

Inexact Software Is the Solution

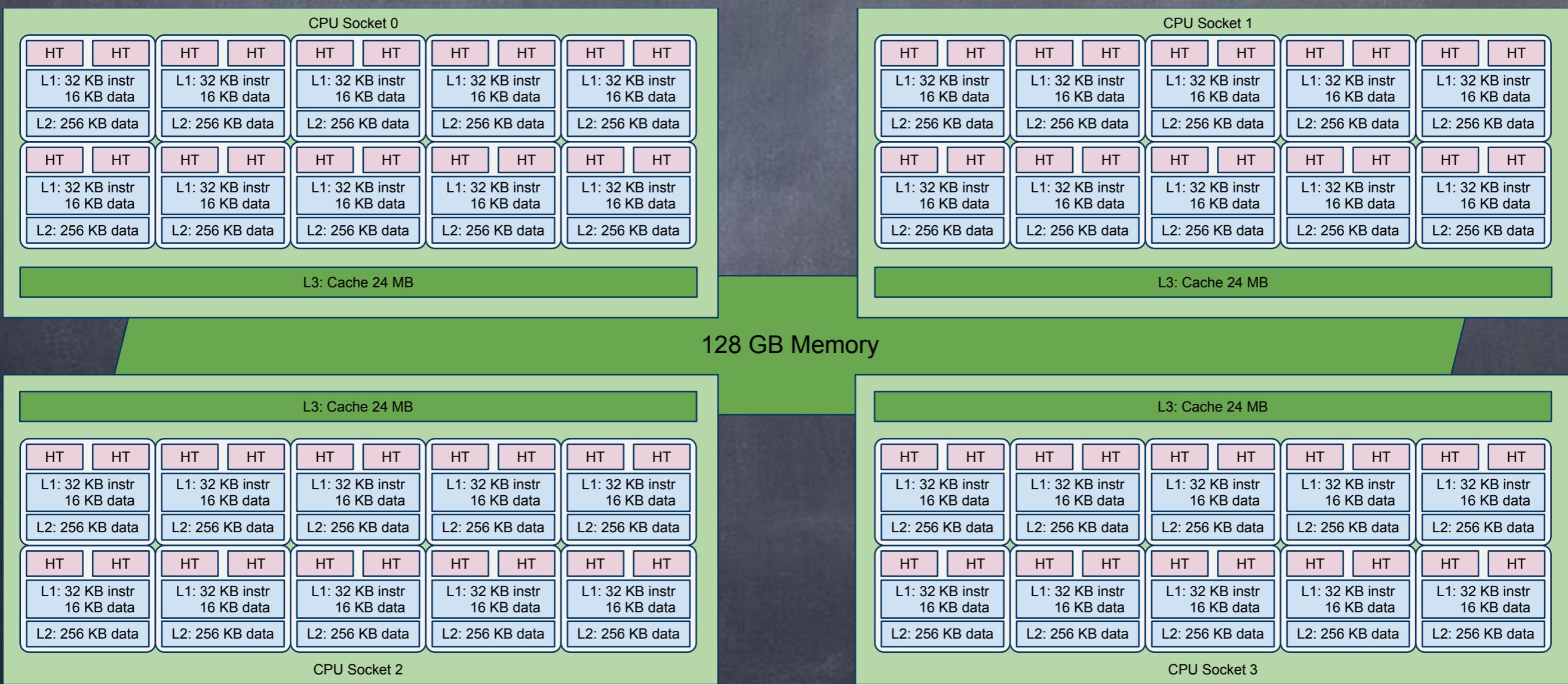
Christoph Kirsch
Universität Salzburg



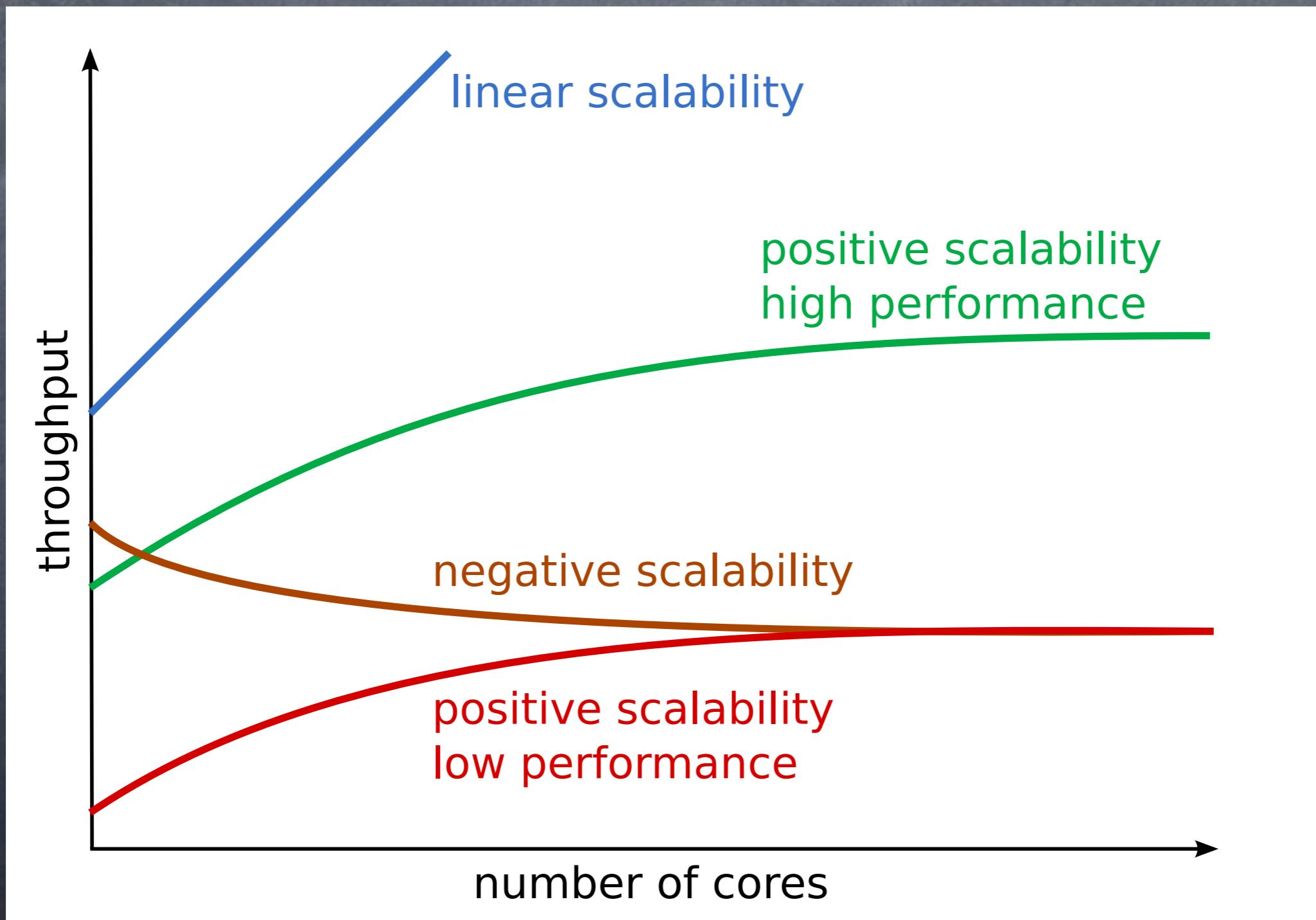
CASA Workshop, Tampere, Finland, October 2012

Joint work w/ A. Haas,
M. Lippautz, H. Payer,
H. Röck, A. Sokolova and our
collaborators at IST Austria
T. Henzinger, A. Sezgin

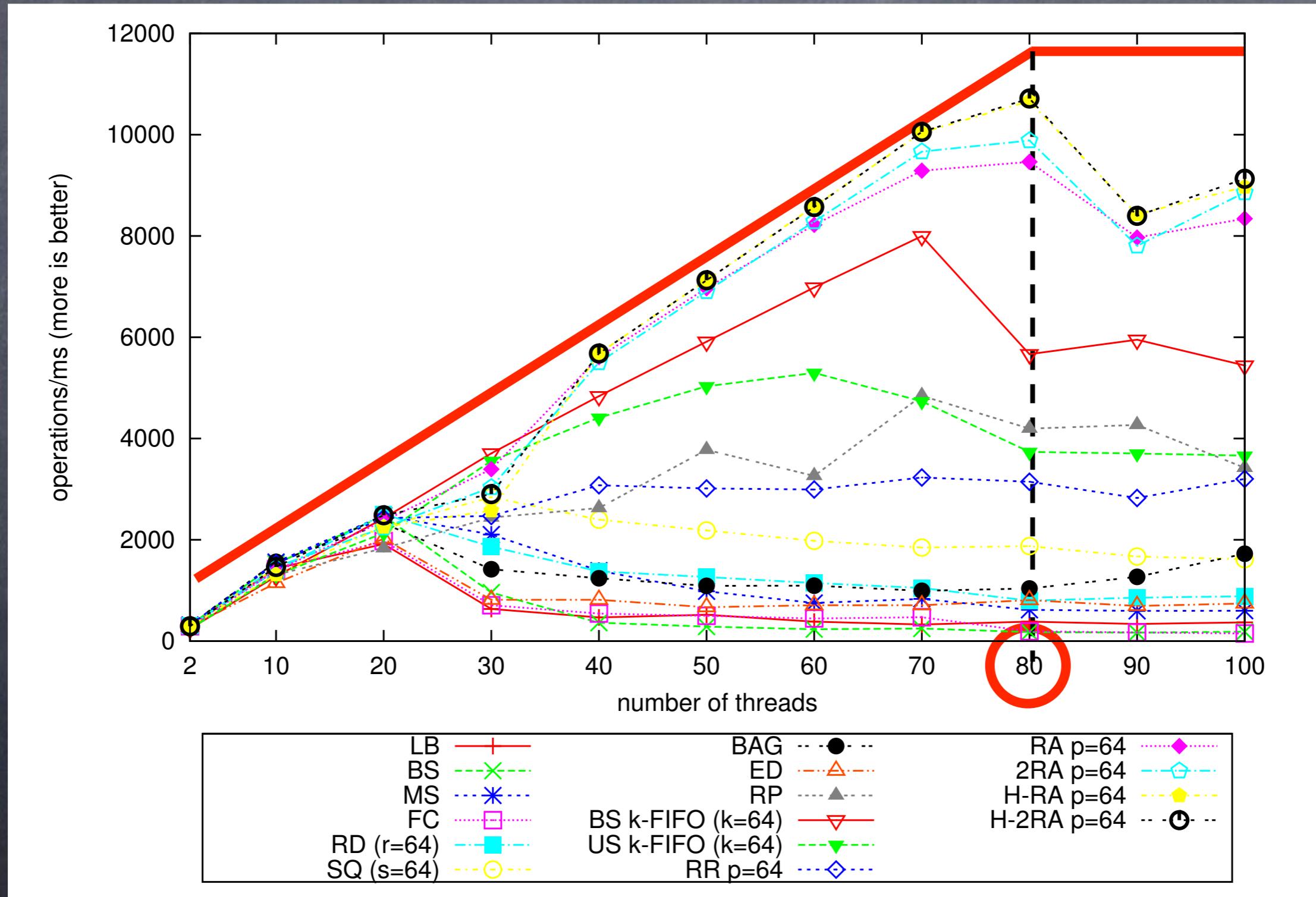
4 processors × 10 cores ×
 2 hardware threads =
 80 hardware threads



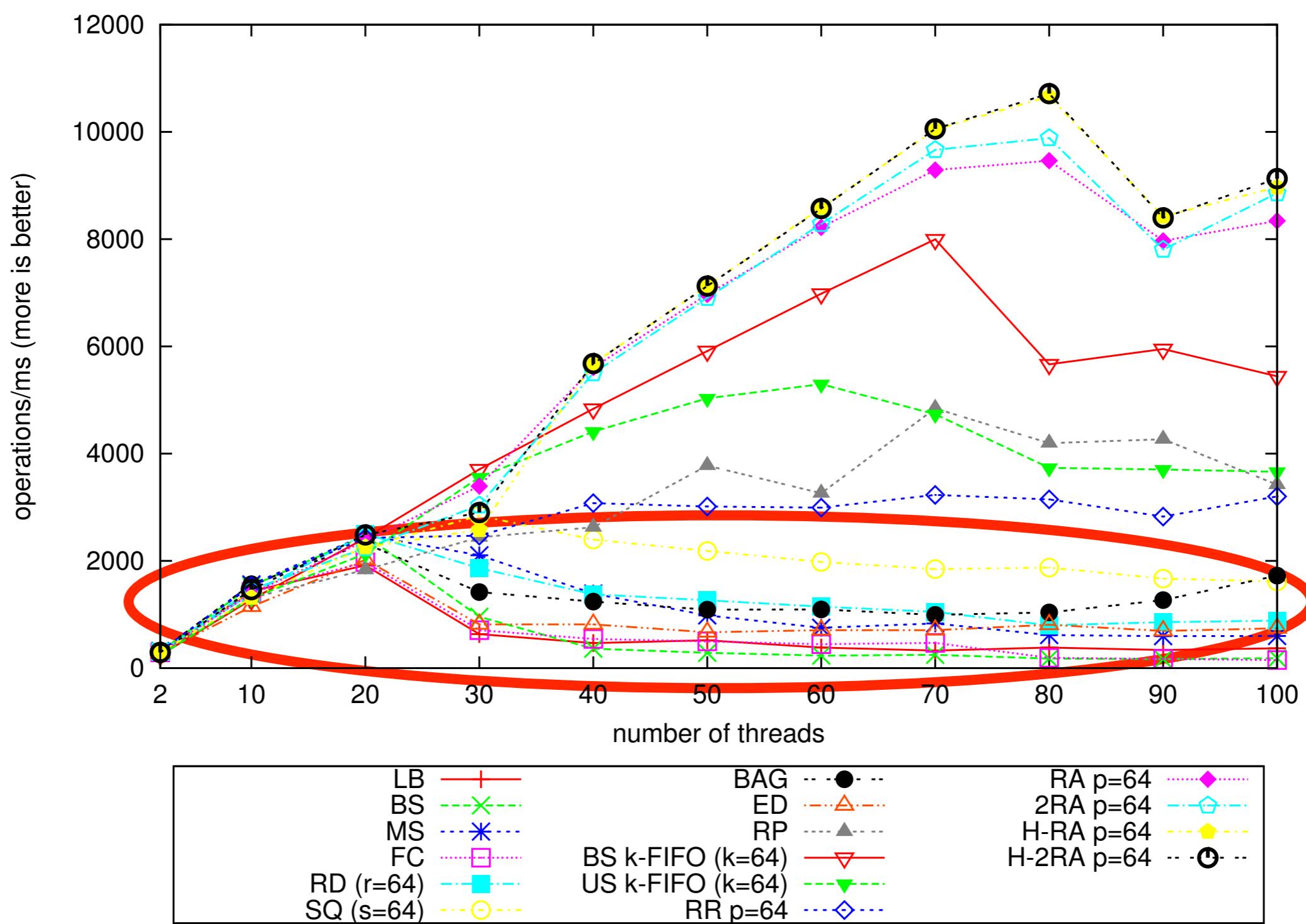
Performance & Scalability



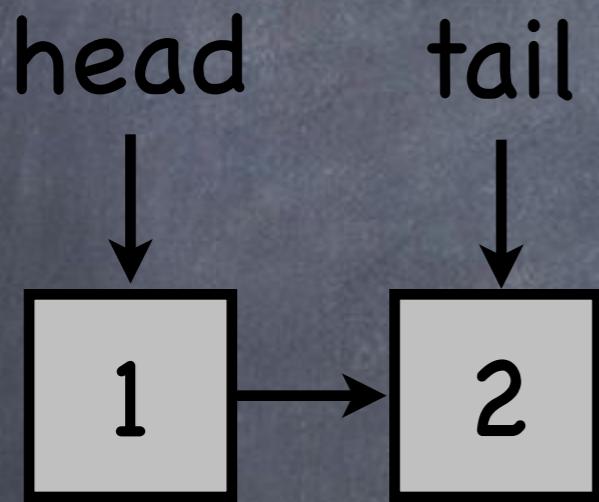
Ideal 80-Thread Performance



Regular FIFO Queues



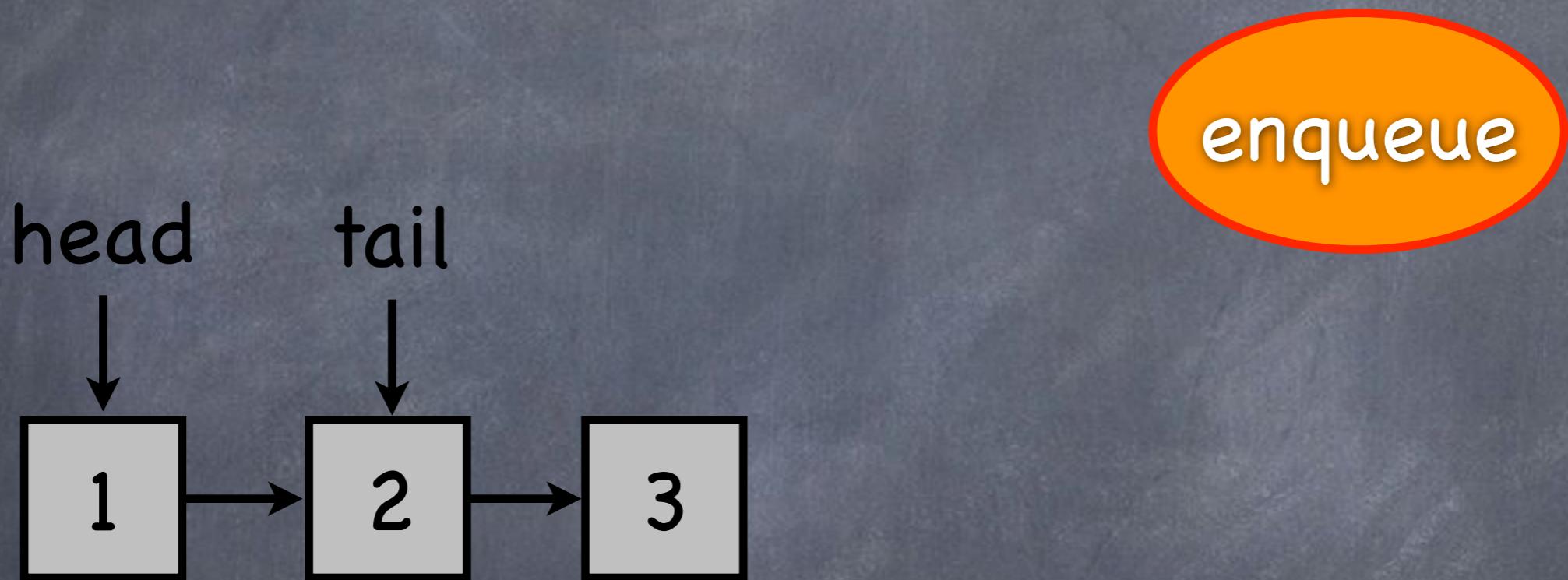
Concurrent First-in- First-out (FIFO) Queue



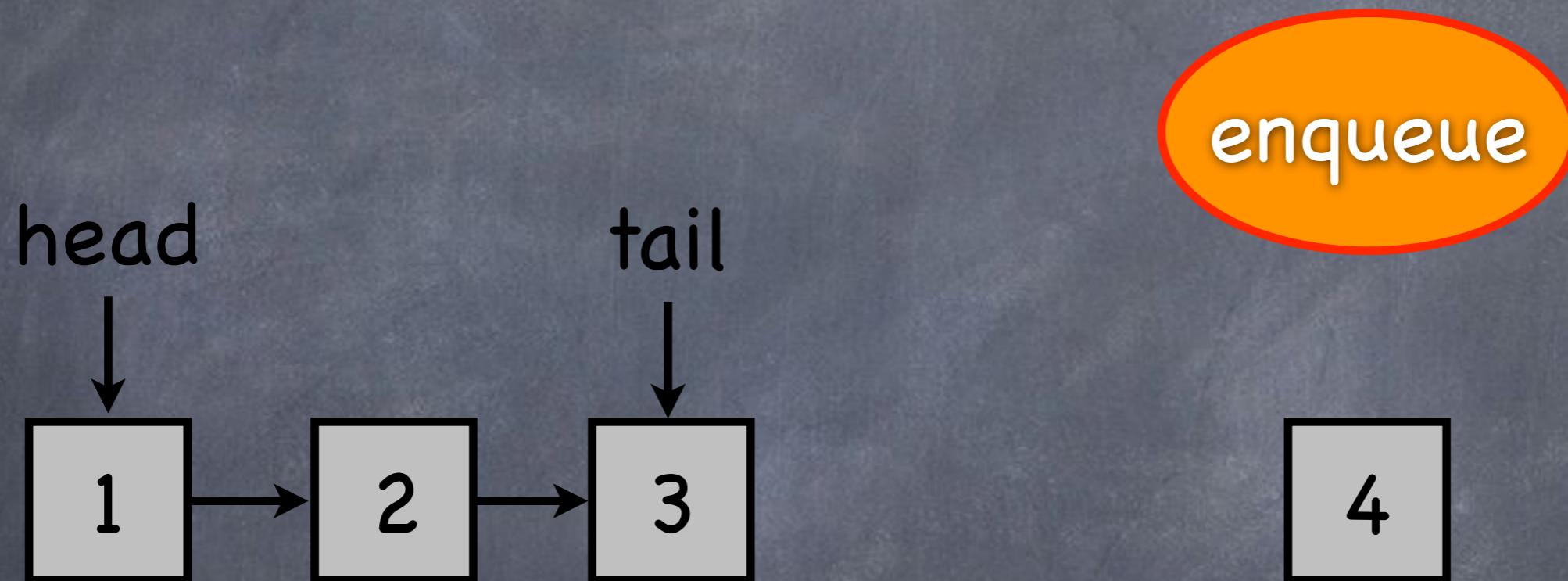
Concurrent First-in-First-out (FIFO) Queue



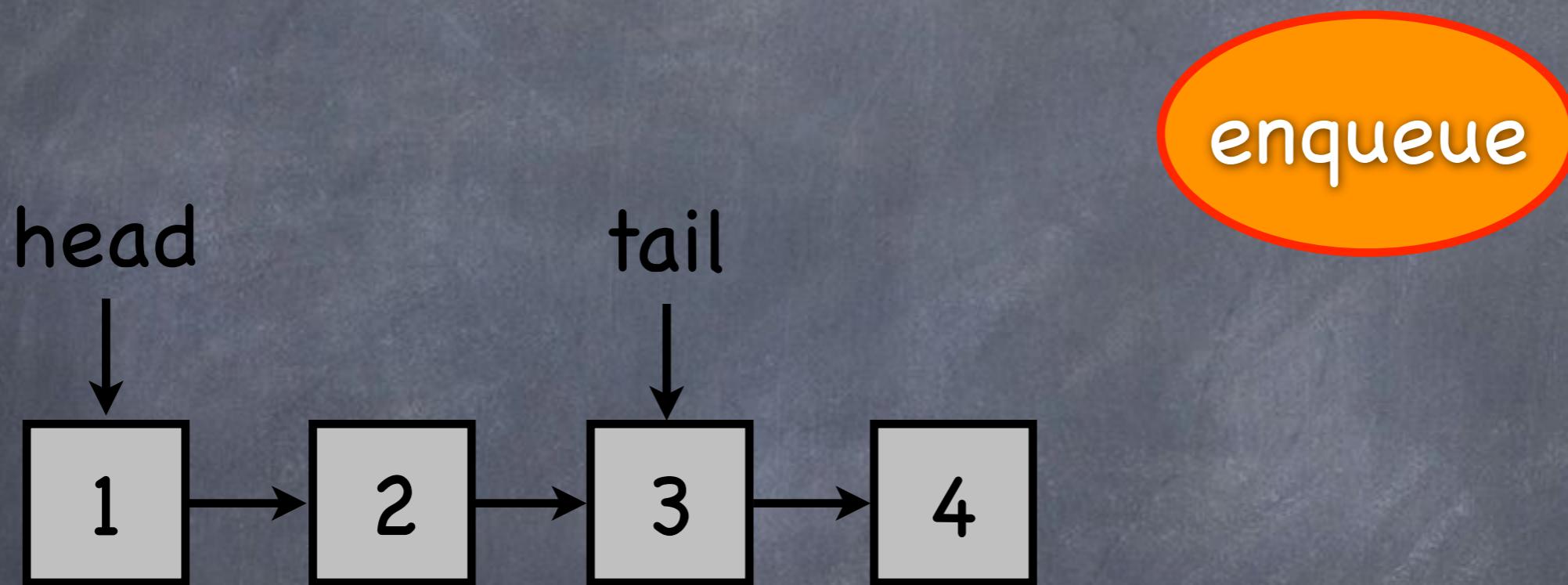
Concurrent First-in-First-out (FIFO) Queue



