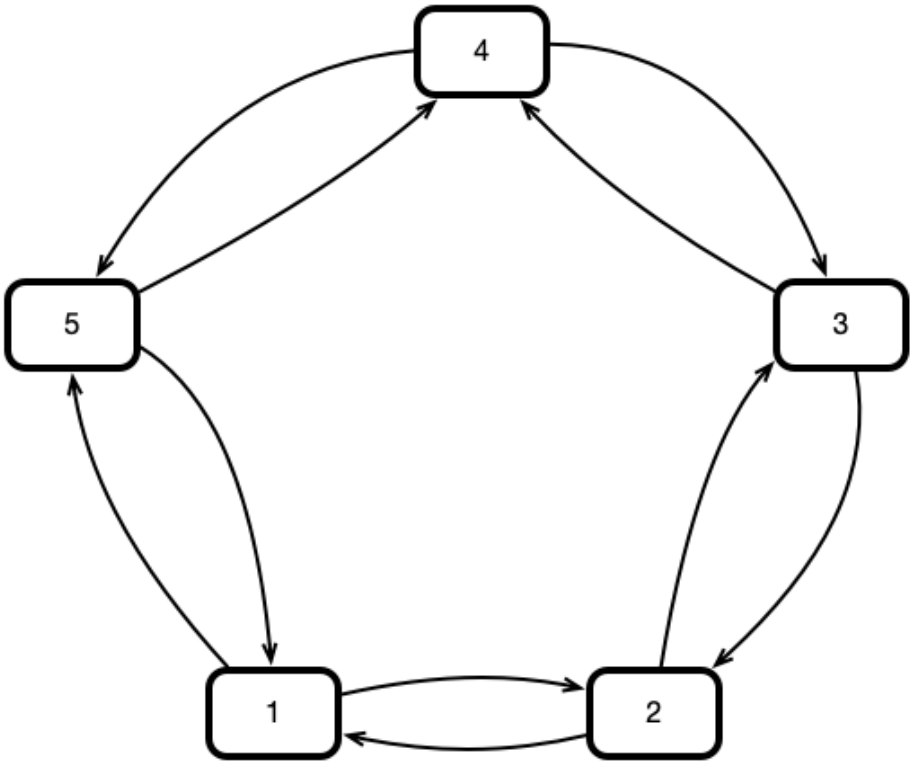
# Blue Finally Acquires Locks



# Blue Inserts & Releases Locks

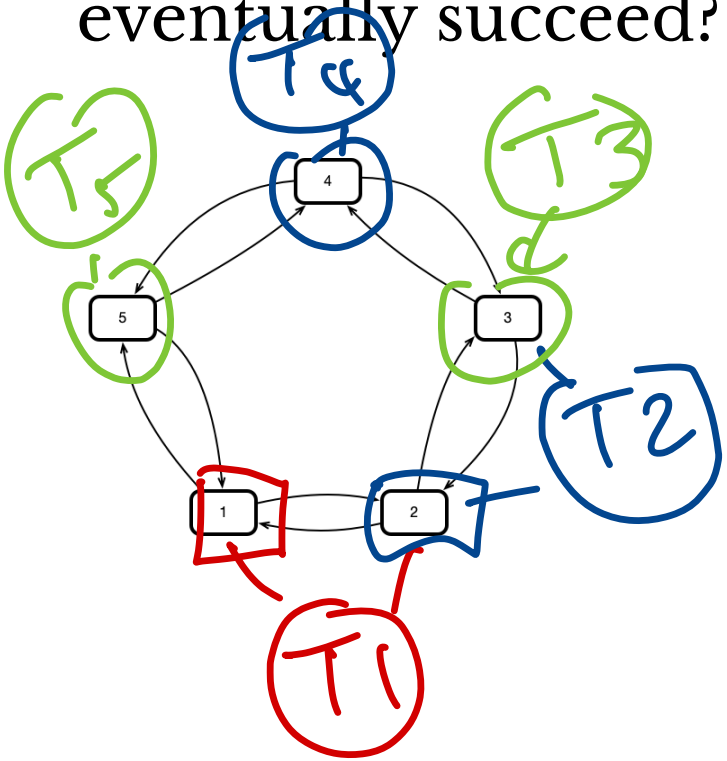
# This Seems Pretty Good!