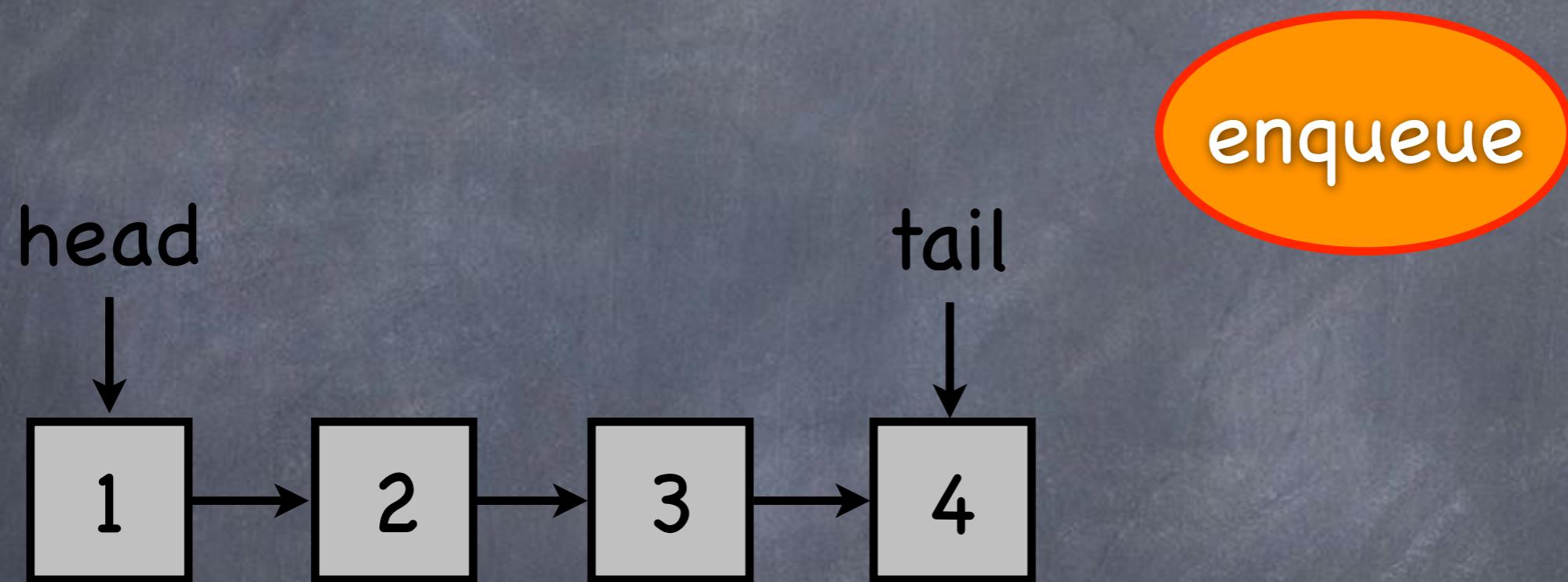
Concurrent First-in-First-out (FIFO) Queue



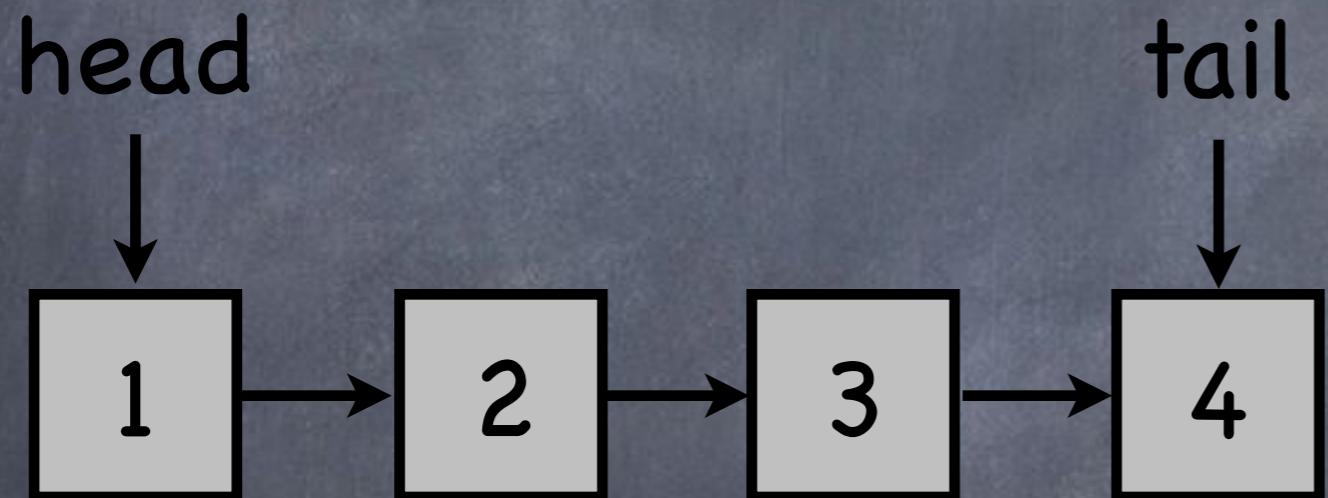
Concurrent First-in-First-out (FIFO) Queue



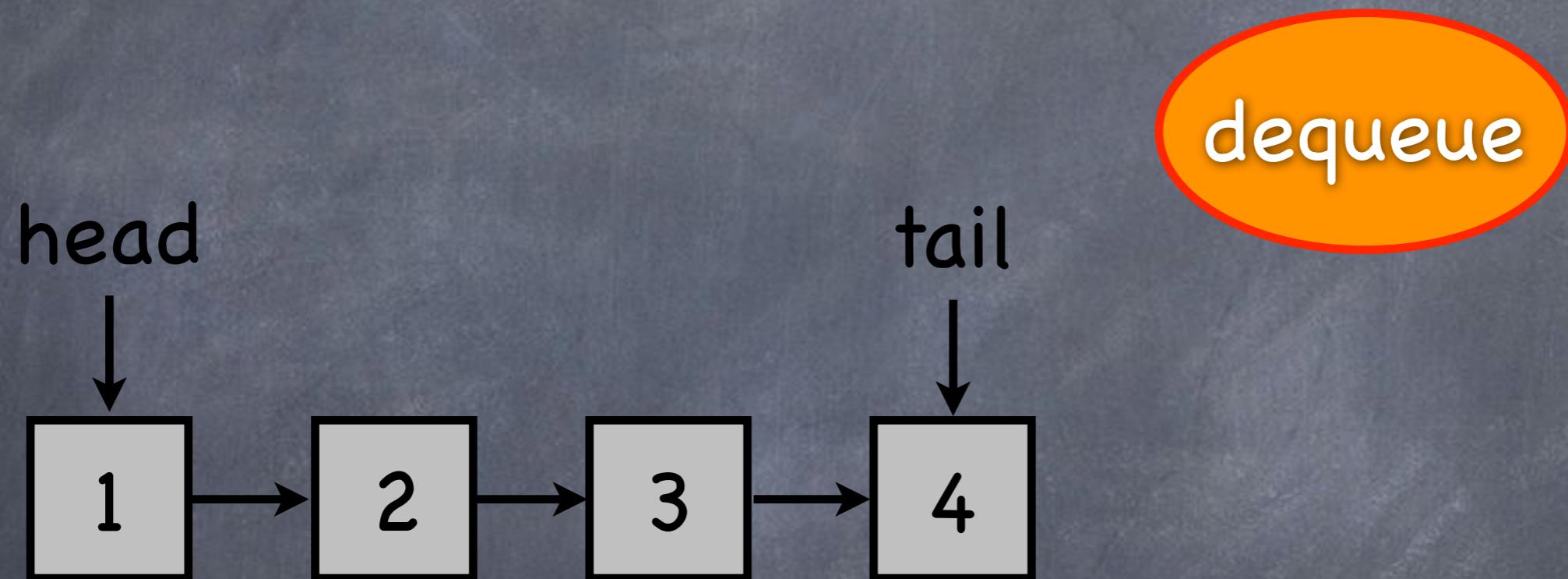
Concurrent First-in-First-out (FIFO) Queue



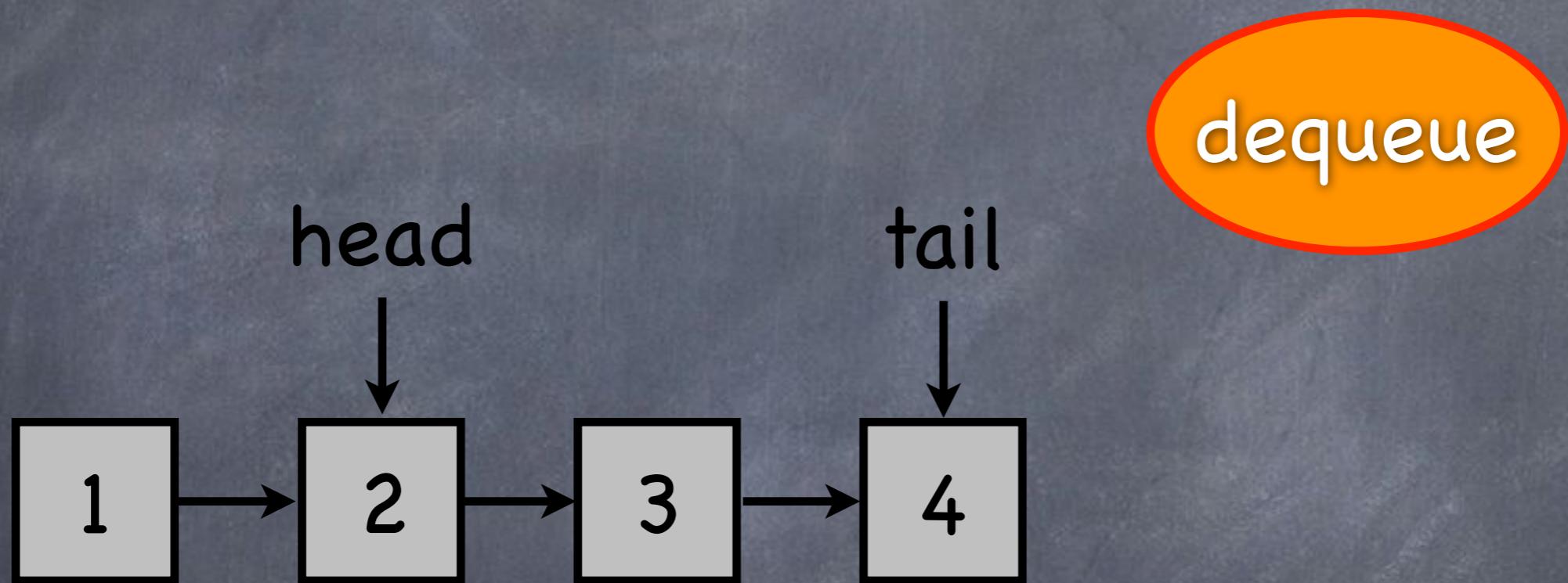
Concurrent First-in- First-out (FIFO) Queue



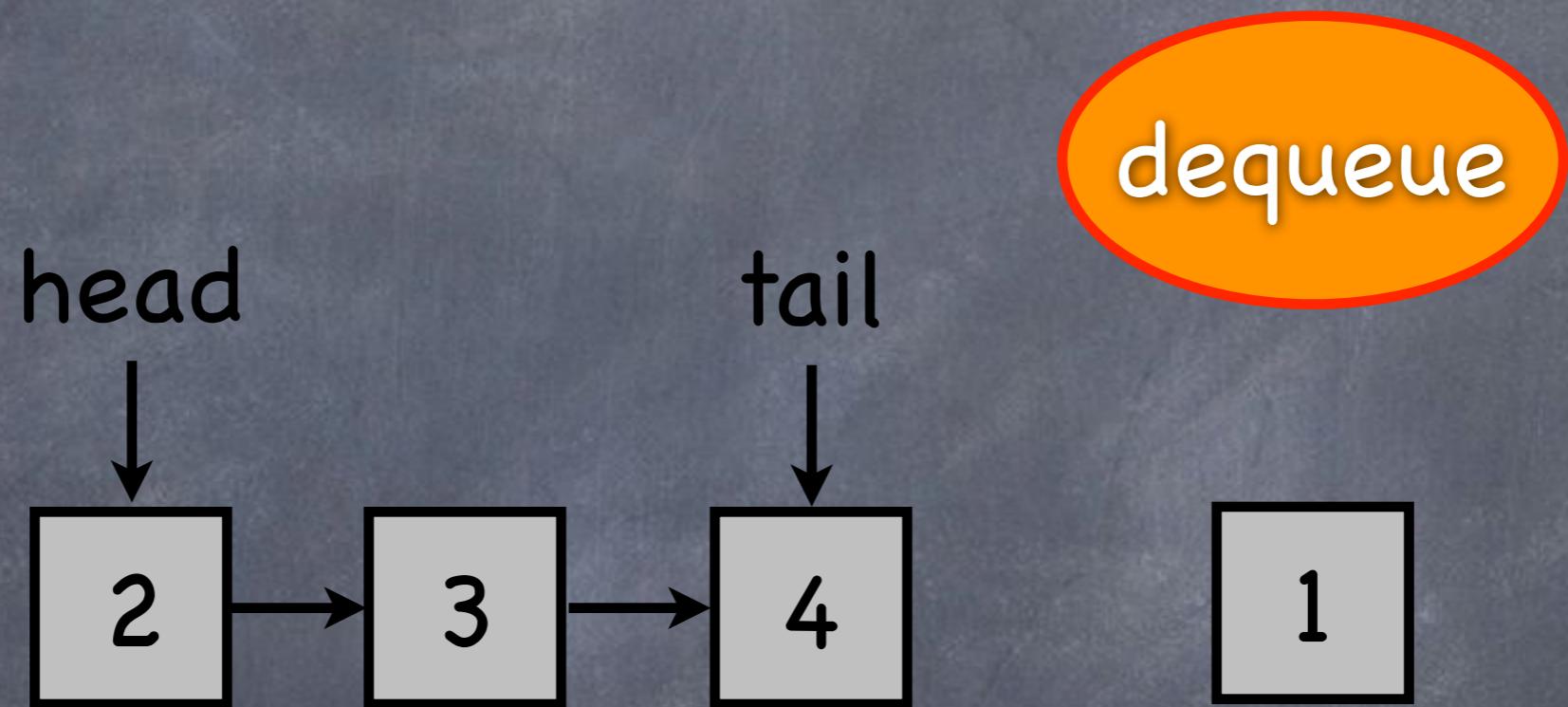
Concurrent First-in-First-out (FIFO) Queue



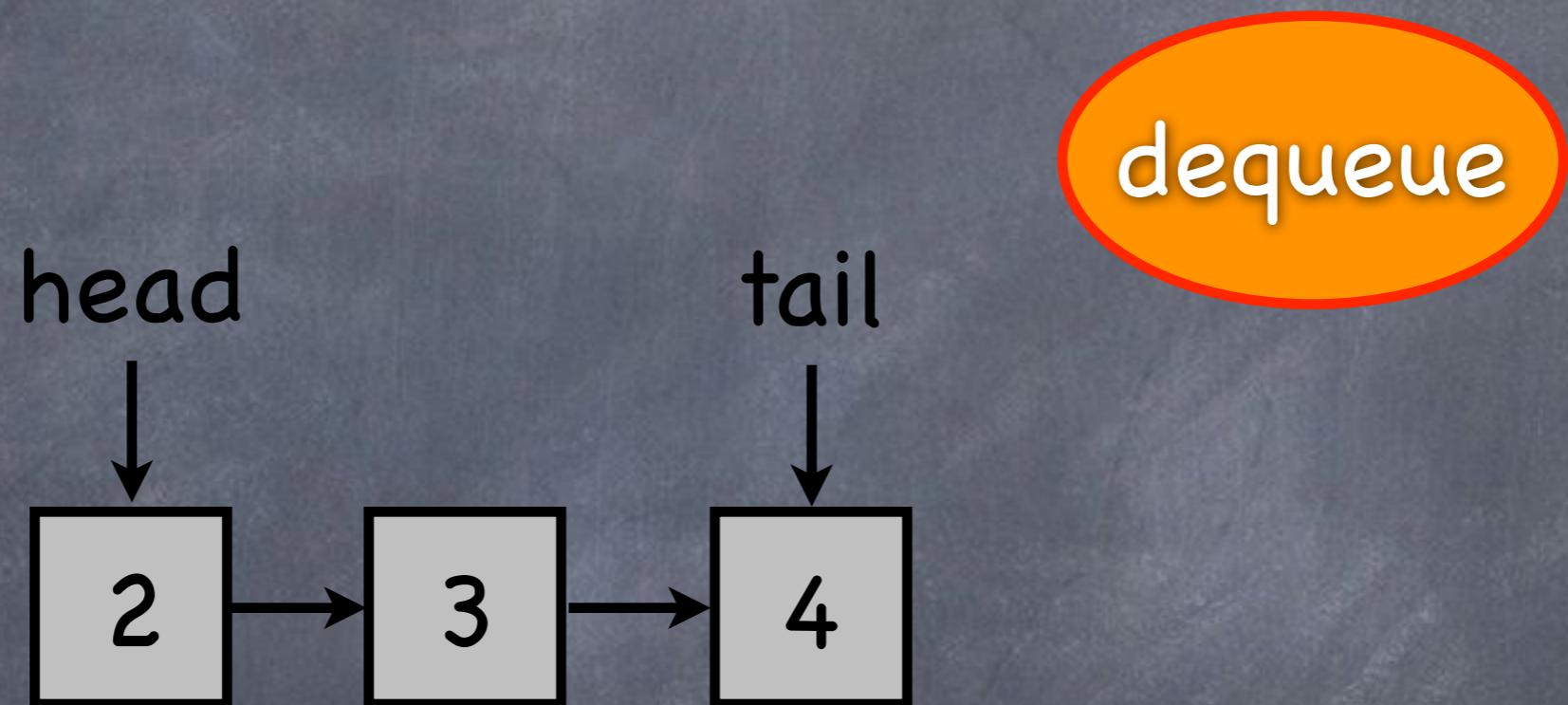
Concurrent First-in-First-out (FIFO) Queue



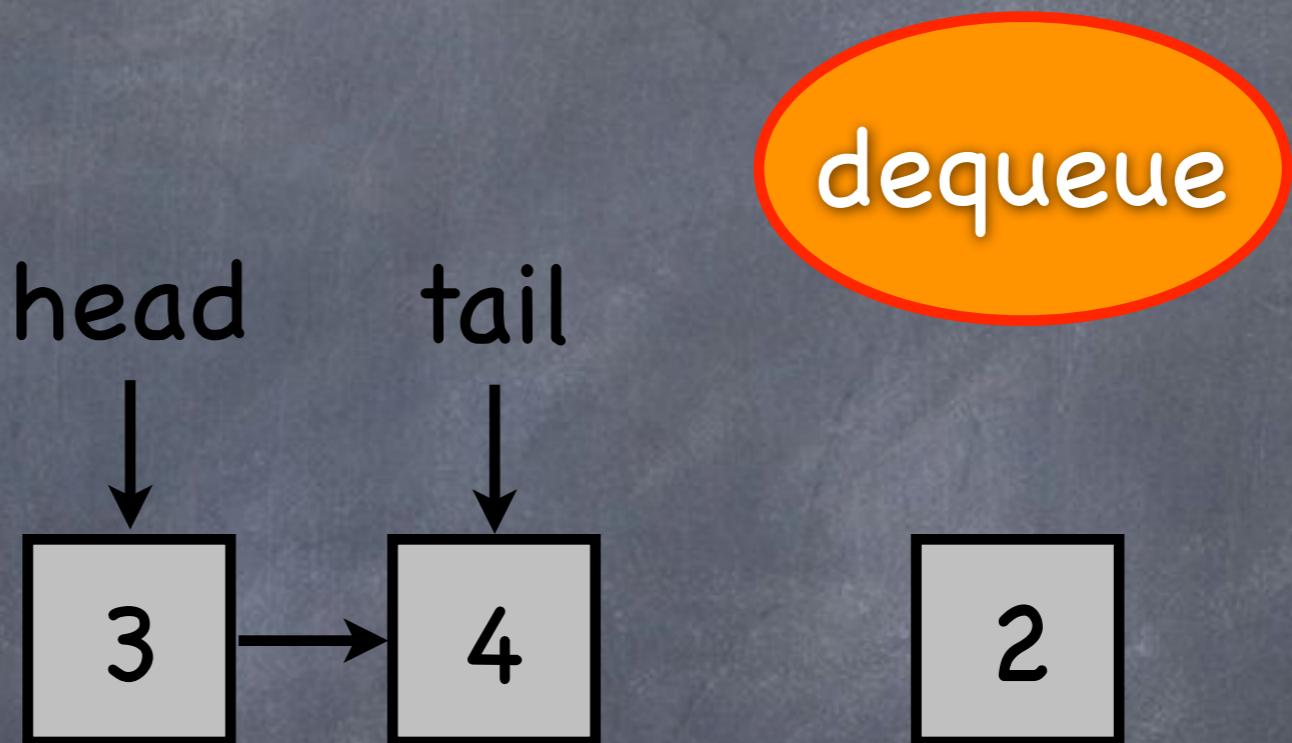
Concurrent First-in-First-out (FIFO) Queue



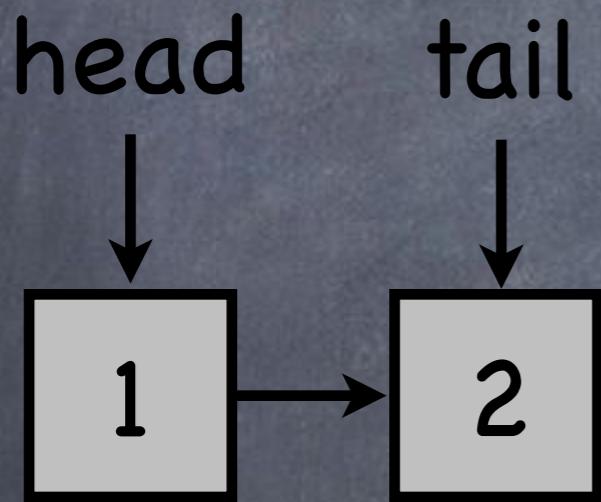
Concurrent First-in-First-out (FIFO) Queue



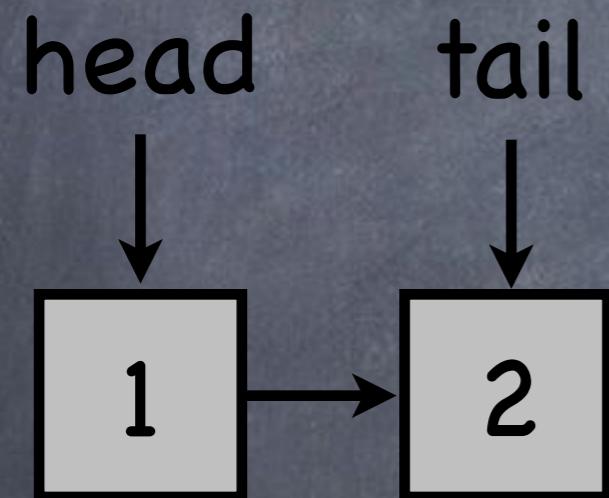
Concurrent First-in-First-out (FIFO) Queue



Concurrent First-in- First-out (FIFO) Queue

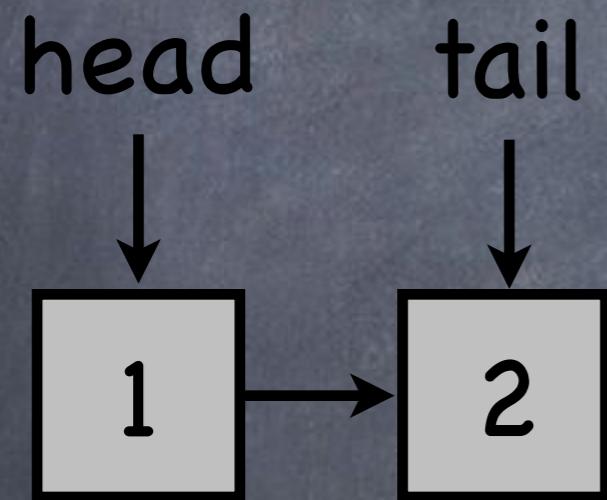


Concurrent First-in- First-out (FIFO) Queue



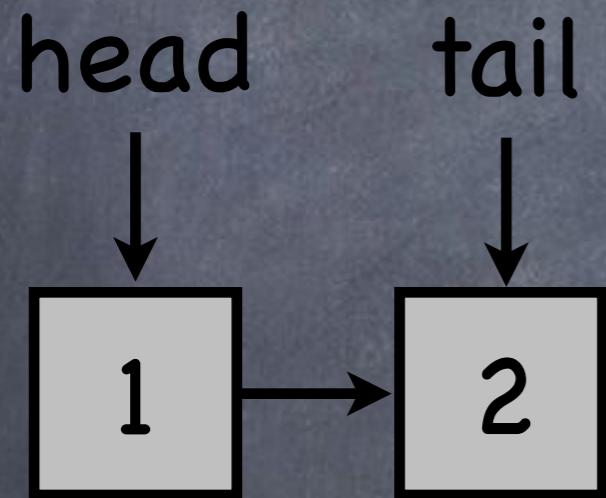
-> 1 lock

Concurrent First-in- First-out (FIFO) Queue



-> 1 lock -> 2 locks

Concurrent First-in-First-out (FIFO) Queue



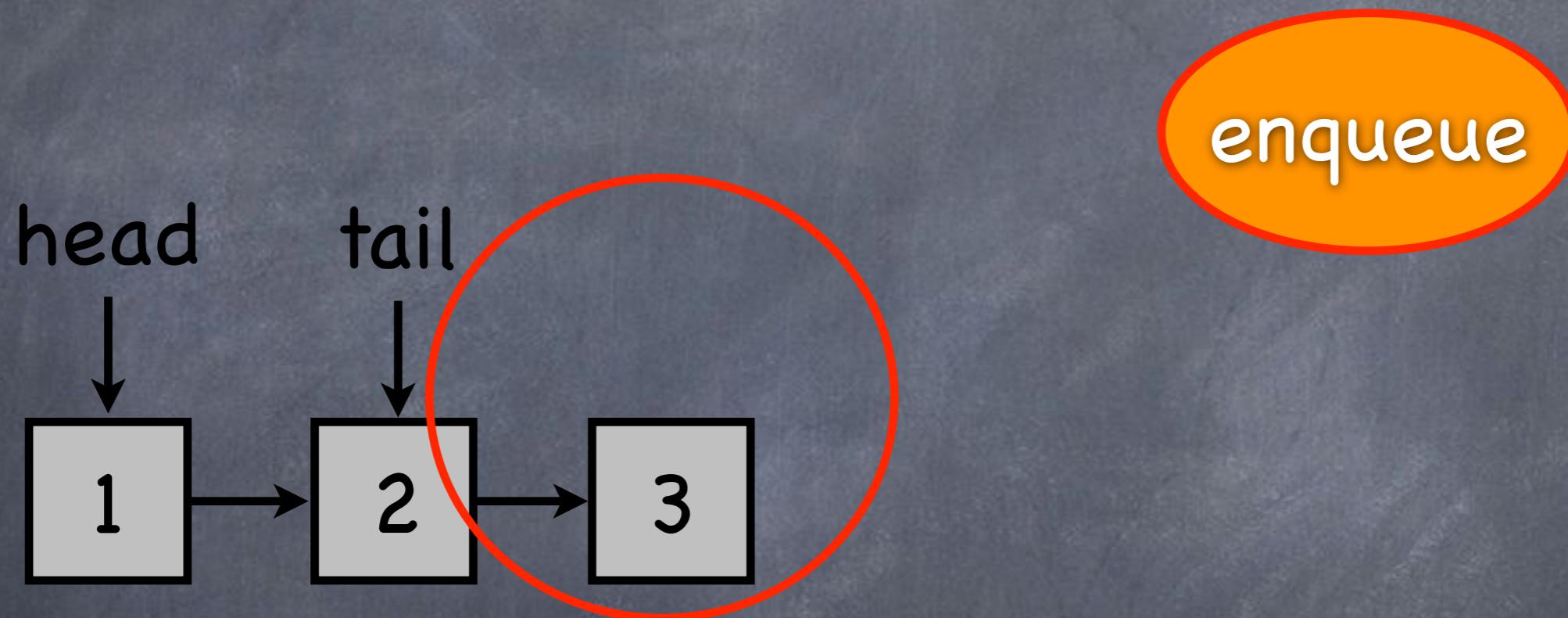
-> 1 lock -> 2 locks -> 0 locks

Concurrent First-in-First-out (FIFO) Queue



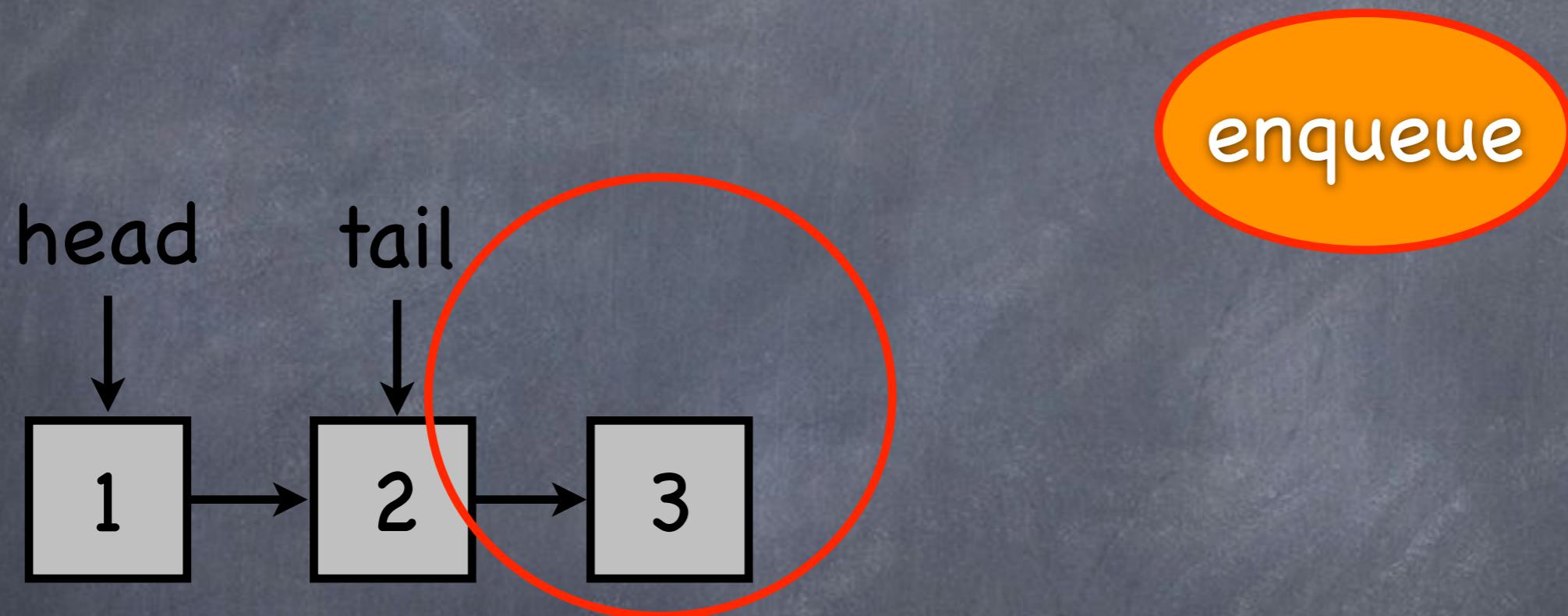
-> 1 lock -> 2 locks -> 0 locks

Concurrent First-in-First-out (FIFO) Queue



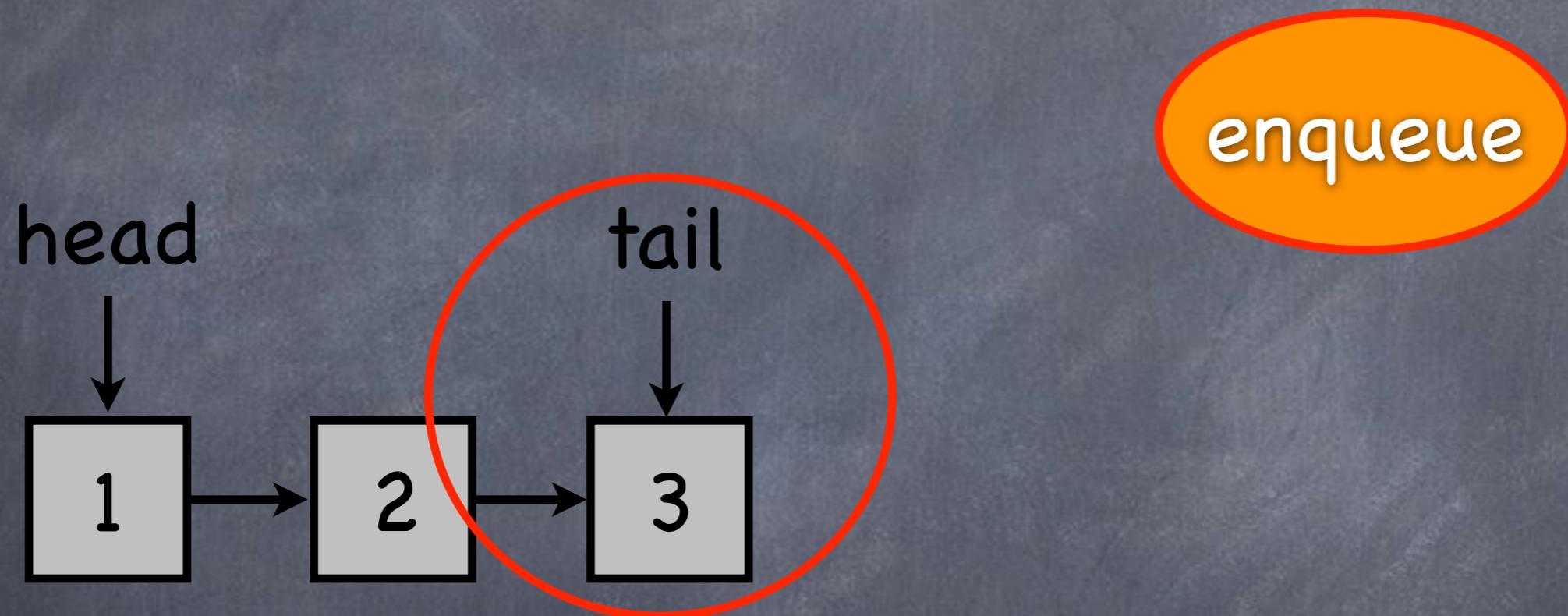
-> 1 lock -> 2 locks -> 0 locks

Concurrent First-in-First-out (FIFO) Queue



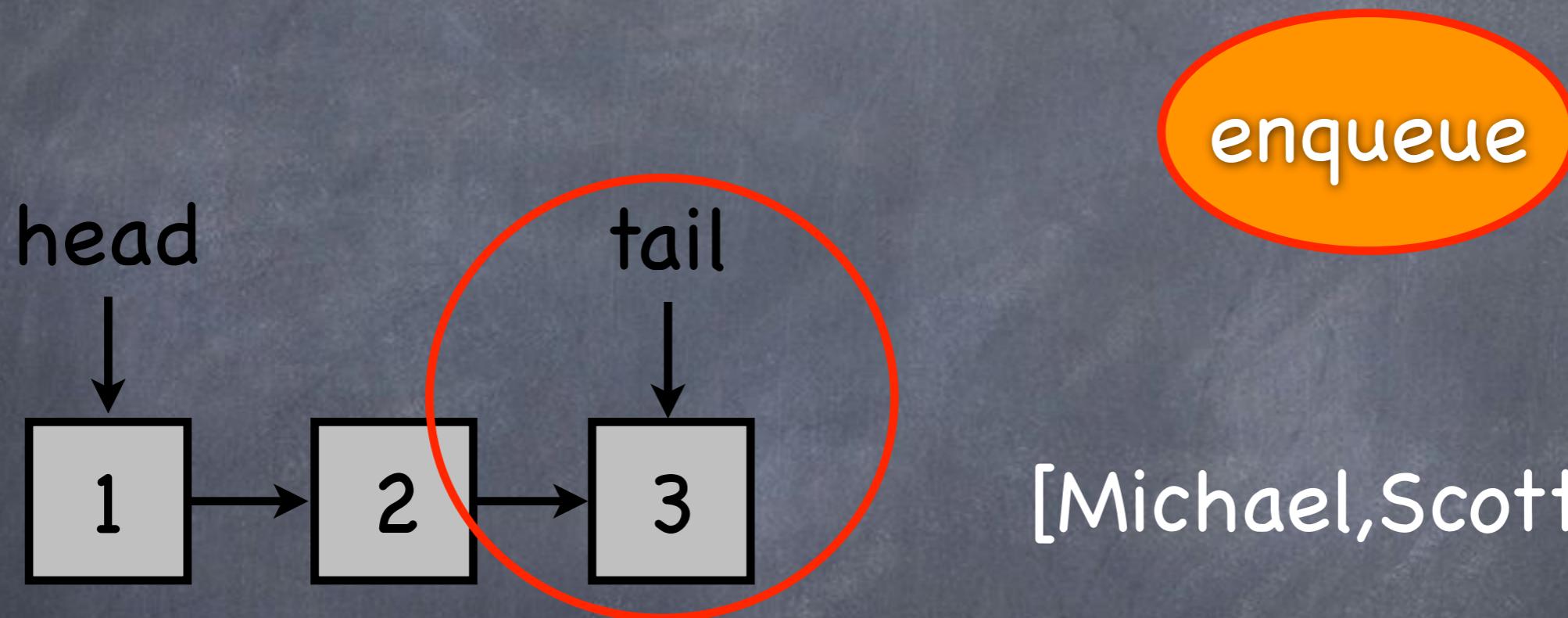
-> 1 lock -> 2 locks -> 0 locks -> compare & swap

Concurrent First-in-First-out (FIFO) Queue



-> 1 lock -> 2 locks -> 0 locks -> compare & swap

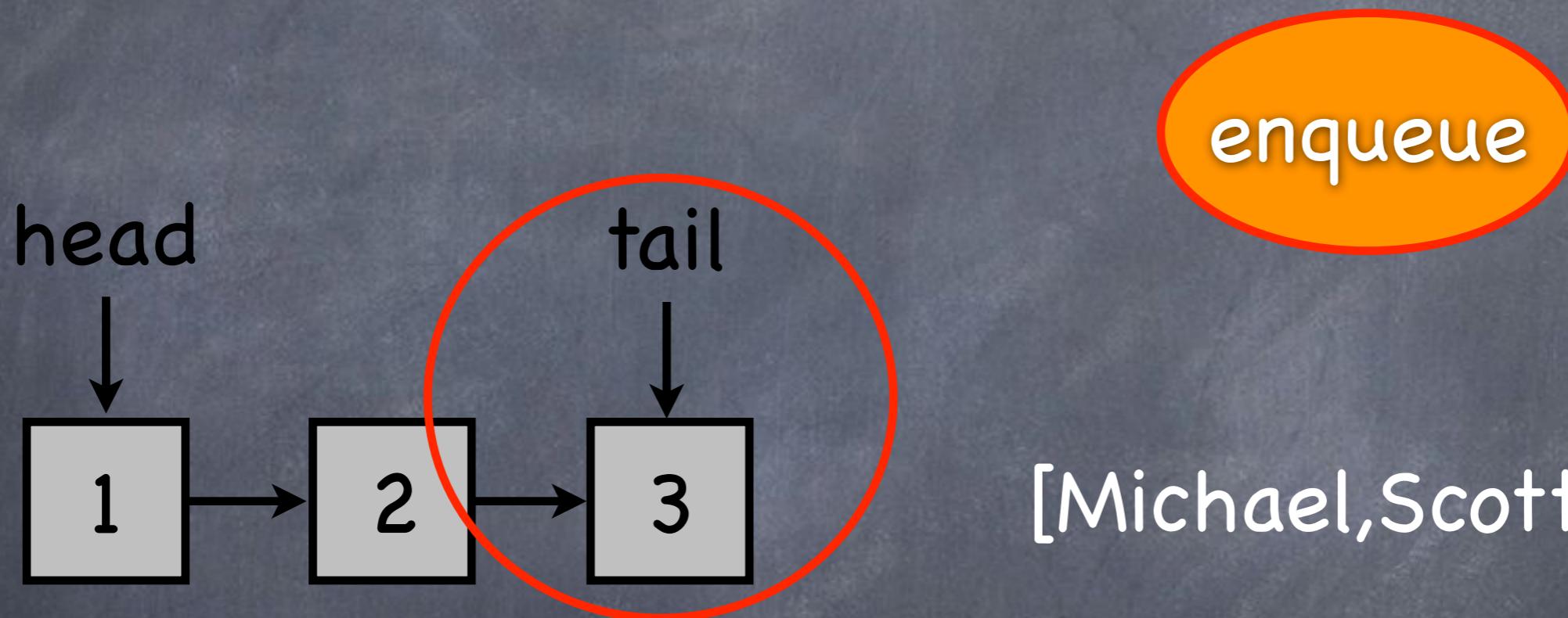
Concurrent First-in-First-out (FIFO) Queue



[Michael, Scott '96]

-> 1 lock -> 2 locks -> 0 locks -> compare & swap

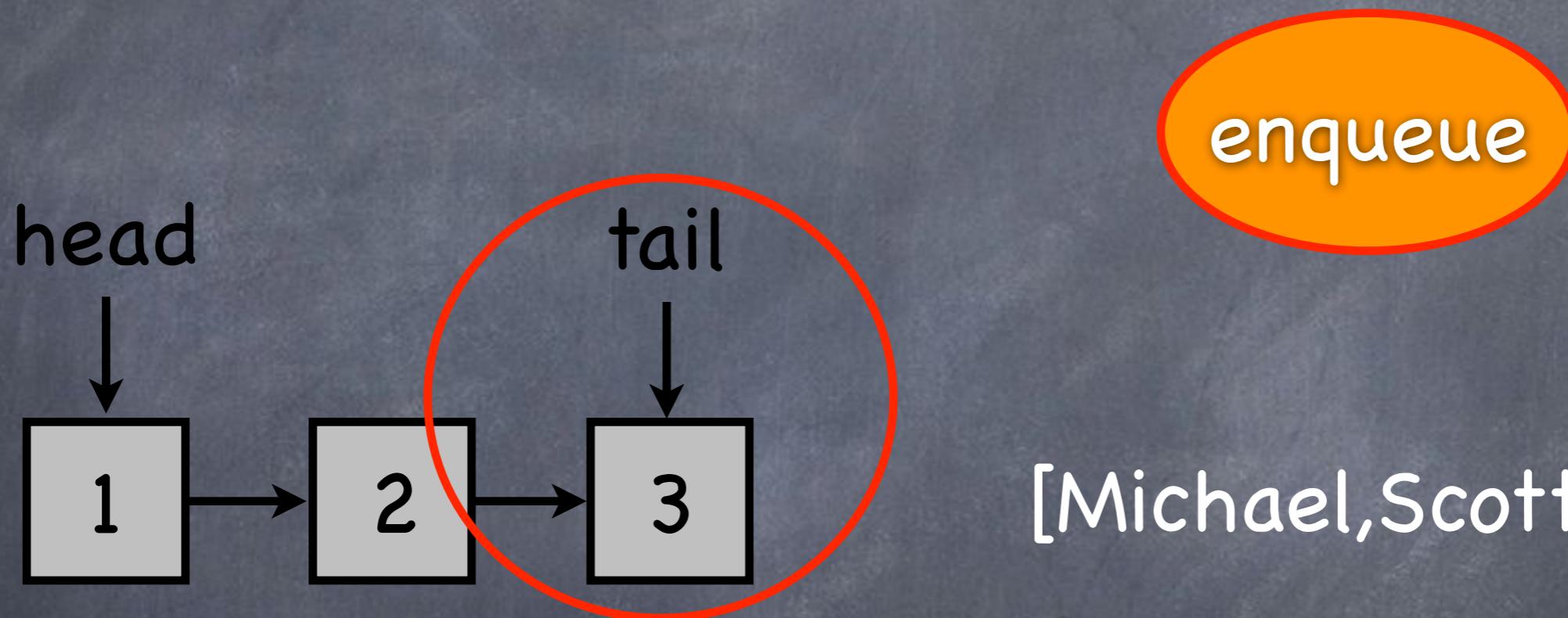
Concurrent First-in-First-out (FIFO) Queue



[Michael, Scott '96]

- > 1 lock
- > 2 locks
- > 0 locks
- > compare & swap
- > lock-based vs. **lock-free** vs. wait-free?

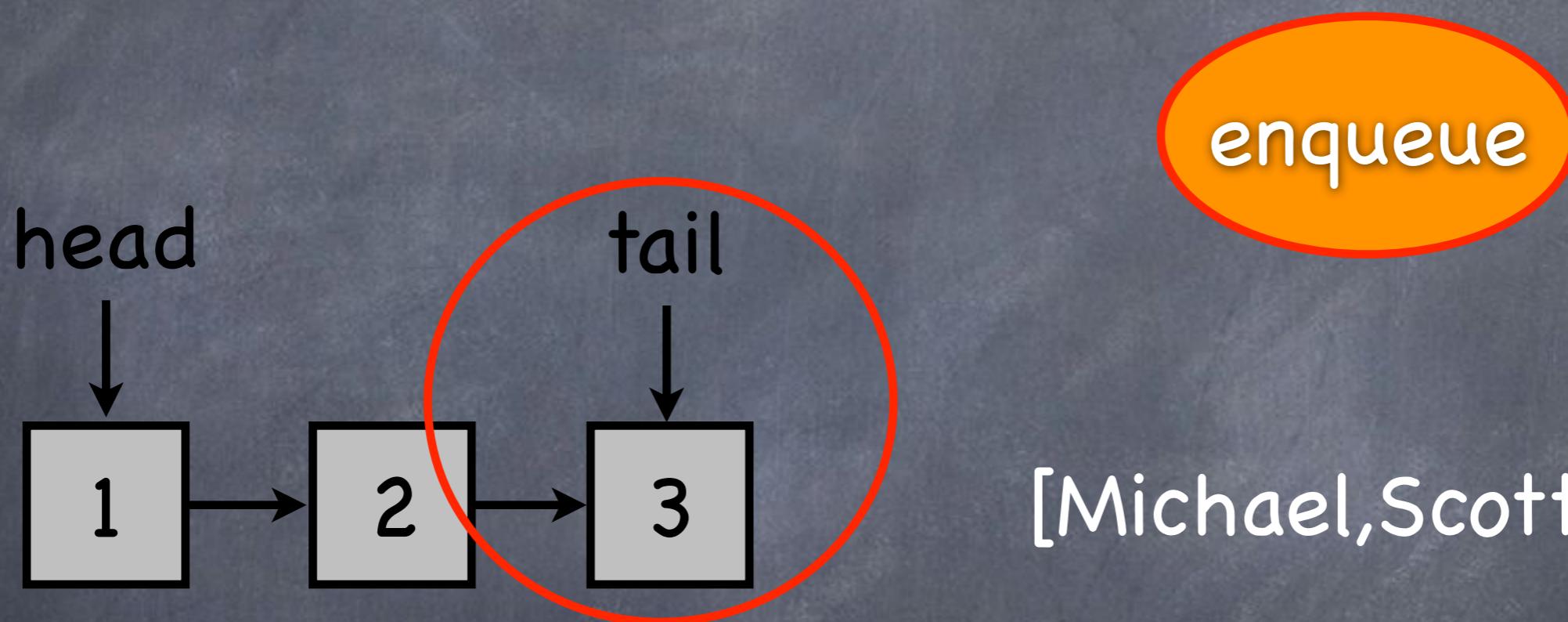
Concurrent First-in-First-out (FIFO) Queue



[Michael, Scott '96]

- > 1 lock -> 2 locks -> 0 locks -> compare & swap
- > lock-based vs. **lock-free** vs. wait-free?
- > memory contention on **head** and **tail** pointers!