# A Slightly Different Scenario



# Question

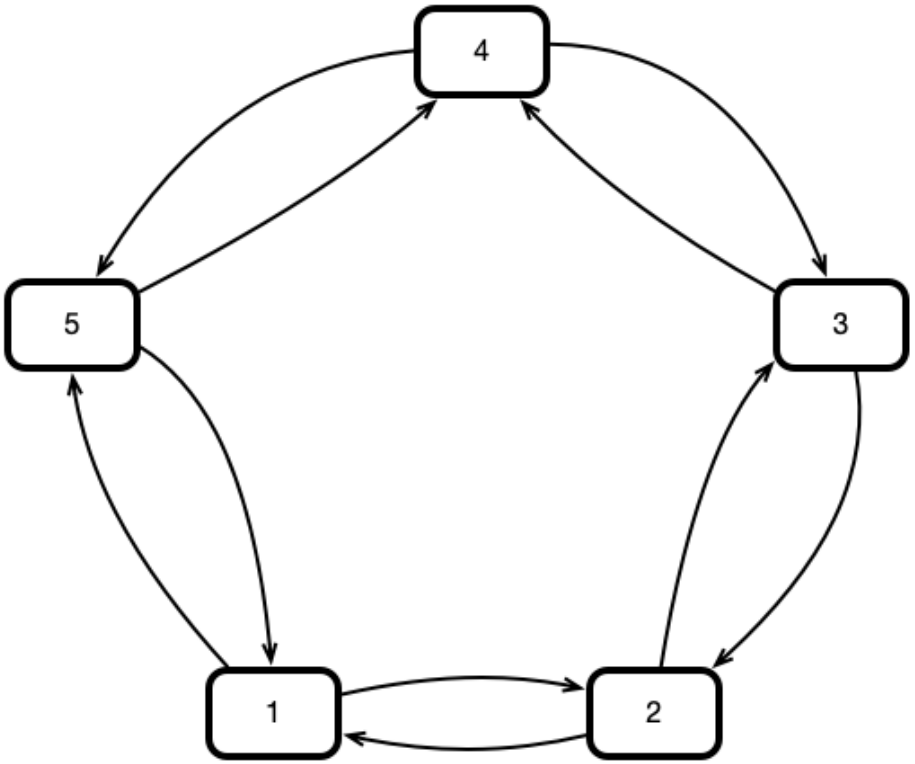
Are multiple concurrent insertions guaranteed to eventually succeed?




STUCK

Blocking execution

# Bad Executions?



# Morals from Previous Examples

1. Locking whole object (linked list)
    - easy to reason about correctness
    - may give poor performance
2. Locking individual parts
    - may give better performance
    - more challenging to reason about correctness

# Queue Example

# An Array-Based Queue

```
public class LockedQueue<T> {
    int head, tail;
    T[] contents;
    Lock lock;
}
```

# Enqueuing

```
public void enq(T x) {
    lock.lock();
    try {
        items[tail] = x;
        tail++;
    } finally {
        lock.unlock();
    }
}
```

# Dequeueing

```
public T deq() {
    lock.lock();
    try {
        T x = items[head];
        head++;
        return x;
    } finally {
        lock.unlock();
    }
}
```

# Question

Why does this implementation give a FIFO queue?

# Executions are Essentially Sequential!

If multiple threads access the queue

- only one thread actually modifies queue at a time
- other threads must wait
- this property is called **blocking**:
  - some method calls cannot make progress while others perform a task

# Another Question

Queues are *first-in first-out* (FIFO) data structures

What does FIFO even *mean* if objects can be enqueued concurrently?

# Blocking Execution

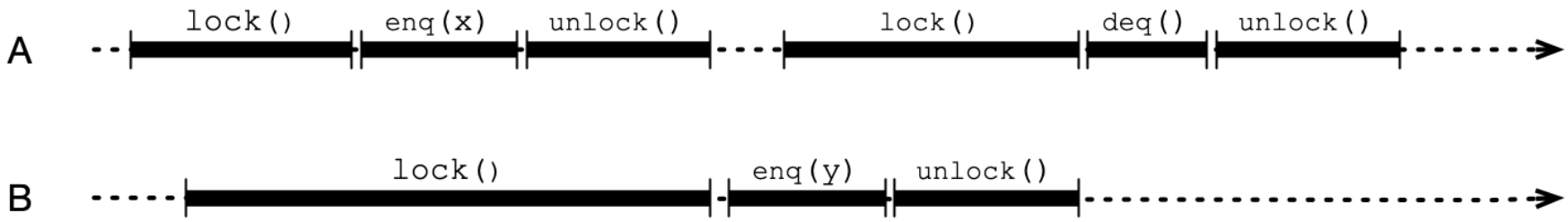
Consider:

- Two threads A, B concurrently call `enq(x)`, `enq(y)`, respectively
- Then thread A calls `deq()`

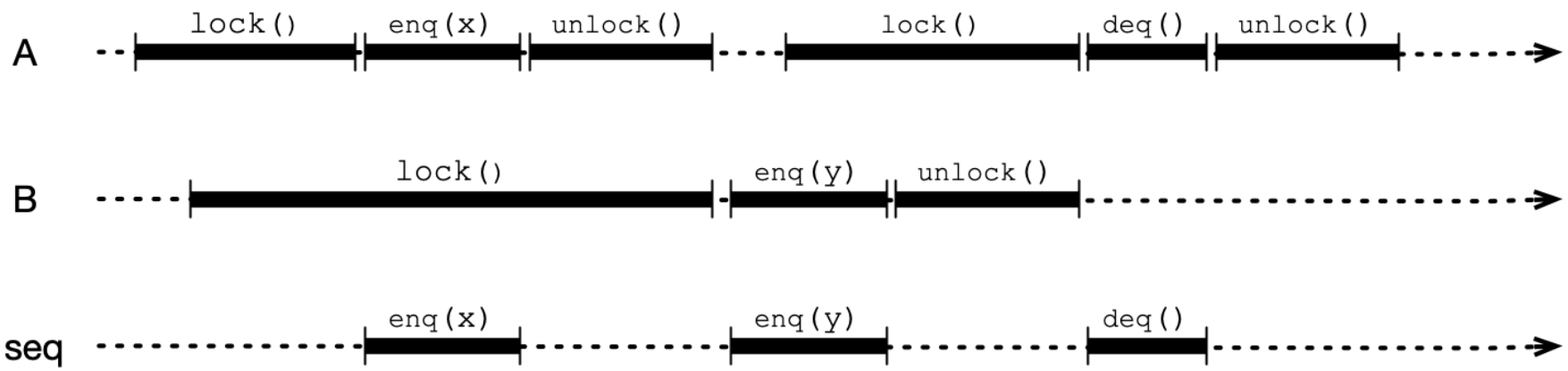
What is expected behavior?