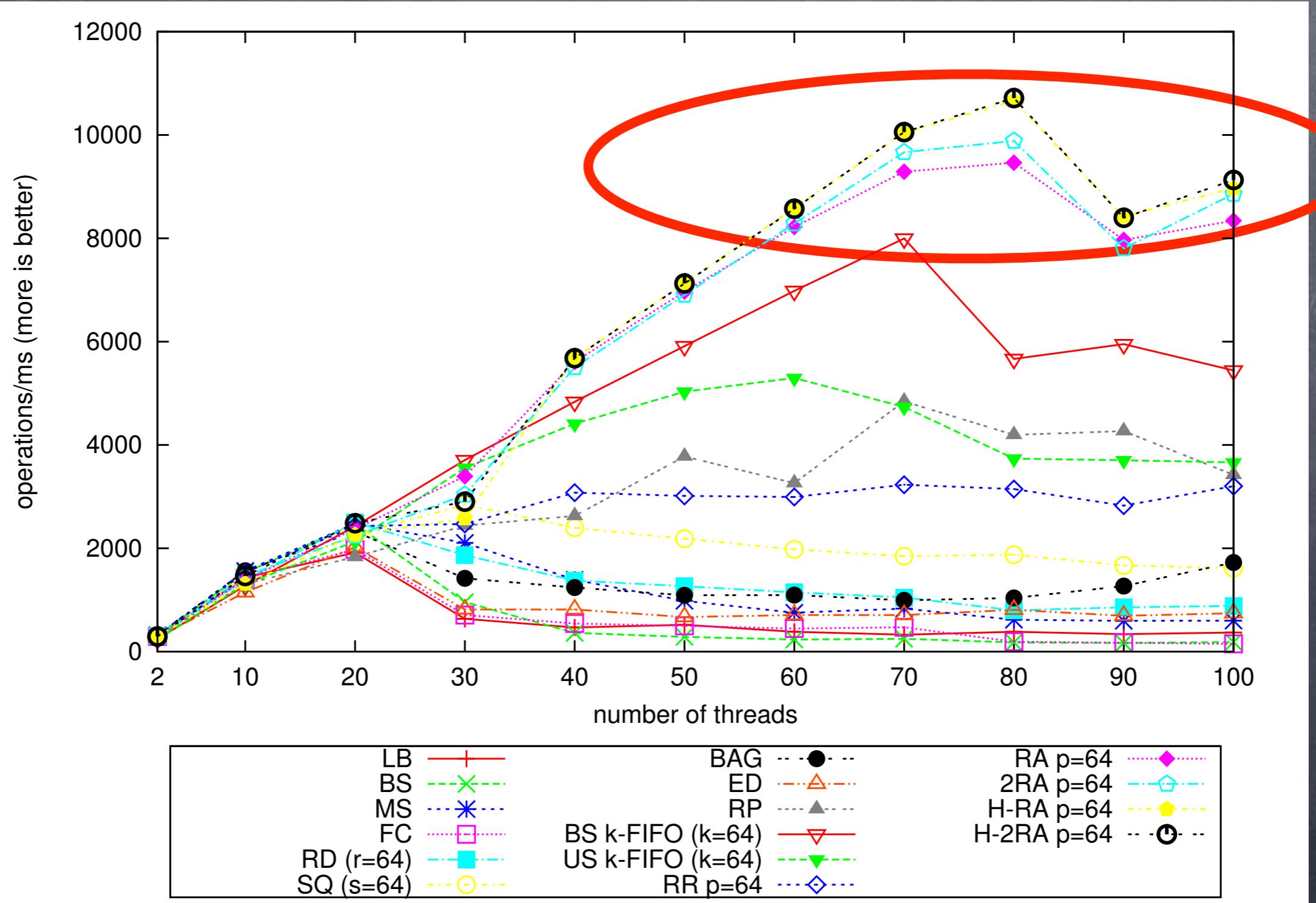
Concurrent First-in-First-out (FIFO) Queue



[Michael, Scott '96]

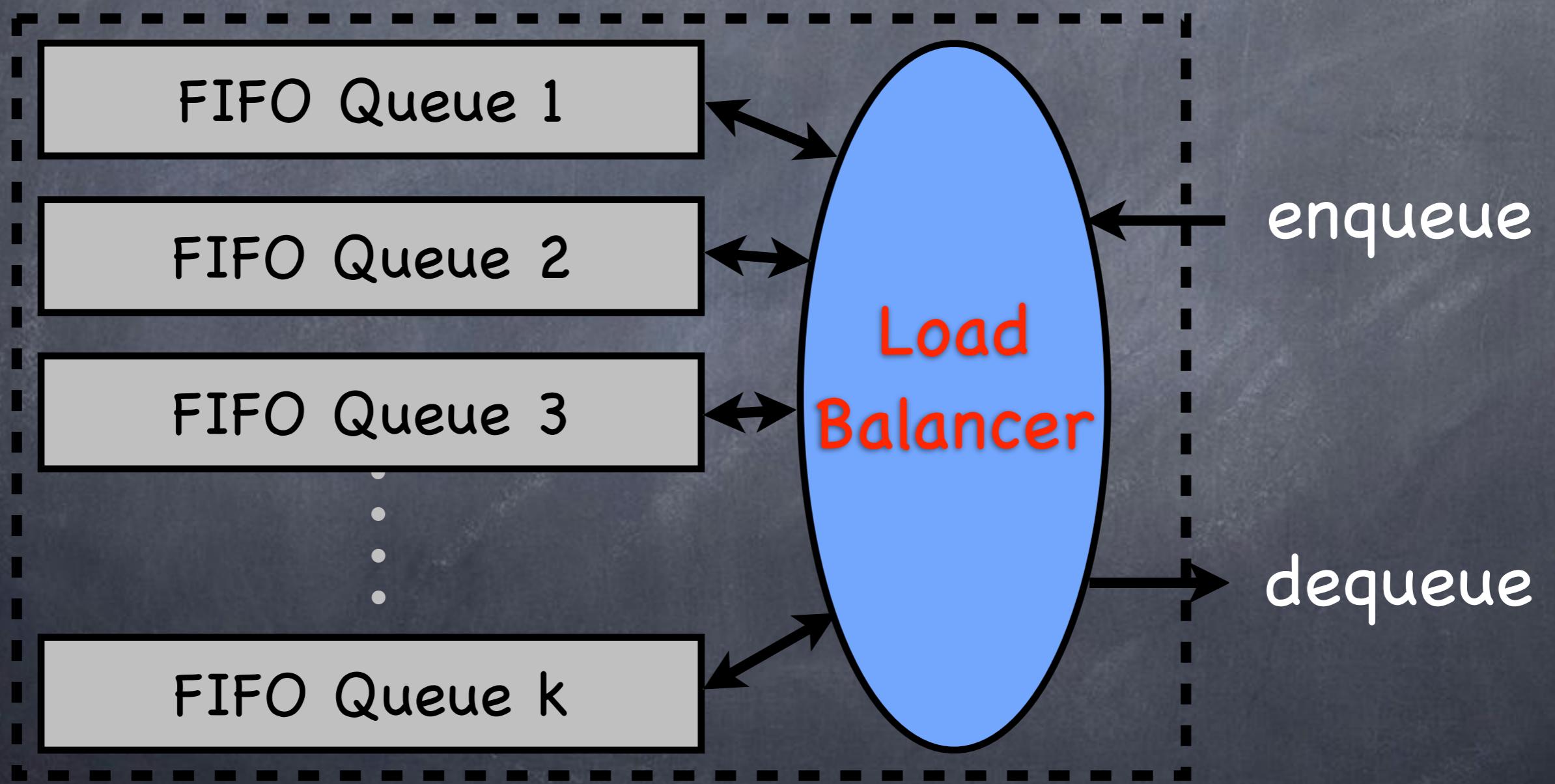
- > 1 lock -> 2 locks -> 0 locks -> compare & swap
- > lock-based vs. **lock-free** vs. wait-free?
- > memory contention on **head** and **tail** pointers!
- > and on **next** pointers!

Distributed Queues

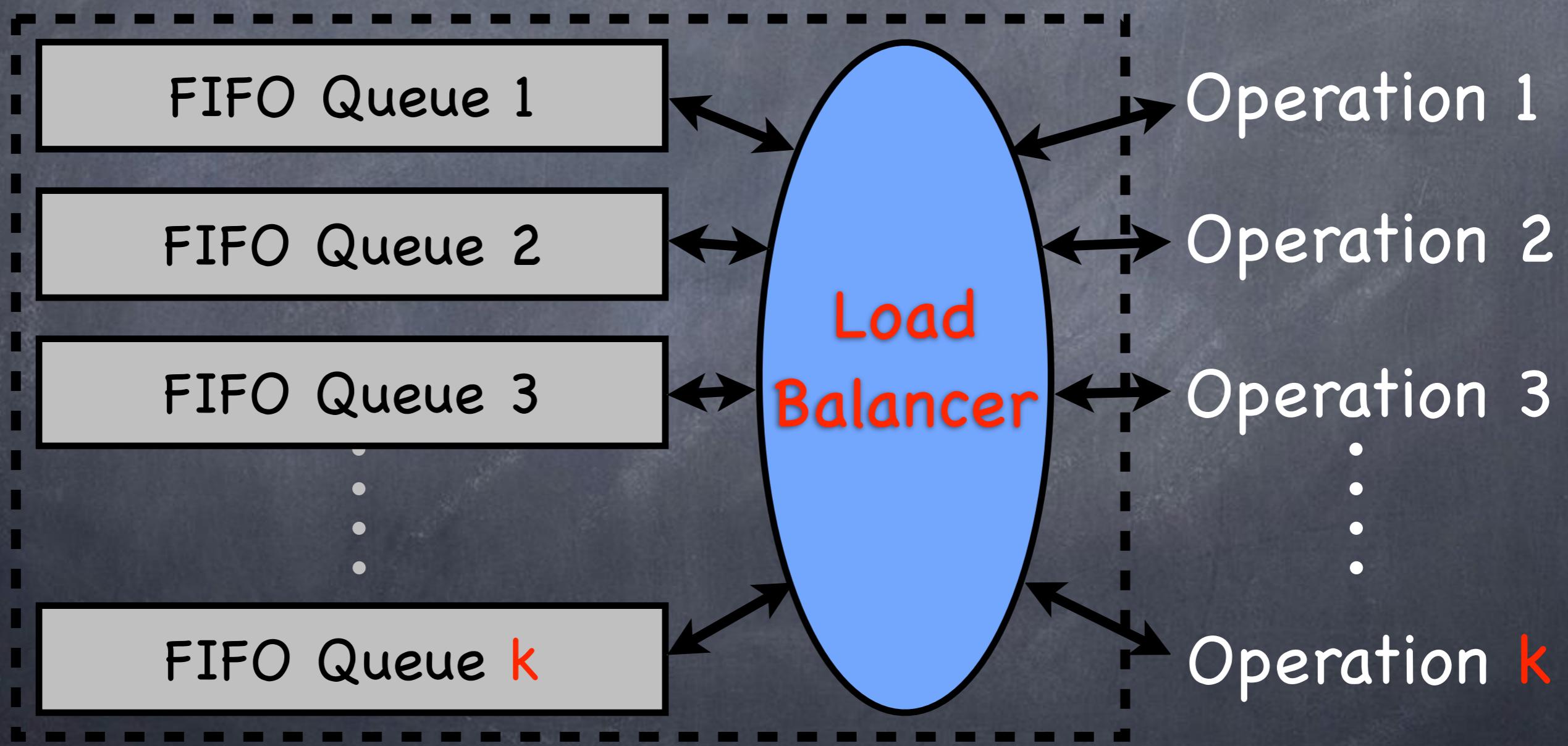


Distributed Queues

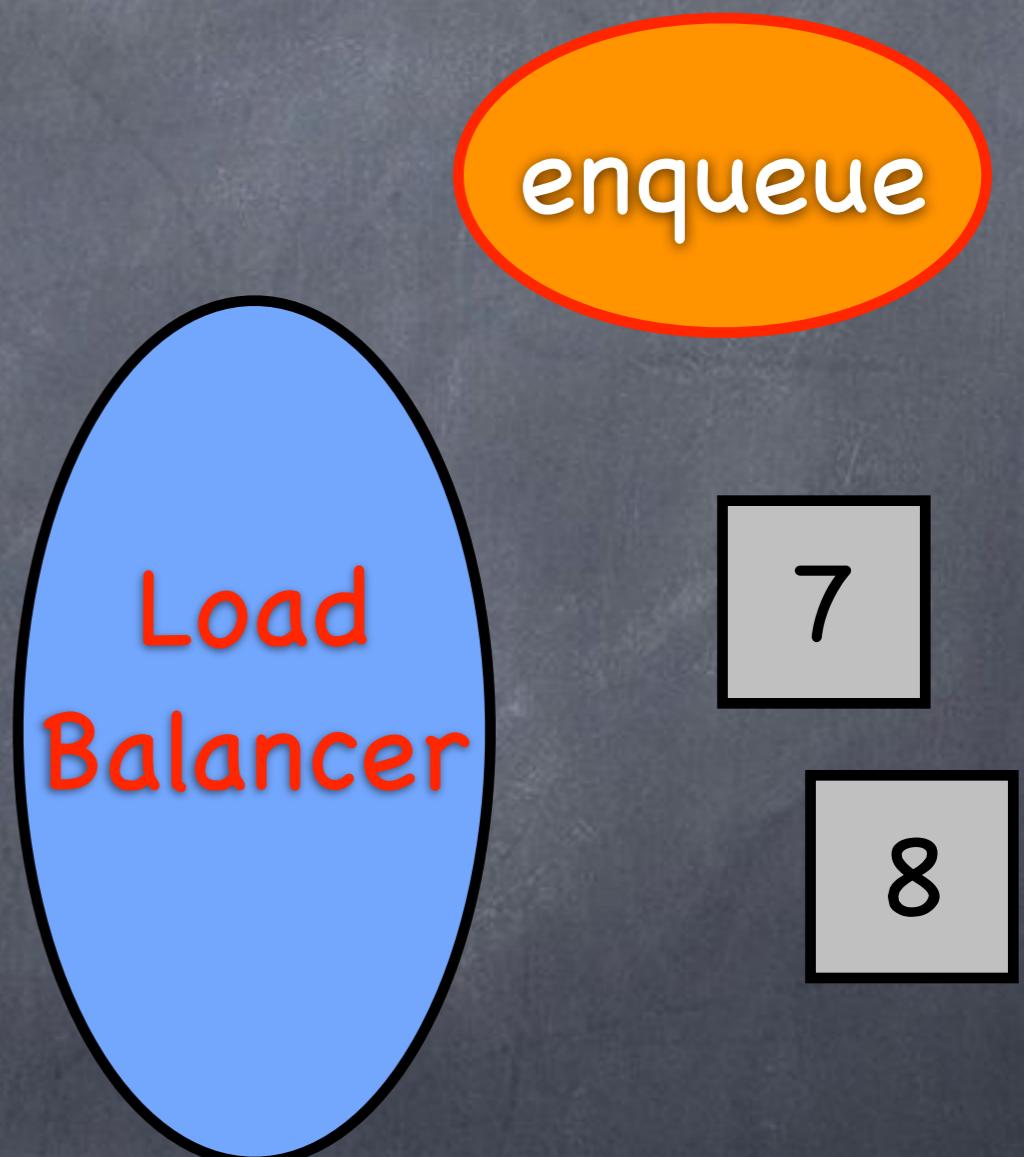
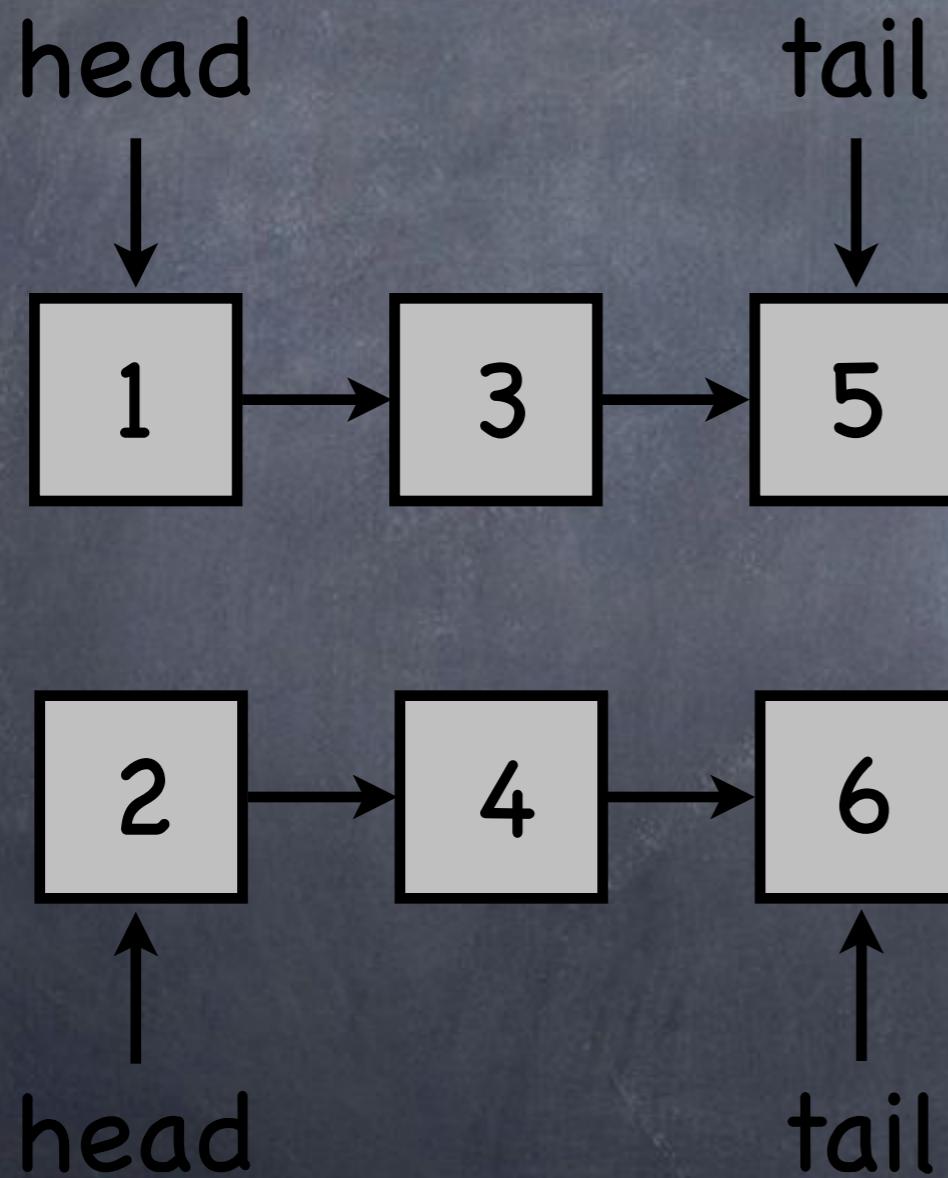
[_,Payer,Röck,Sokolova'12]



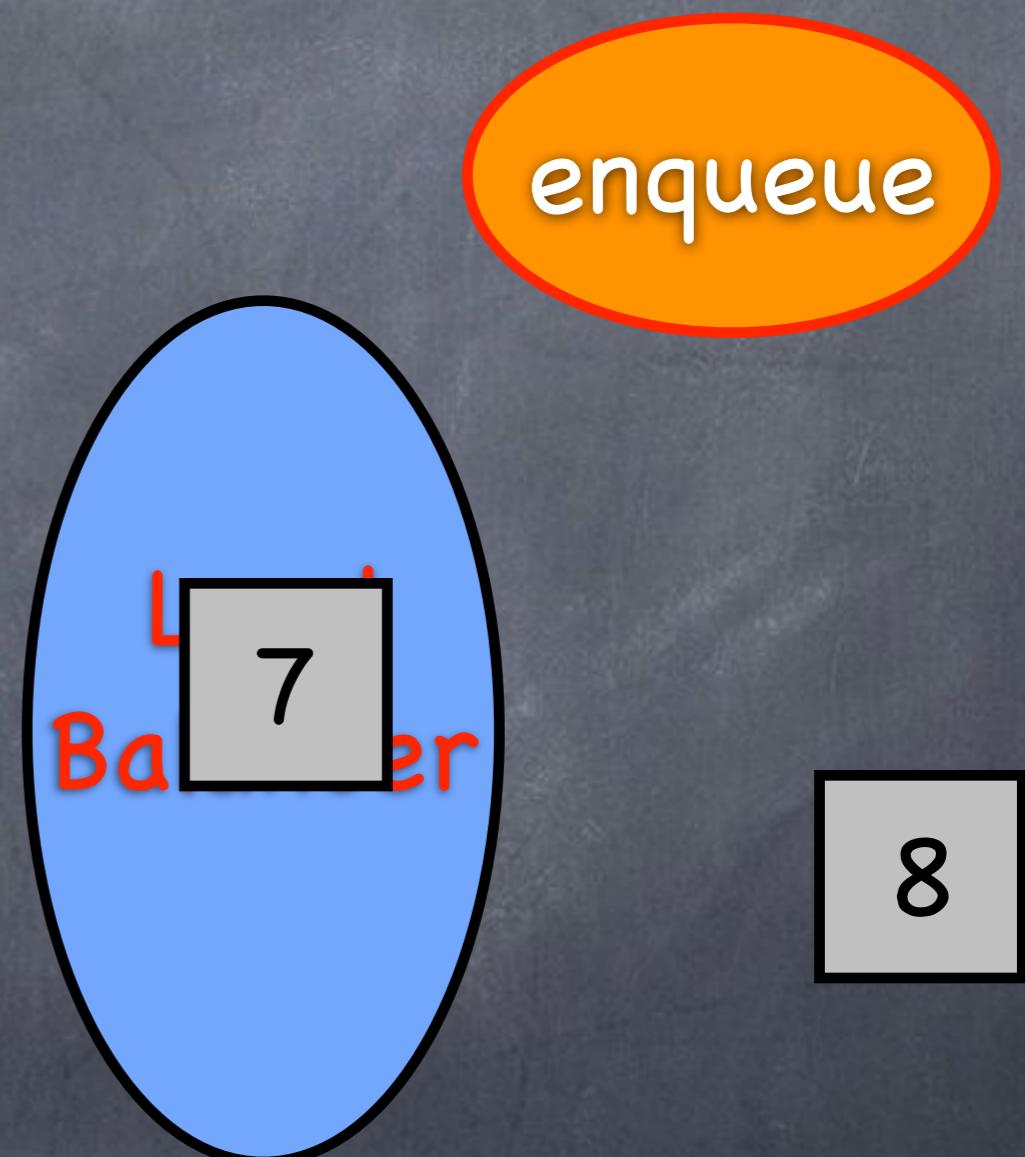
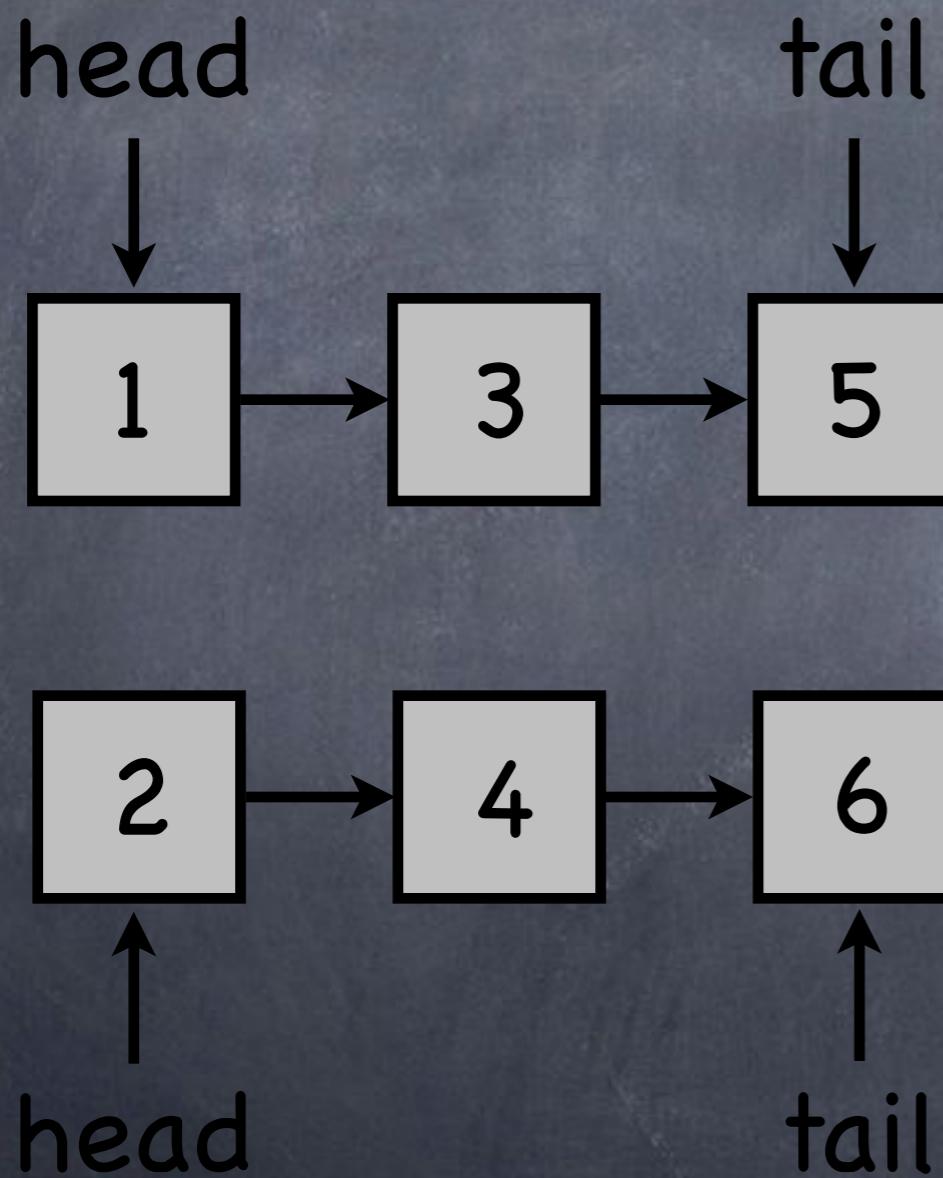
Up to k Parallel Enqueues and k Parallel Dequeues



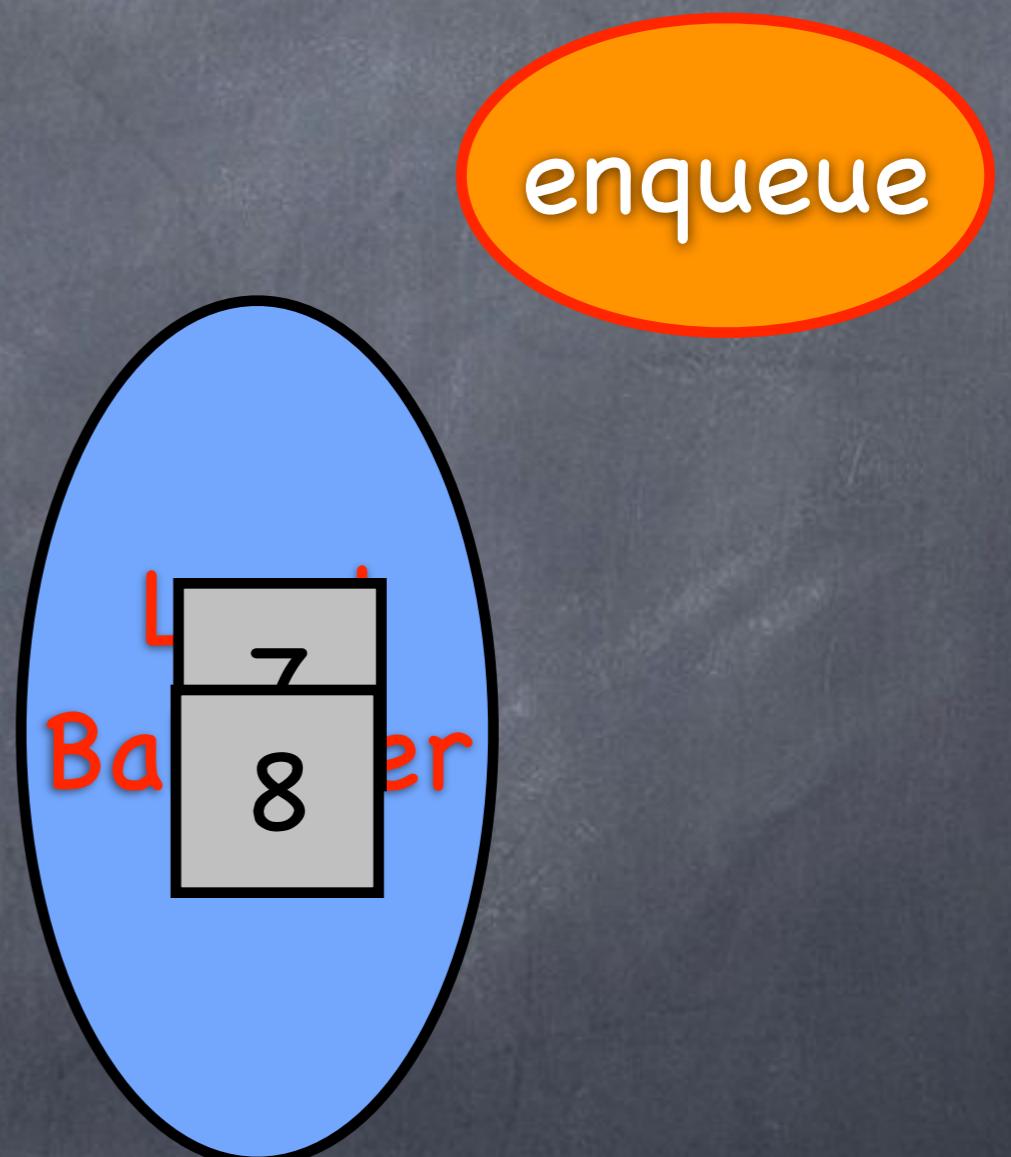
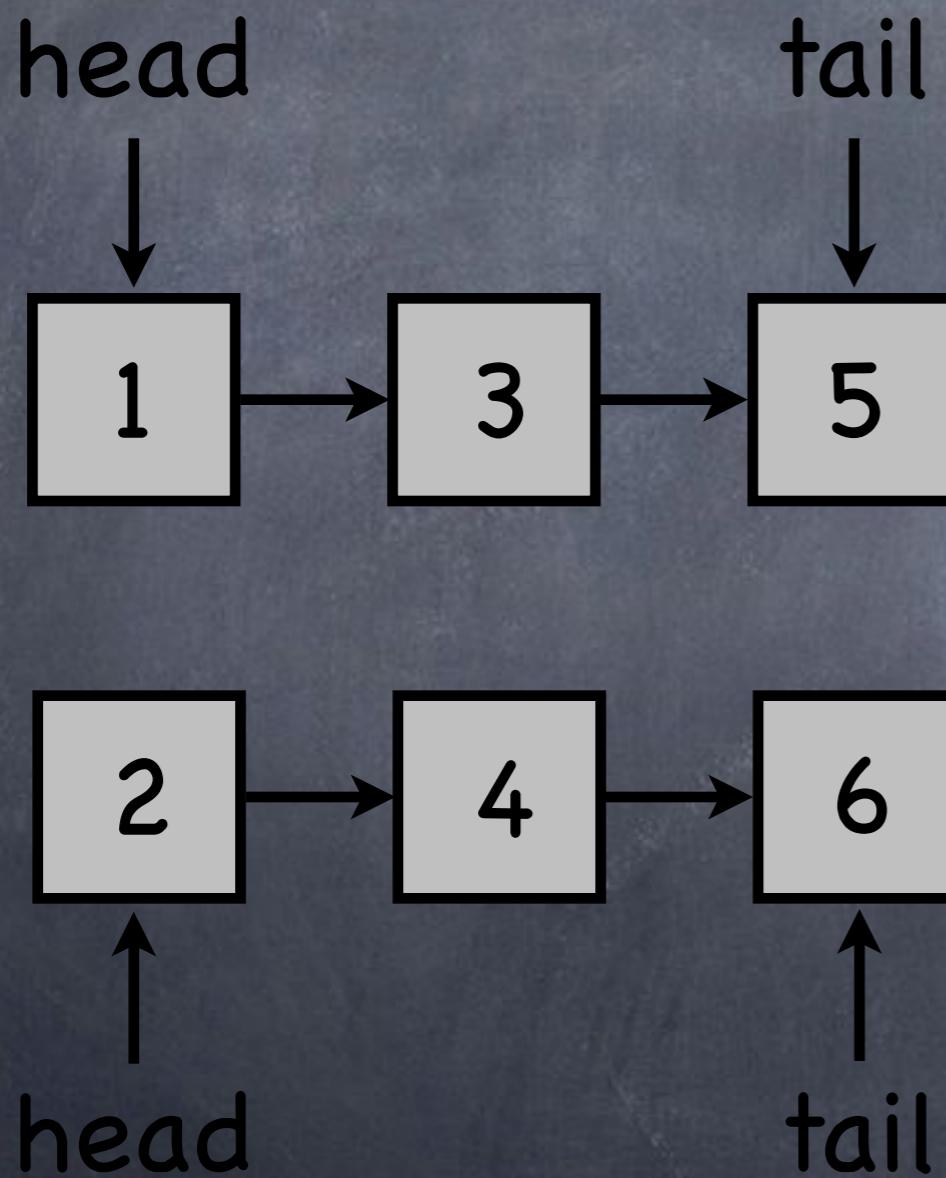
Load Balancing