# What Happens?

Depends on how concurrent operations are resolved!



# Equivalent Sequential Execution



# In Sequential Execution

Method calls are linearly sorted:

- Method calls:
  - invocation
  - response
- Each call's response preceeds next call's invocation

# Reasoning About Sequential Executions

1. Assume object in some state
   - **precondition**
2. Method specifies
   - **postcondition**
     - return value
     - change of internal state (side effect)

Method calls performed sequentially $\Longrightarrow$ state well defined *between* method calls.

- Specifying pre/post-conditions for each method define object's **sequential specification**

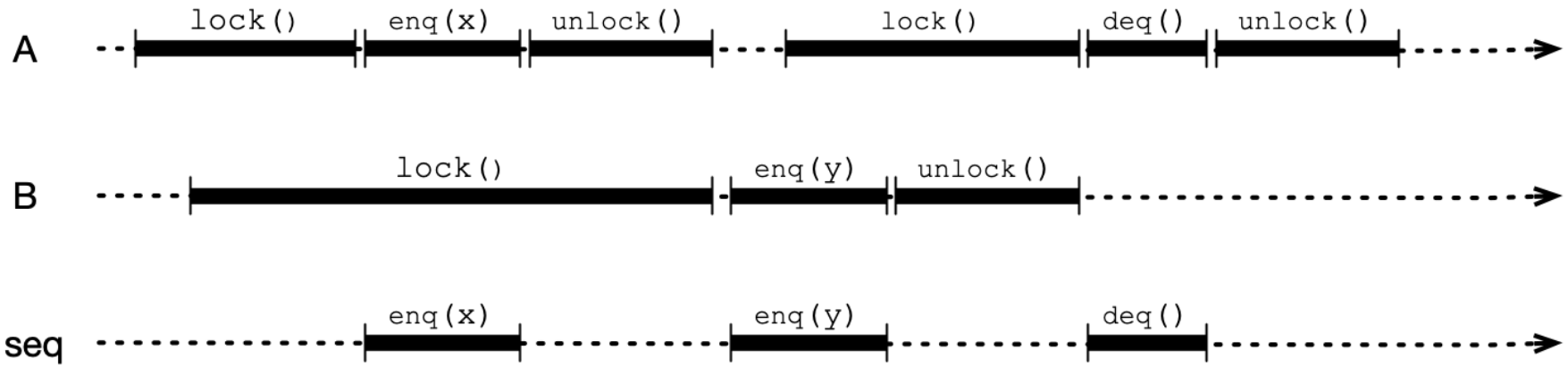
# Reasoning About Concurrent Executions

"Correct" behavior no longer well defined!


- call to `deq()` could return either `x` or `y`
  - both reasonable!

# A Reasonable Goal

A concurrent execution of a data structure is “correct” if it is consistent with *some* sequential execution of the data structure.




Response to each method call in concurrent execution is the same as the sequential execution.

- What other features of concurrent execution can/should the sequential execution maintain?

# Our Goal

Define sensible qualities for how executions should behave:

1. Sequential consistency
2. Linearizability

These are less rigid requirements than being essentially sequential

- May allow for less synchronization (locking) between threads
- Tradeoff: more lenient behavioral guarantees

# Sequential Consistency

# A Sensible Feature

Consider all method calls made by all threads

- Each method has precondition, postcondition

Behavior of execution should be consistent with *some* sequential execution of the method calls.

# Is This Enough?

Behavior of execution should be consistent with *some* sequential execution of the method calls.

# Probably Not!

Queue with multiple threads:

- thread 1 calls `enq(1)` then `enq(2)`
- other threads enqueue stuff, not 1 or 2
- thread 1 calls `deq()` a bunch of times

Should have:

- thread 1 dequeues 1 before 2

# Another Sensible Feature

Method calls should appear to take effect in **program order**

- if a single thread calls `methodOne()` before `methodTwo()`, then `methodOne()` should take effect before `methodTwo()` in sequential execution.

# Sequential Consistency

An execution is **sequentially consistent** if all method calls can be ordered such that:

1. they are consistent with program order
2. they meet object's sequential specification

An implementation of an object is sequentially consistent if

1. it guarantees *every* execution is sequentially consistent

# Example

What are possible outcomes of `deq()` calls in a sequentially consistent execution?