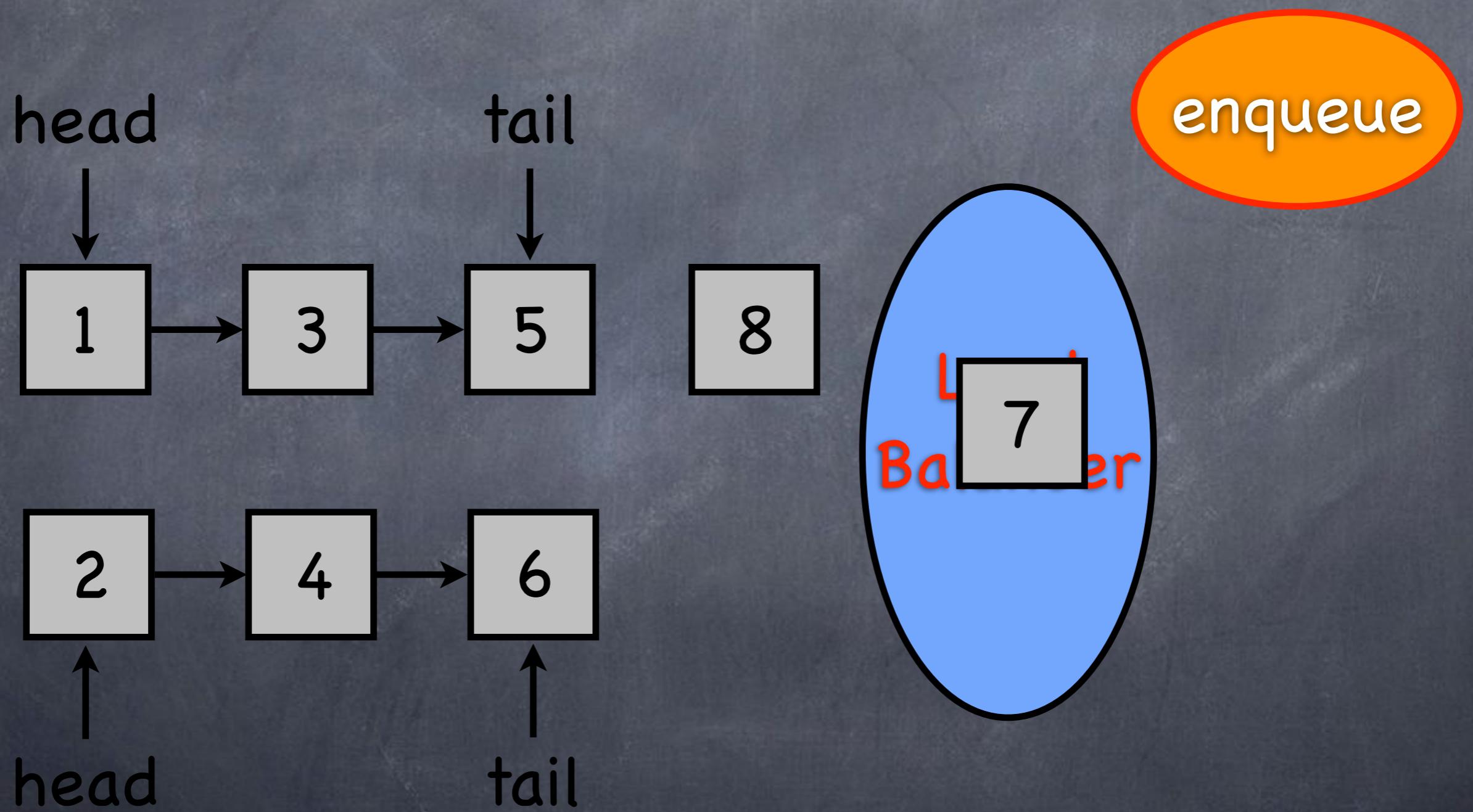
Load Balancing



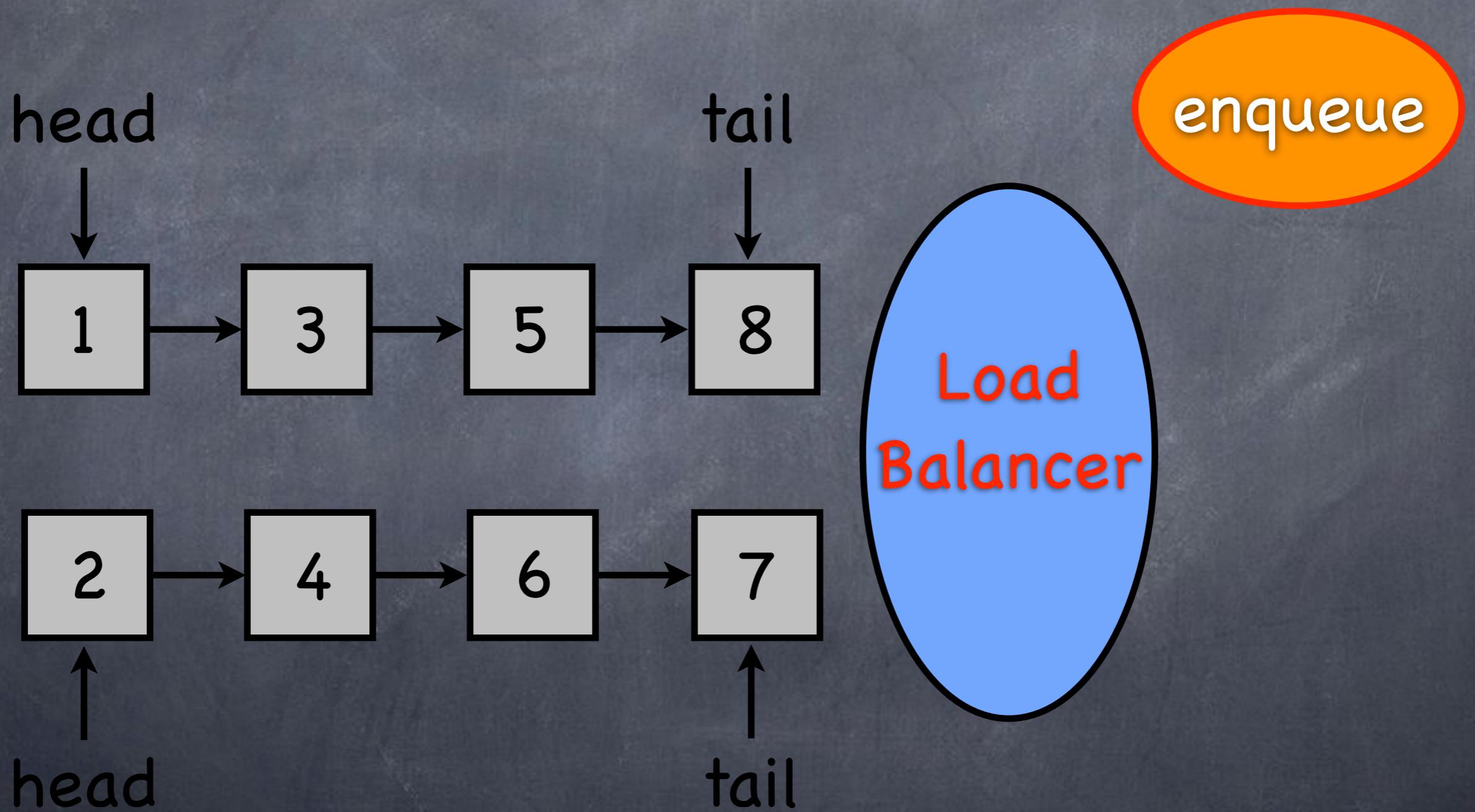
Load Balancing



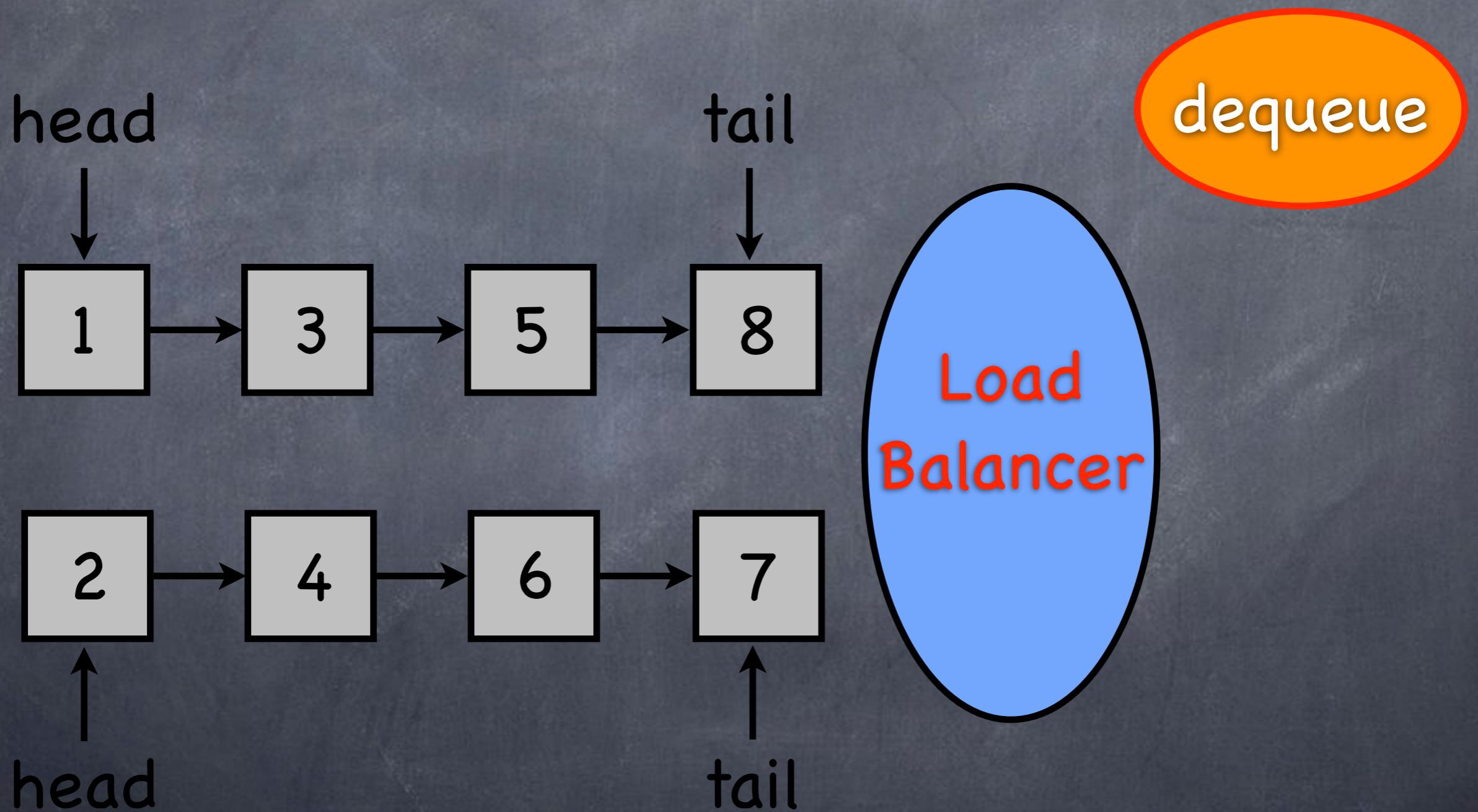
Load Balancing



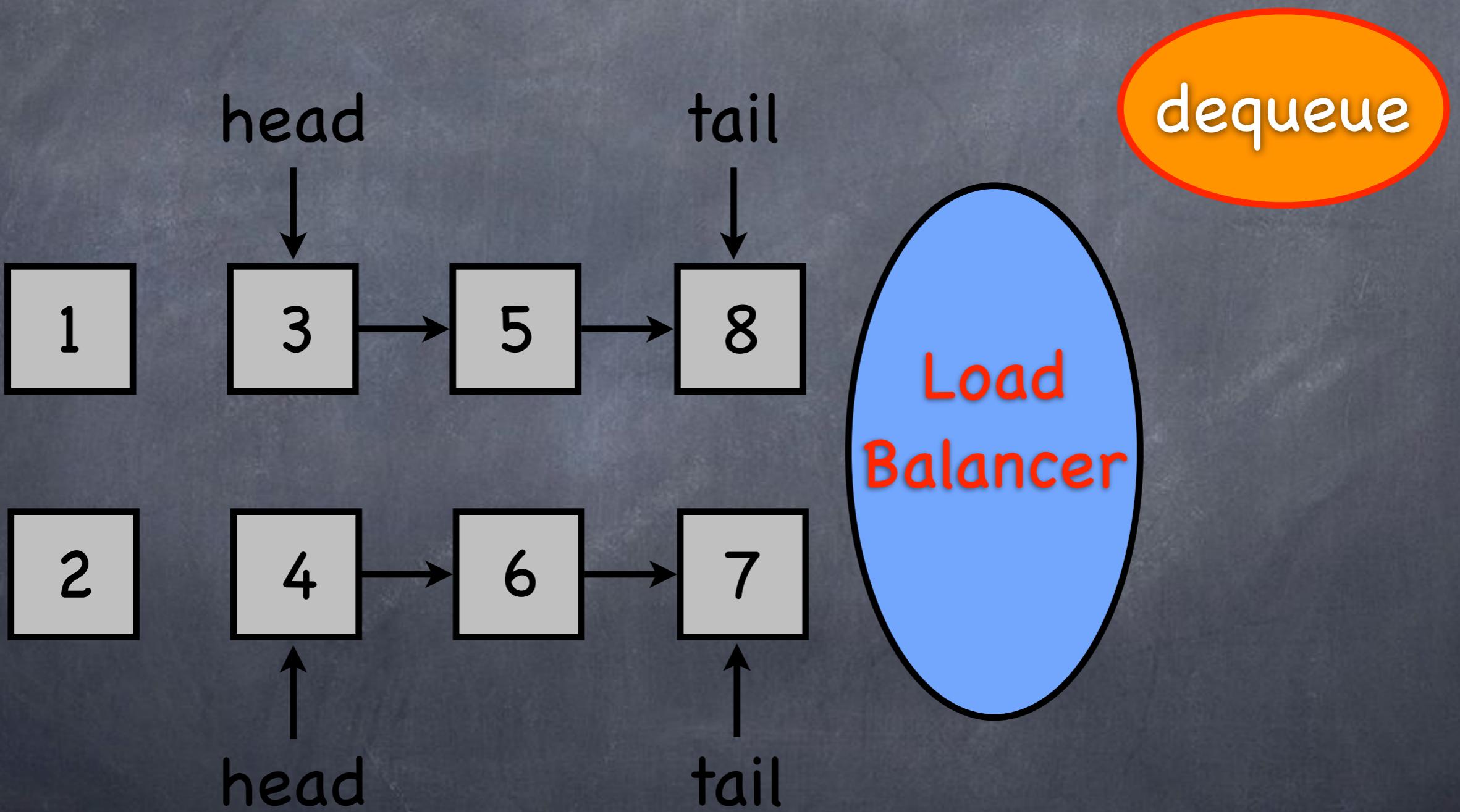
Load Balancing



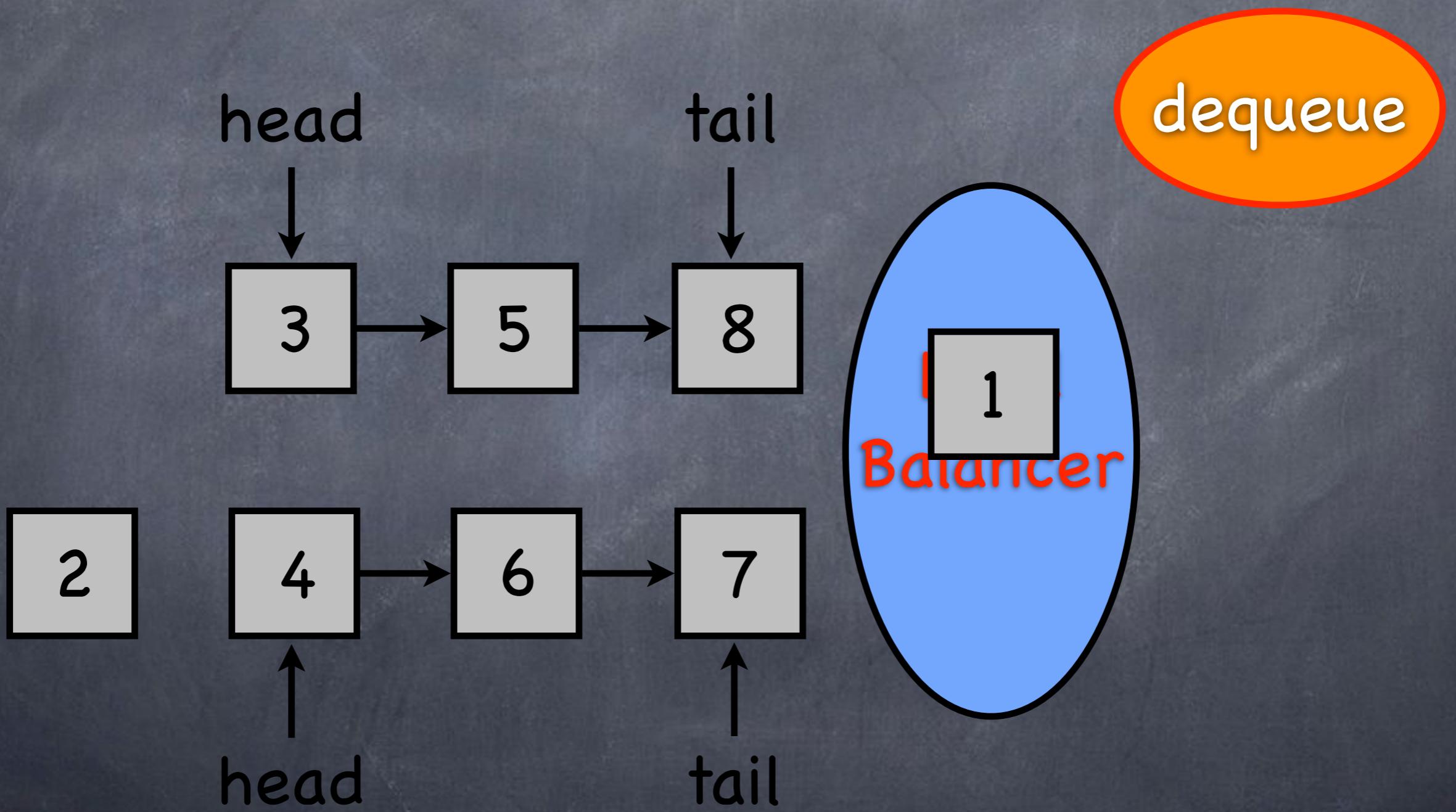
Load Balancing



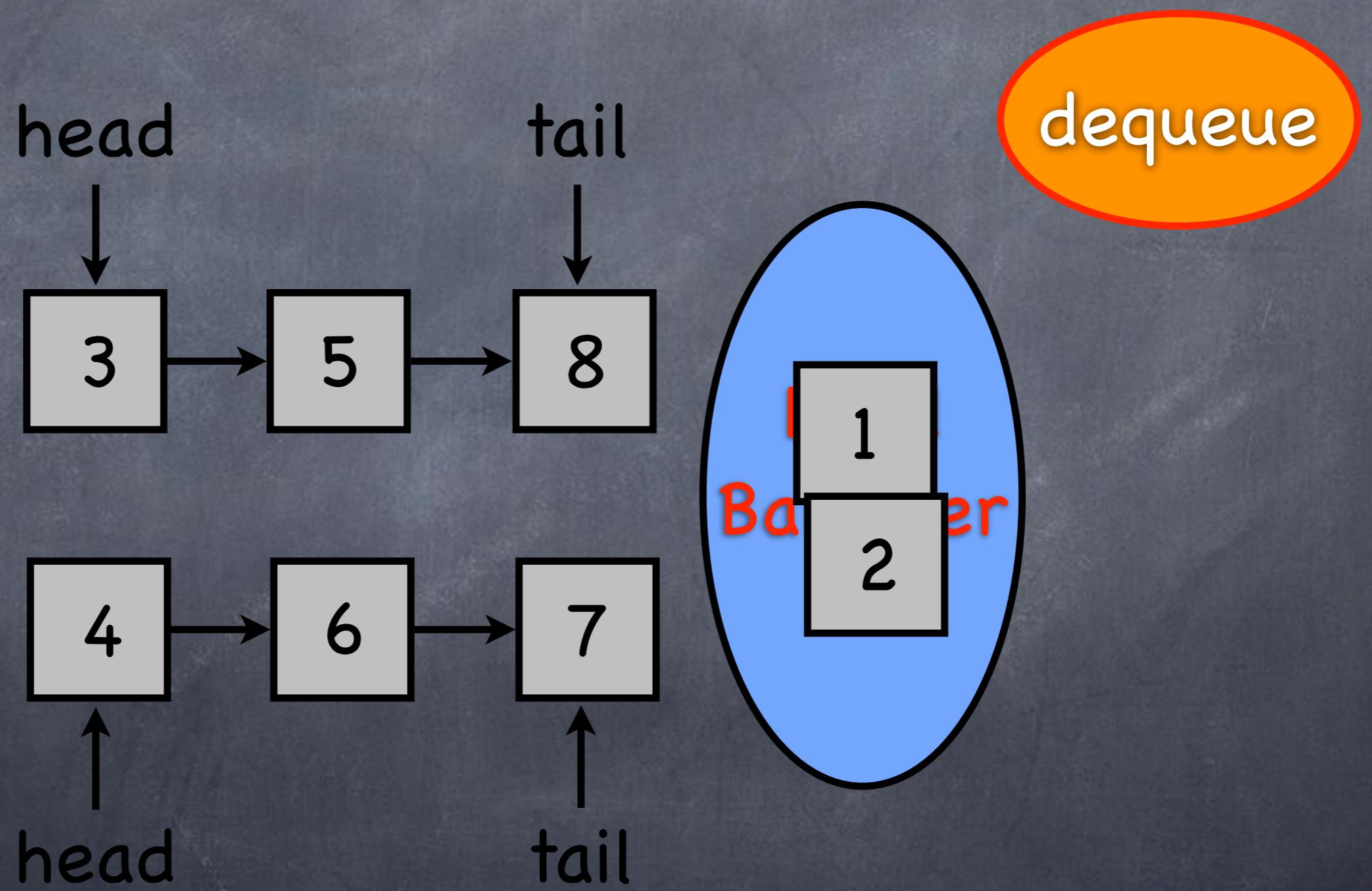
Load Balancing



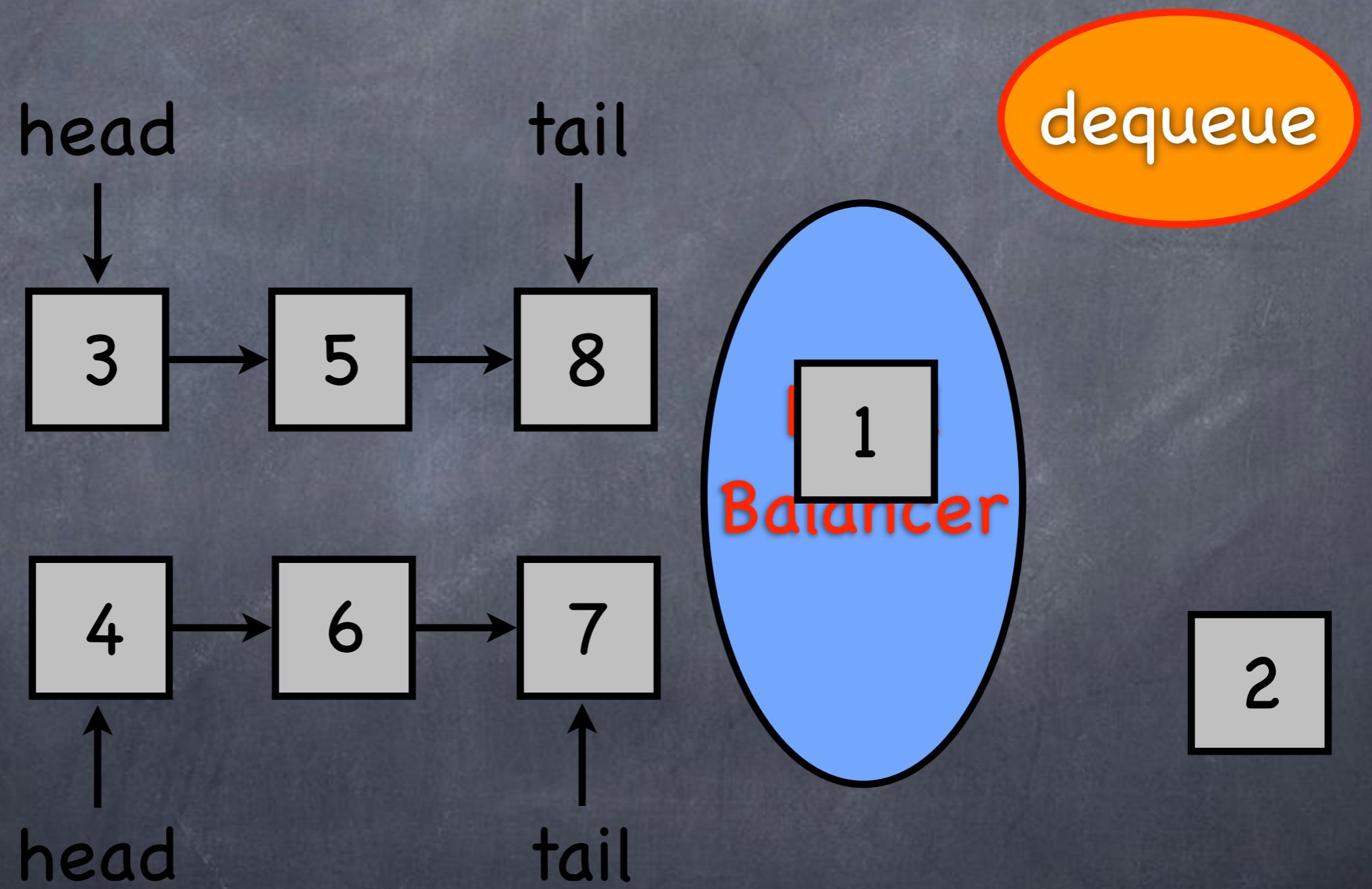
Load Balancing



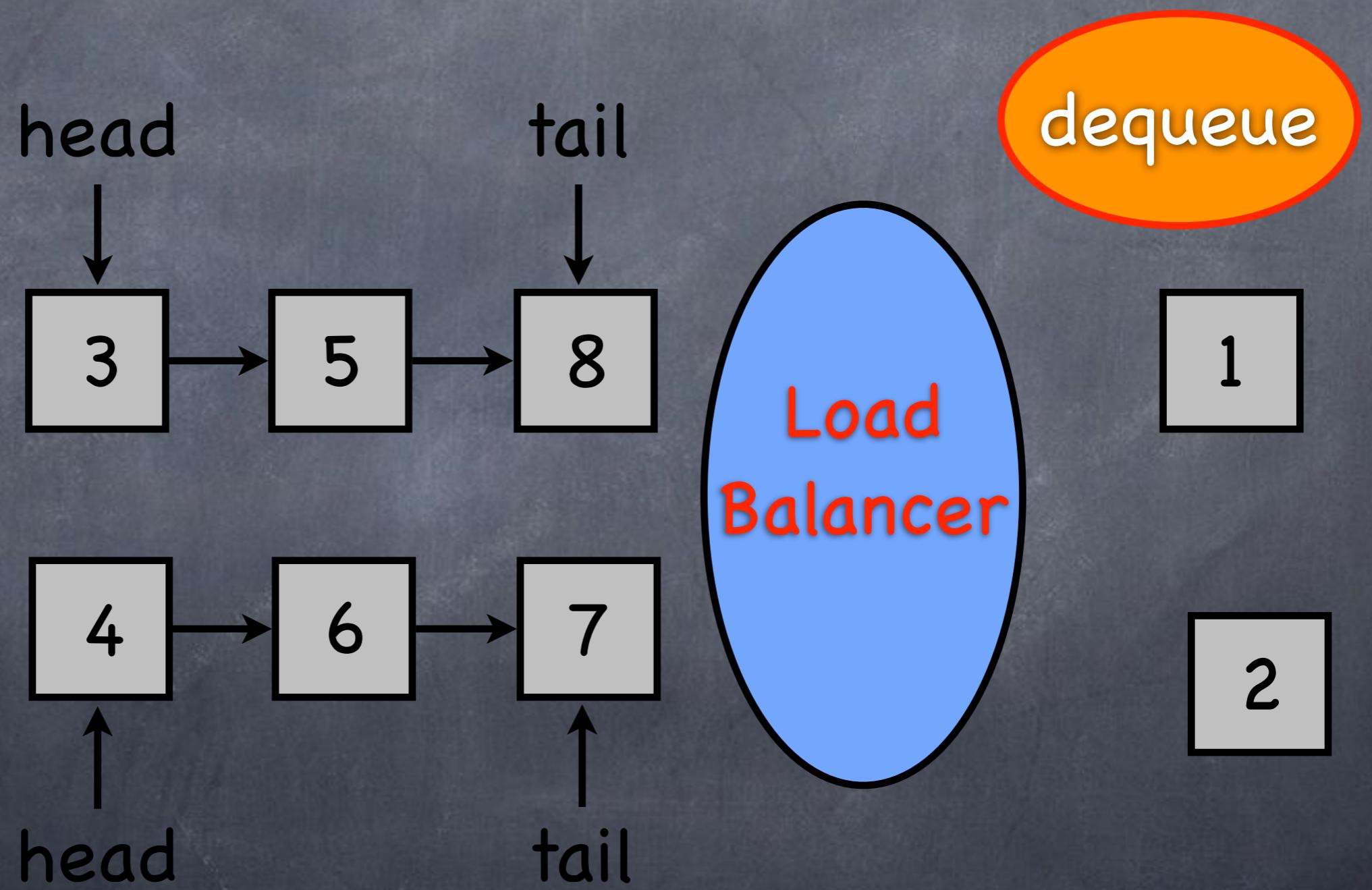
Load Balancing



Load Balancing

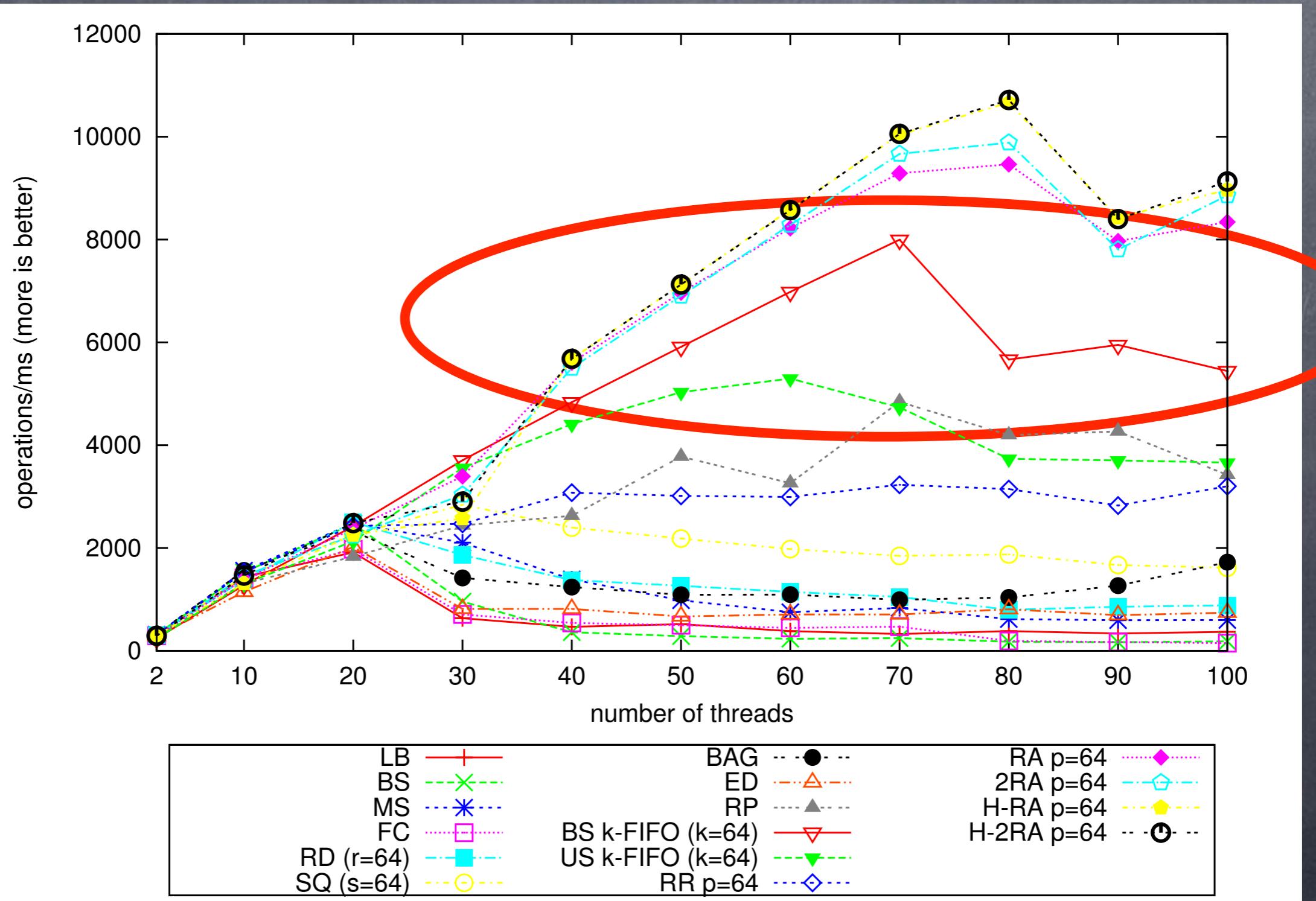


Load Balancing



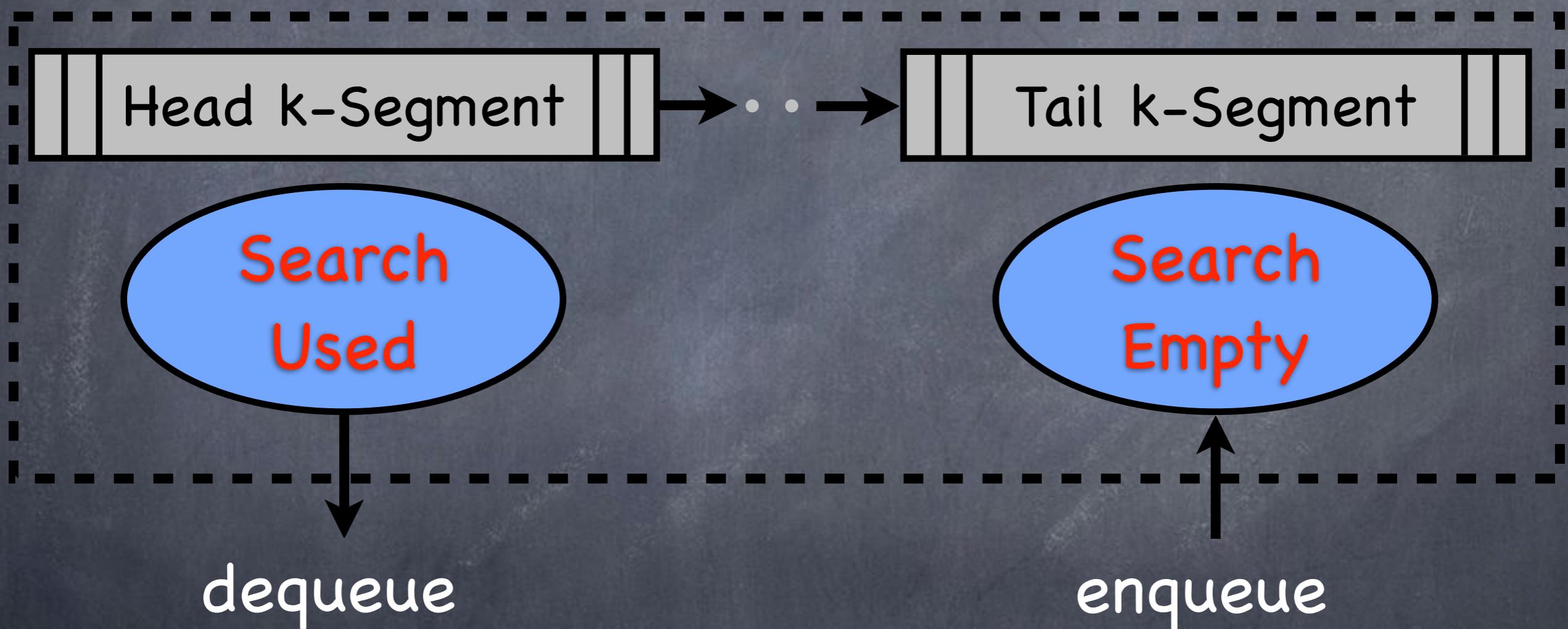
Emptiness
Check?

Segmented Queues



Segmented Queues

[Afek,Korland,Yanovsky'10],[_,Lippautz,Payer'12]



Emptiness
Check?

```

1 bool enqueue(item):
2     while true:
3         tail_old = get_tail();
4         head_old = get_head();
5         item_old, index = find_empty_slot(tail_old, k, TESTS);
6         if tail_old == get_tail():
7             if item_old.value == EMPTY:
8                 item_new = atomic_value(item, item_old.counter + 1);
9                 if CAS(&tail_old[index], item_old, item_new):
10                     if committed(tail_old, item_new, index):
11                         return true;
12             else:
13                 if queue_full(head_old, tail_old):
14                     if segment_not_empty(head_old, k) && head == get_head():
15                         return false;
16                     advance_head(head_old, k);
17                     advance_tail(tail_old, k);
18
19 bool committed(tail_old, item_new, index):
20     if tail_old[index] != item_new:
21         return true;
22     head_current = get_head();
23     tail_current = get_tail();
24     item_empty = atomic_value(EMPTY, item_new.counter + 1);
25     if in_queue_after_head(tail_old, tail_current, head_current):
26         return true;
27     else if not_in_queue(tail_old, tail_current, head_current):
28         if !CAS(&tail_old[index], item_new, item_empty):
29             return true;
30     else: //in queue at head
31         head_new = atomic_value(head_current.value, head_current.counter + 1);
32         if CAS(&head, head_current, head_new):
33             return true;
34         if !CAS(&tail_old[index], item_new, item_empty):
35             return true;
36     return false;

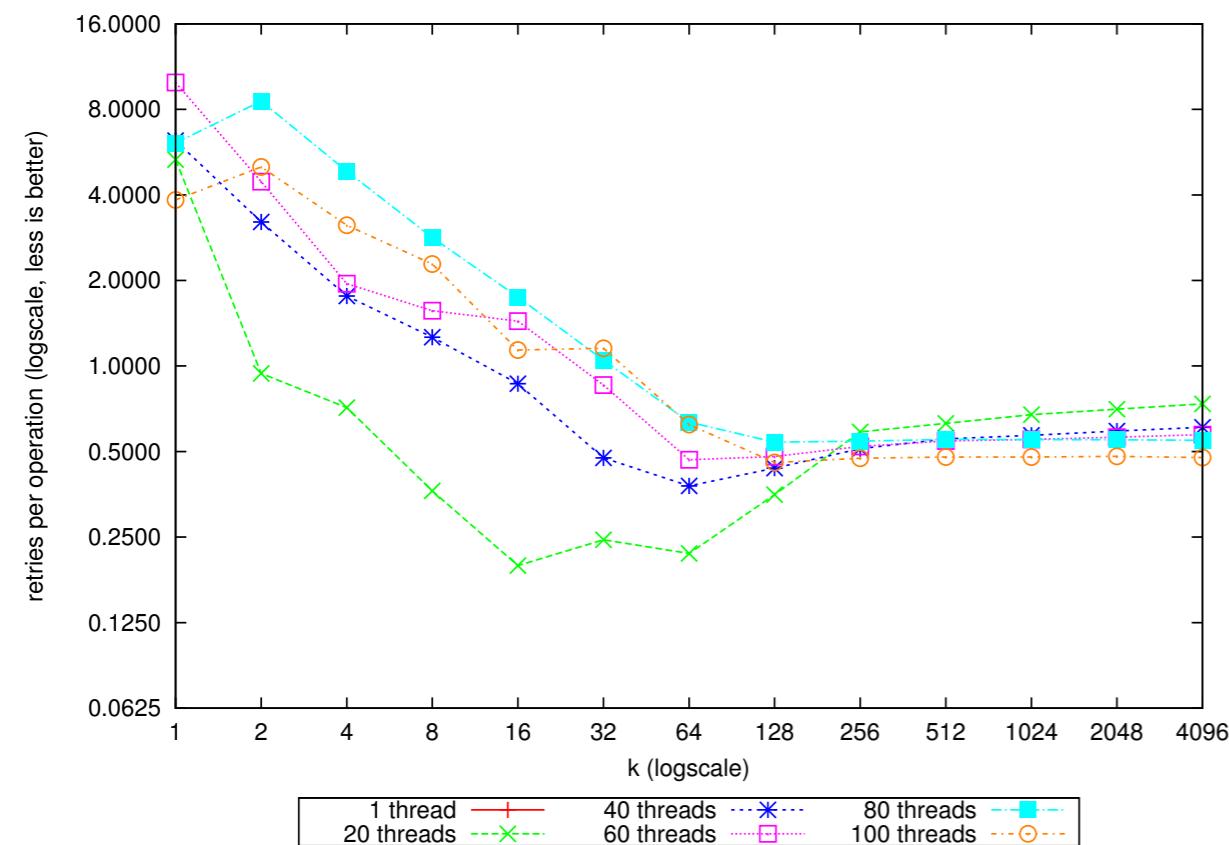
```

enqueue

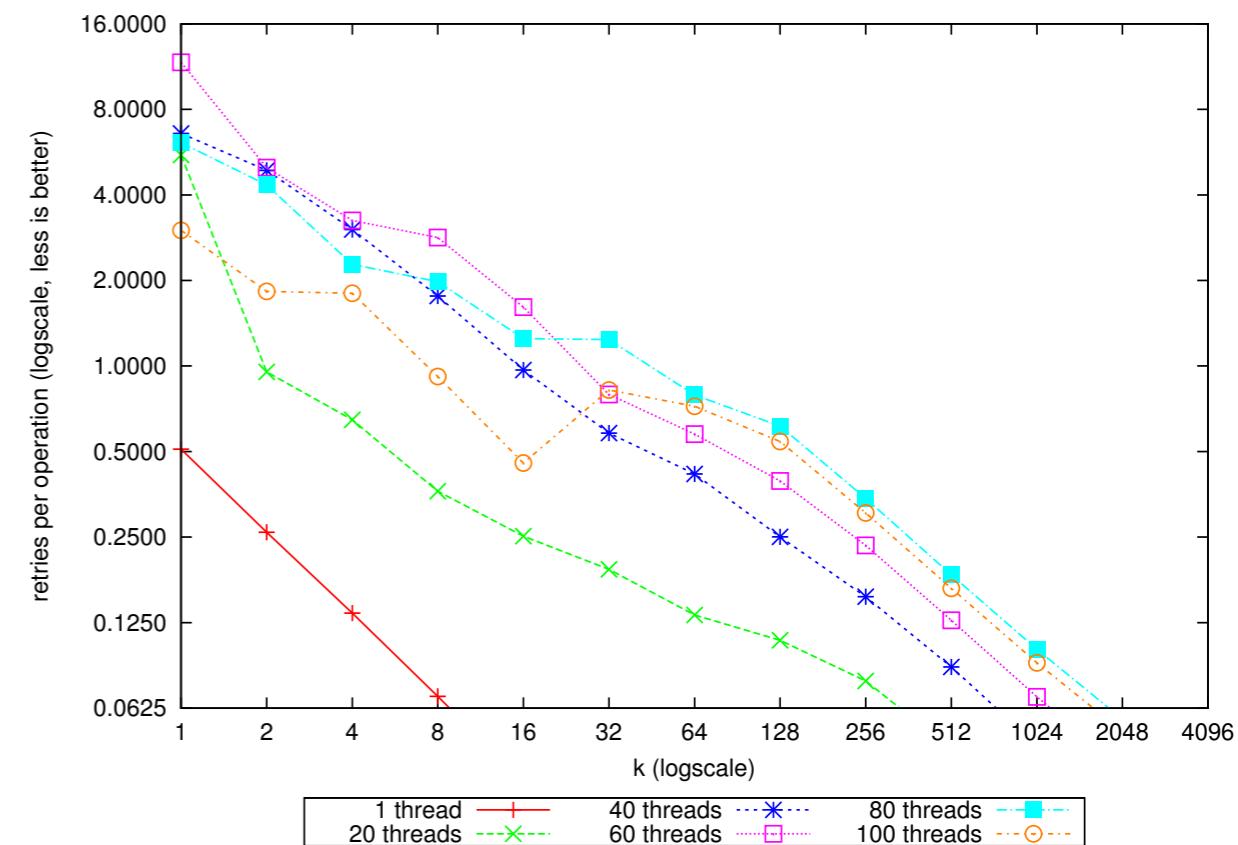
dequeue

```
38 item dequeue():
39     while true:
40         tail_old = get_tail();
41         head_old = get_head();
42         item_old, index = find_item(head_old, k);
43         if head_old == head:
44             if item_old.value != EMPTY:
45                 if head_old.value == tail_old.value:
46                     advance_tail(tail_old, k);
47                     item_empty = atomic_value(EMPTY, item_old.counter + 1);
48                     if CAS(&head_old[index], item_old, item_empty):
49                         return item_old.value;
50             else:
51                 if head_old.value == tail_old.value && tail_old == get_tail():
52                     return null;
53                     advance_head(head_old, k);
```

CAS Retries/Operation in k

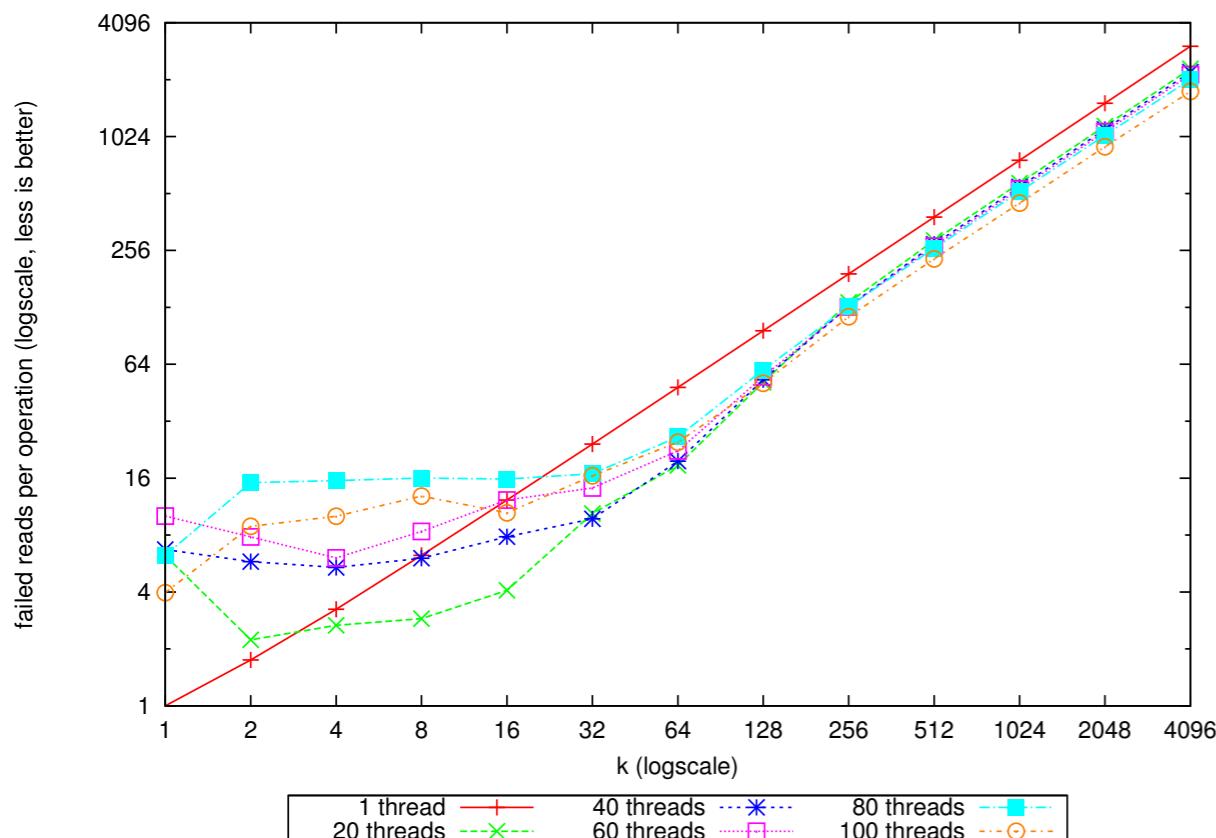


(a) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000, i = 0$)

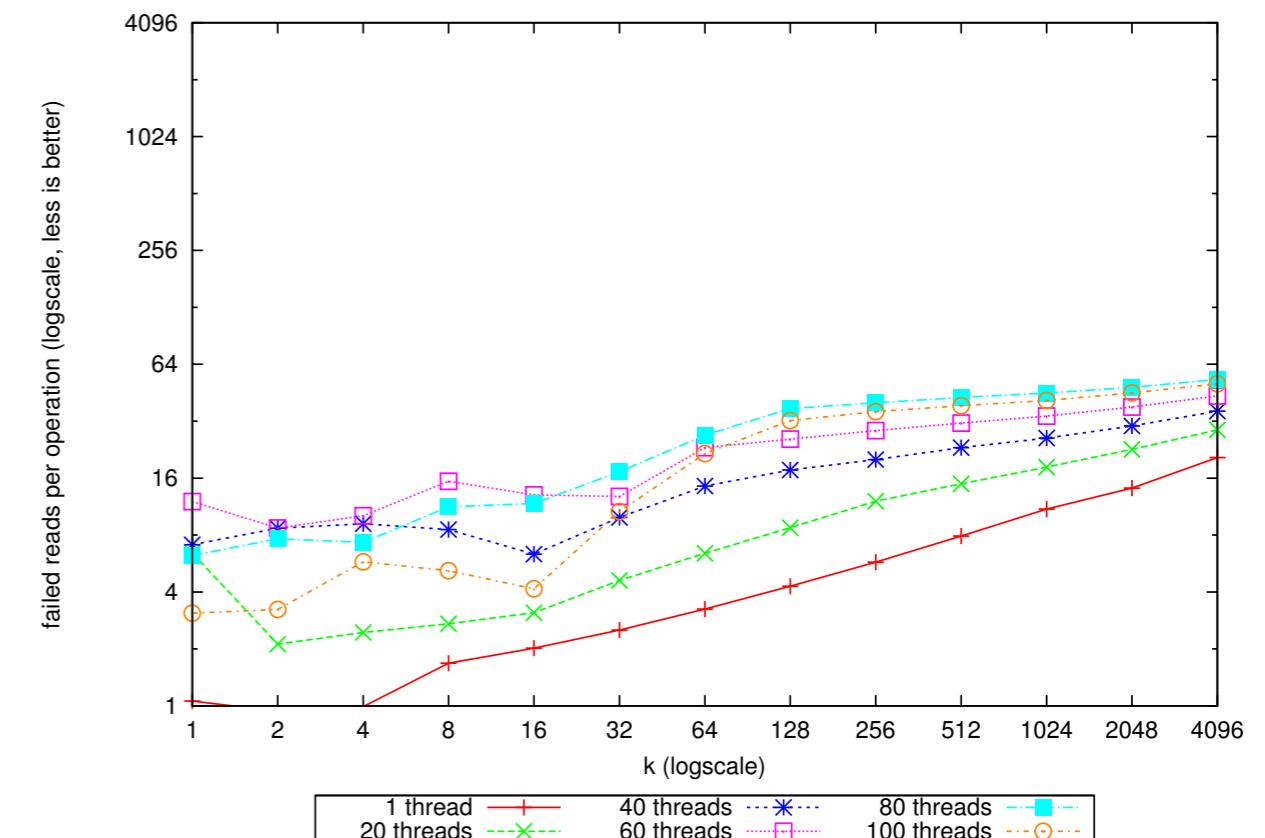


(b) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000, i = 5000$)

Failed Searches/Operation in κ

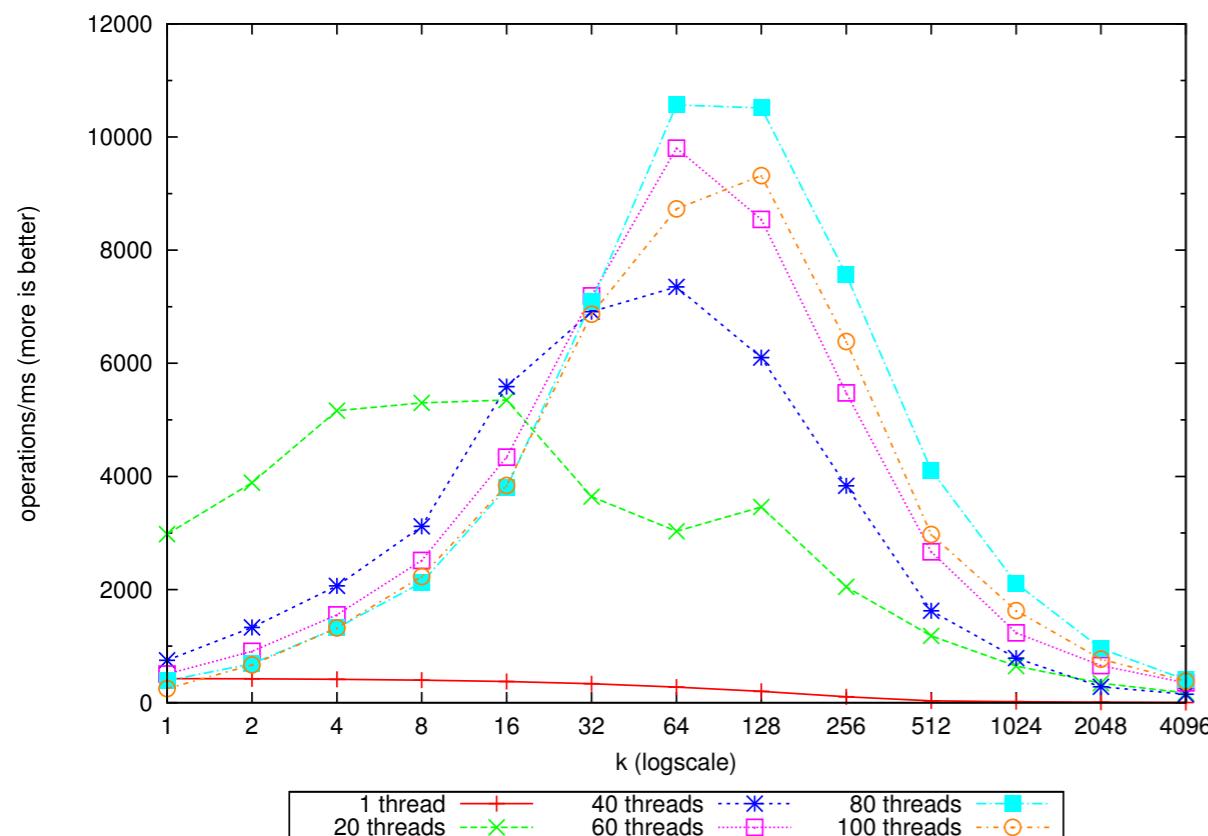


(c) BS number of failed reads per operation of very high contention producer-consumer benchmark ($c = 1000, i = 0$)

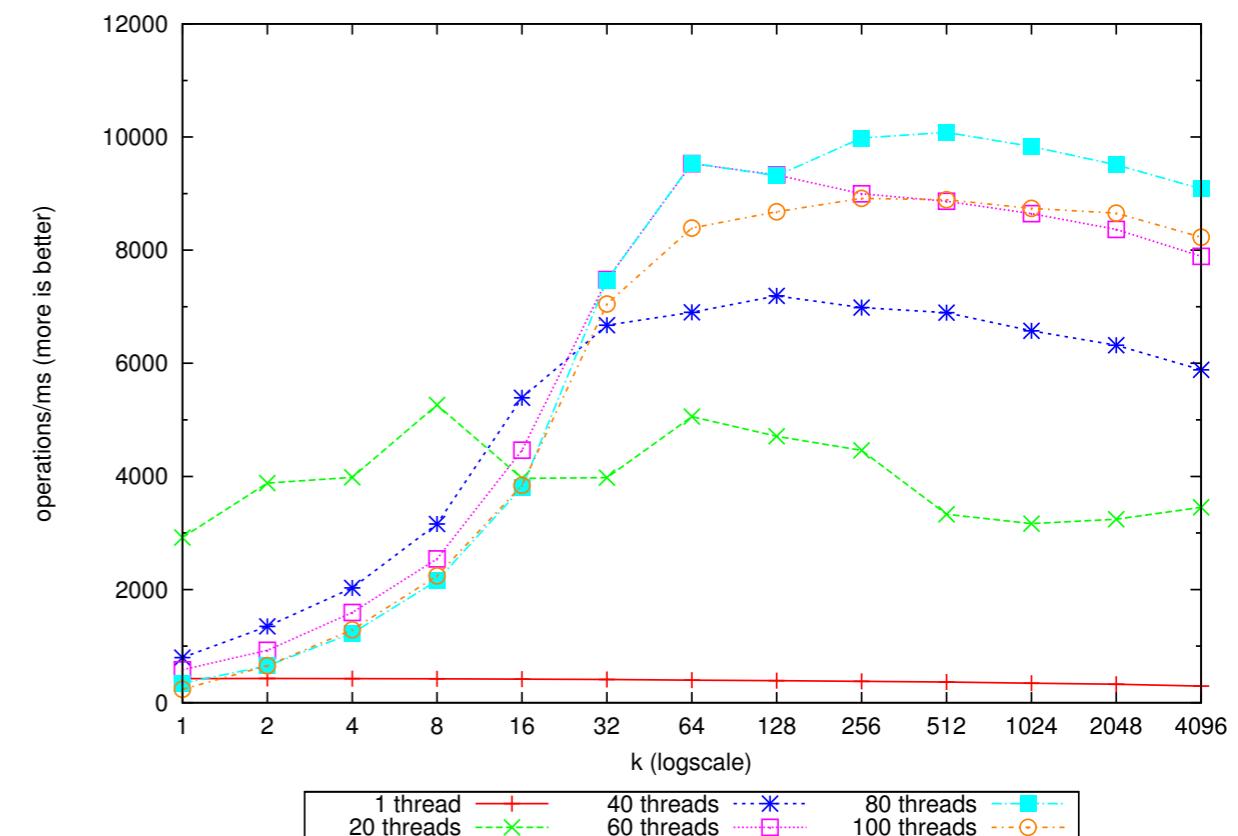


(d) BS number of failed reads per operation of very high contention producer-consumer benchmark ($c = 1000, i = 5000$)

Performance in k

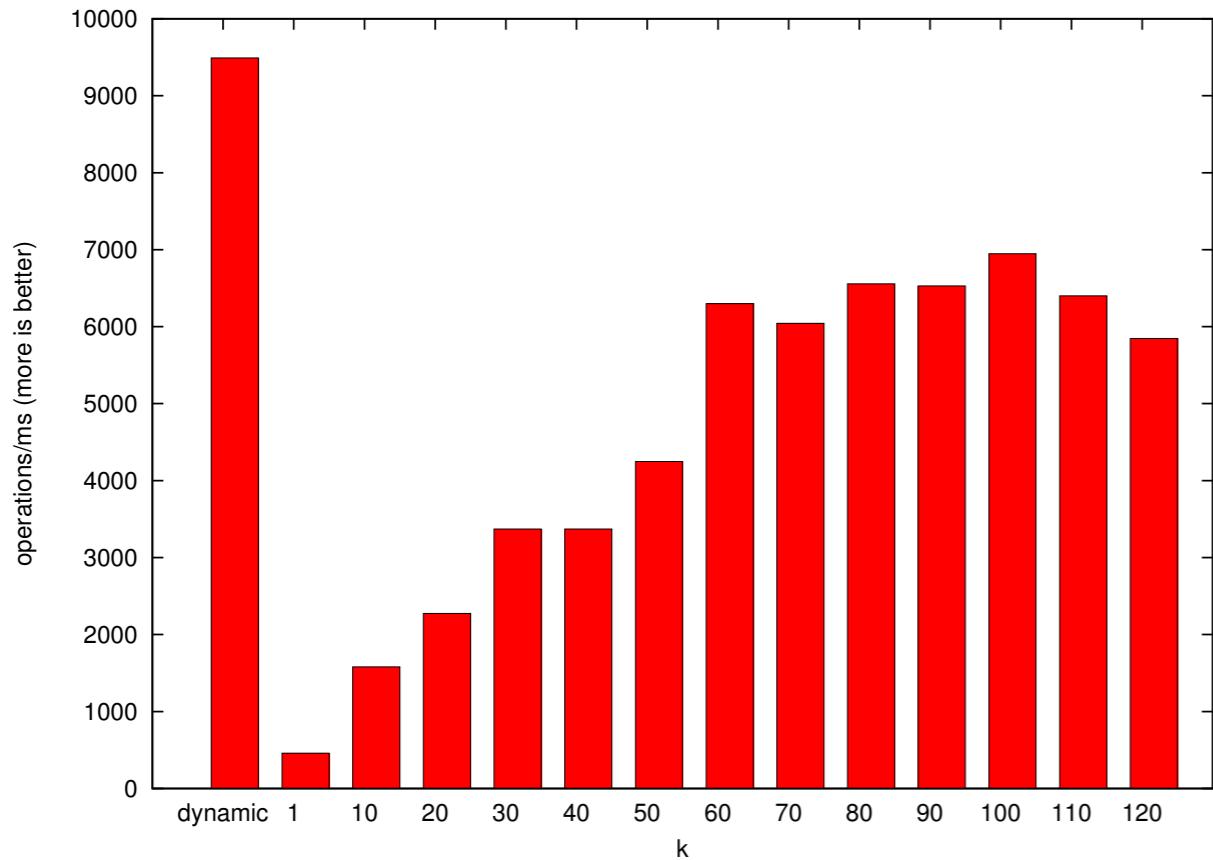


(e) BS performance of very high contention producer-consumer benchmark
($c = 1000, i = 0$)

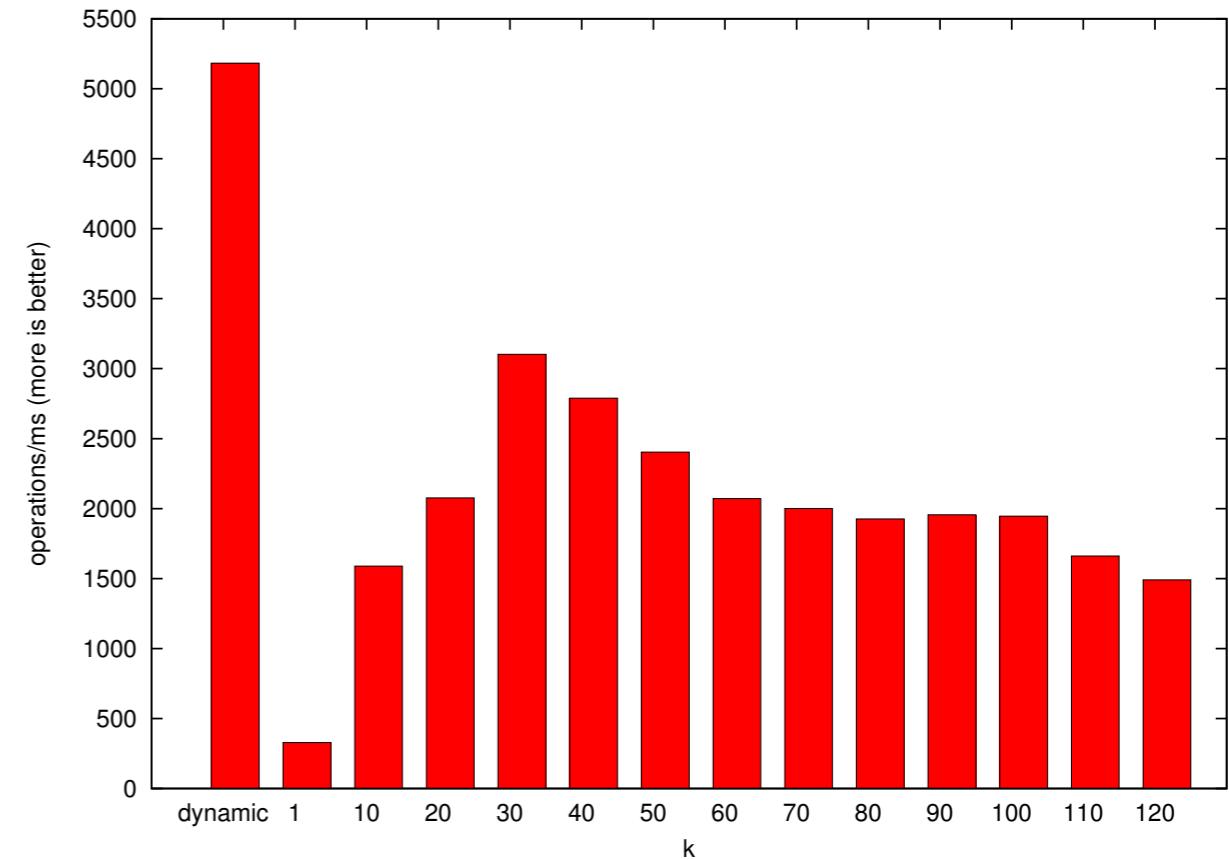


(f) BS performance of very high contention producer-consumer benchmark
($c = 1000, i = 5000$)

Dynamic k



(a) BS k -FIFO queue



(b) US k -FIFO queue

Fig. 4. Variable-load producer-consumer microbenchmarks with an increasing number of static k versus a dynamically controlled k on a 40-core (2 hyperthreads per core) server

Concurrent k -FIFO Queue

- with a k -FIFO queue elements may be returned **out-of-FIFO order up to k**

Concurrent k -FIFO Queue

- with a k -FIFO queue elements may be returned **out-of-FIFO order up to k**
- the **oldest** element is returned after at most k **dequeue** operations that may return **elements not younger than $k-1$** (or return nothing)

Concurrent k -FIFO Queue

- with a k -FIFO queue elements may be returned **out-of-FIFO order** up to k
- the **oldest** element is returned after at most k **dequeue** operations that may return **elements** not younger than $k-1$ (or return nothing)
- **starvation-free** for finite k

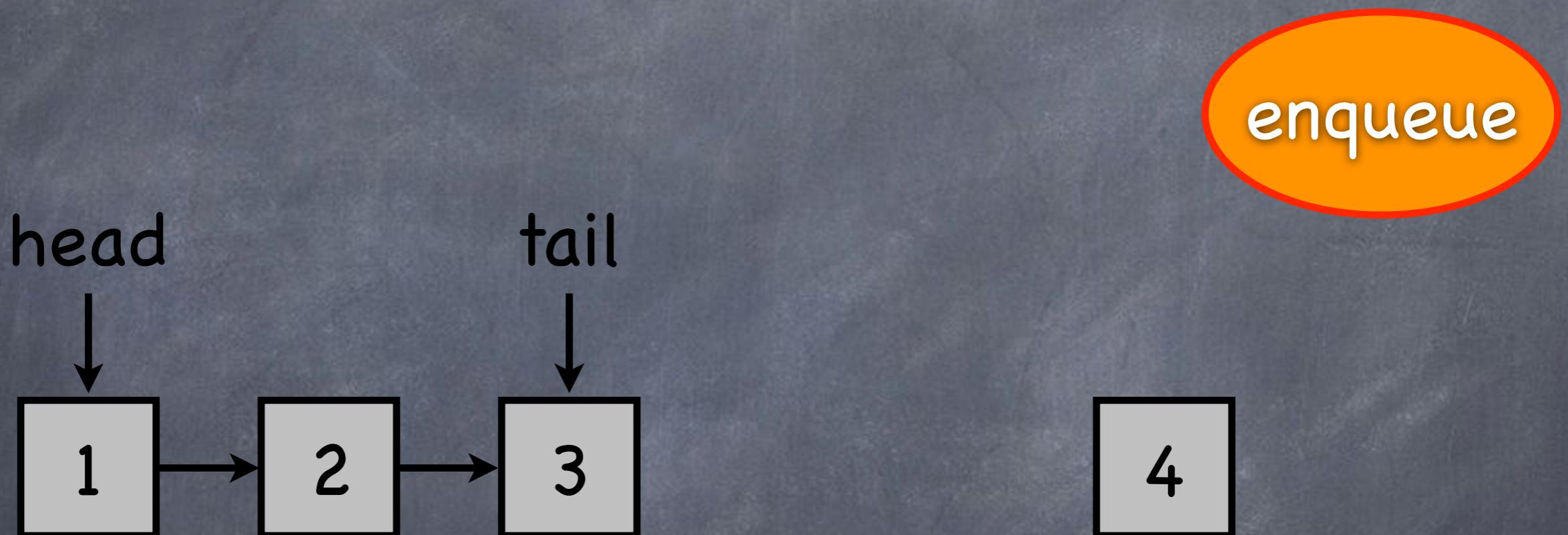
Concurrent k -FIFO Queue

- with a k -FIFO queue elements may be returned **out-of-FIFO order** up to k
- the **oldest** element is returned after at most k **dequeue** operations that may return elements not younger than $k-1$ (or return nothing)
- **starvation-free** for finite k
- **1**-FIFO queue = regular FIFO queue

Concurrent k -FIFO Queue

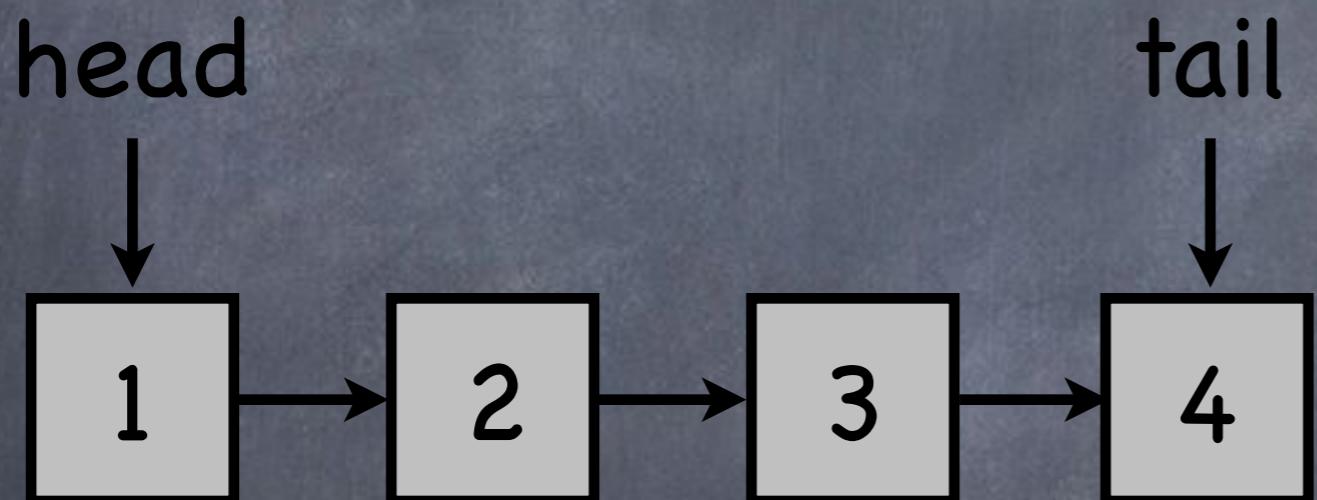
- with a k -FIFO queue elements may be returned **out-of-FIFO order** up to k
- the **oldest** element is returned after at most k **dequeue** operations that may return elements not younger than $k-1$ (or return nothing)
- **starvation-free** for finite k
- **1**-FIFO queue = regular FIFO queue
- bigger $k \rightarrow$ better performance, scalability?

Concurrent 3-FIFO Queue (k=3)

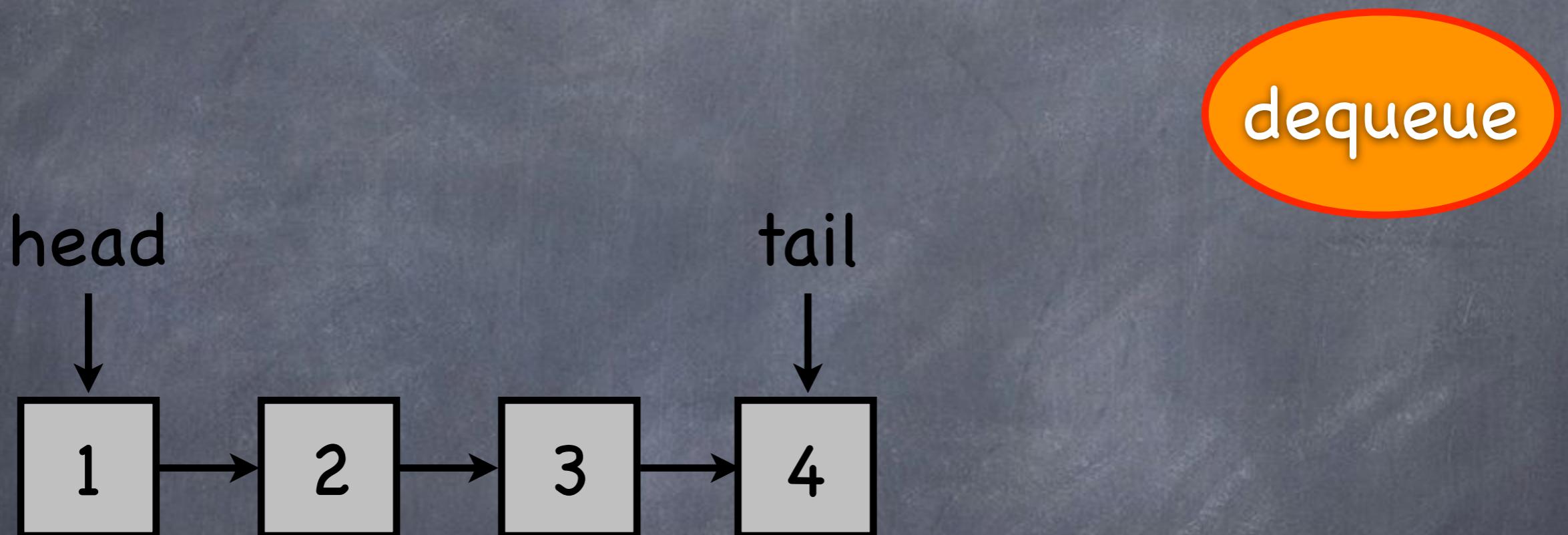


Concurrent 3-FIFO Queue (k=3)

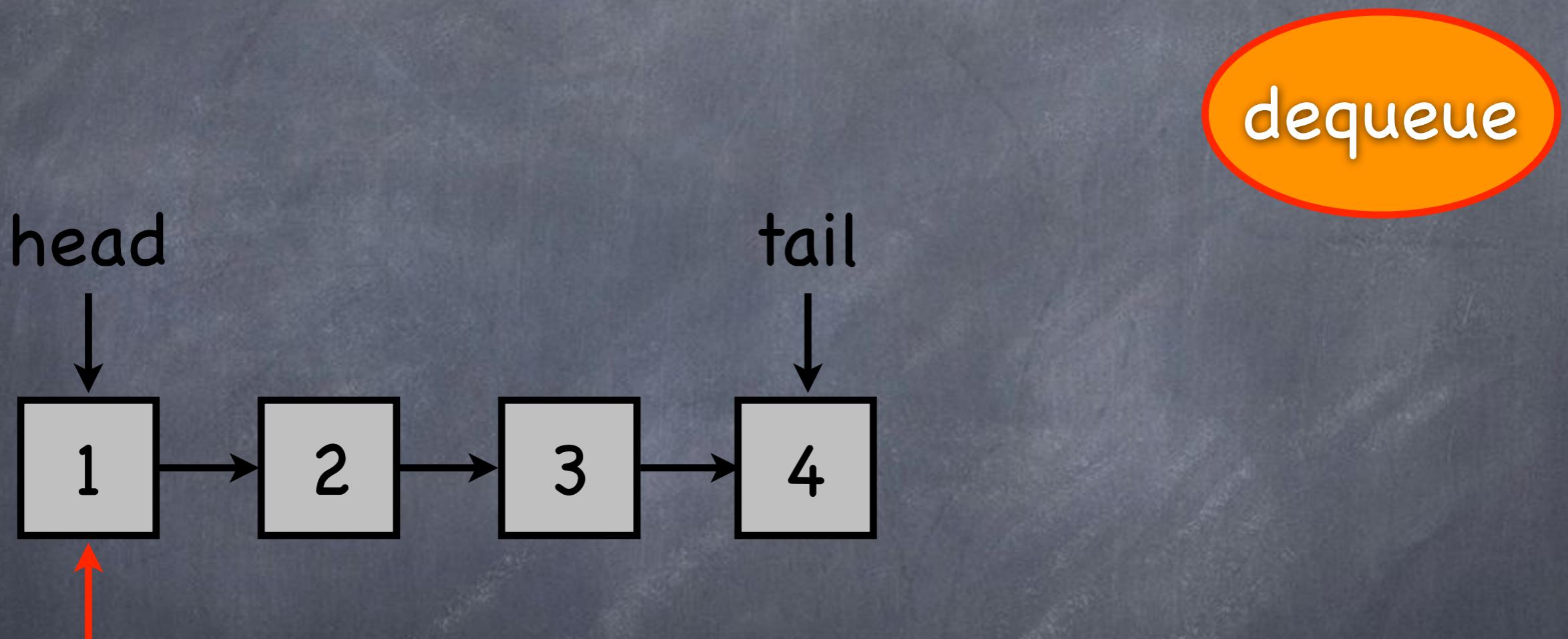
enqueue



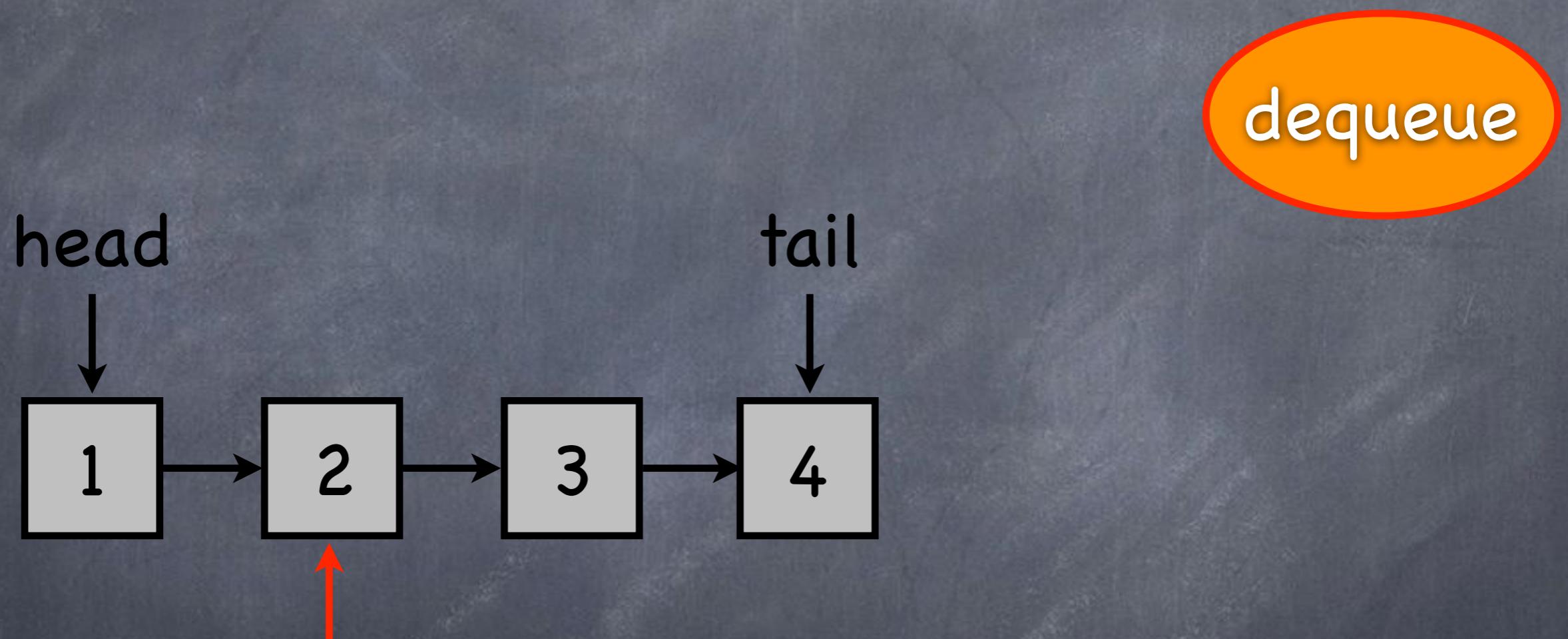
Concurrent 3-FIFO Queue (k=3)



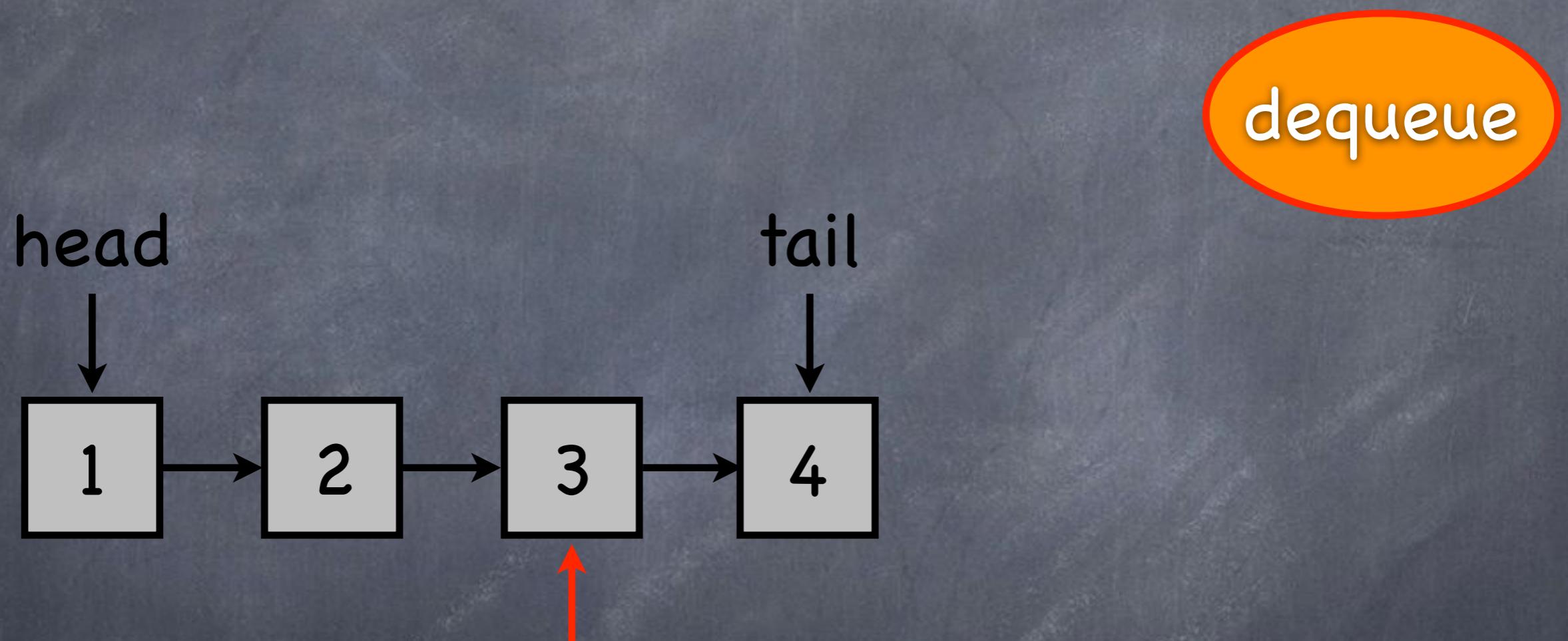
Concurrent 3-FIFO Queue (k=3)



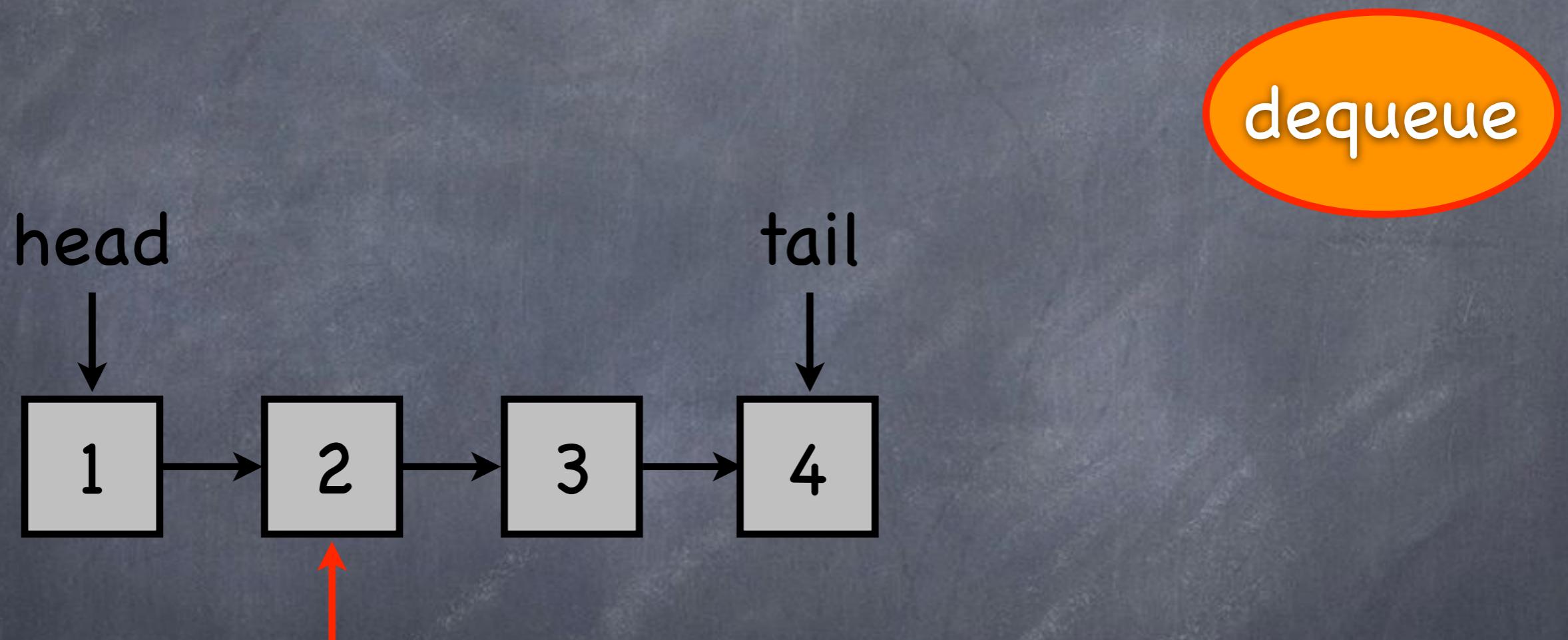
Concurrent 3-FIFO Queue (k=3)



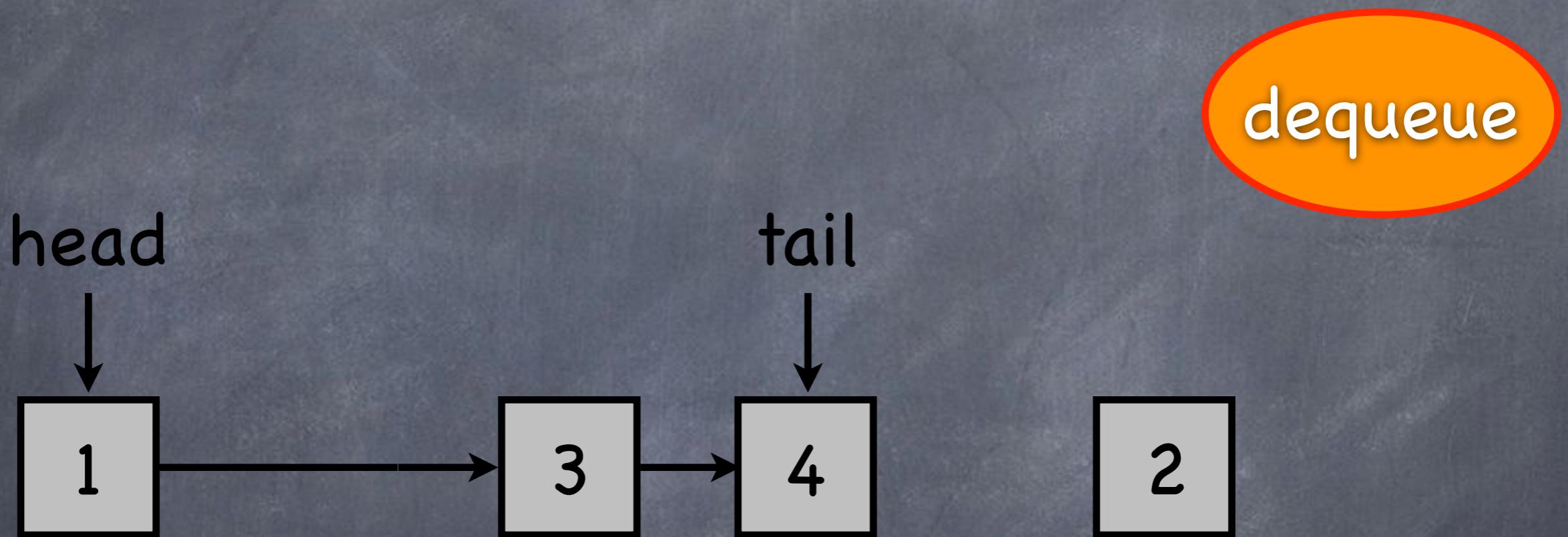
Concurrent 3-FIFO Queue (k=3)



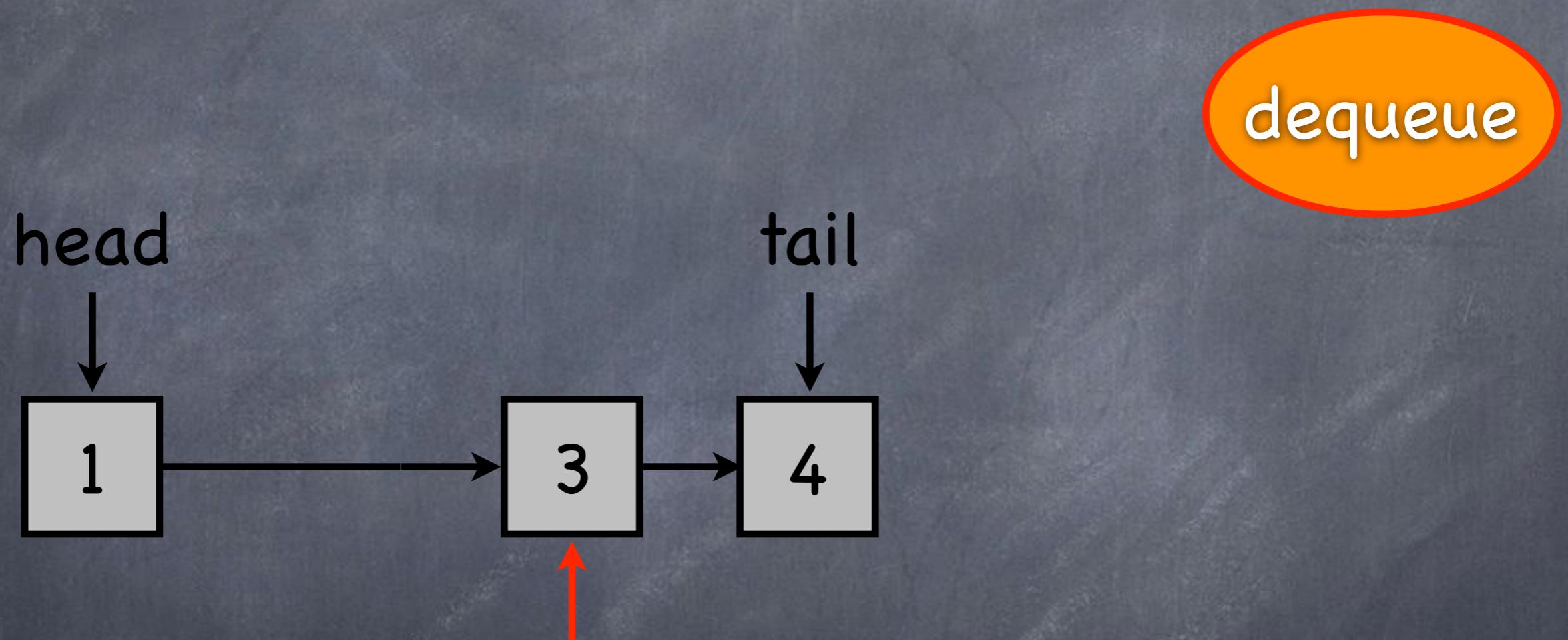
Concurrent 3-FIFO Queue (k=3)



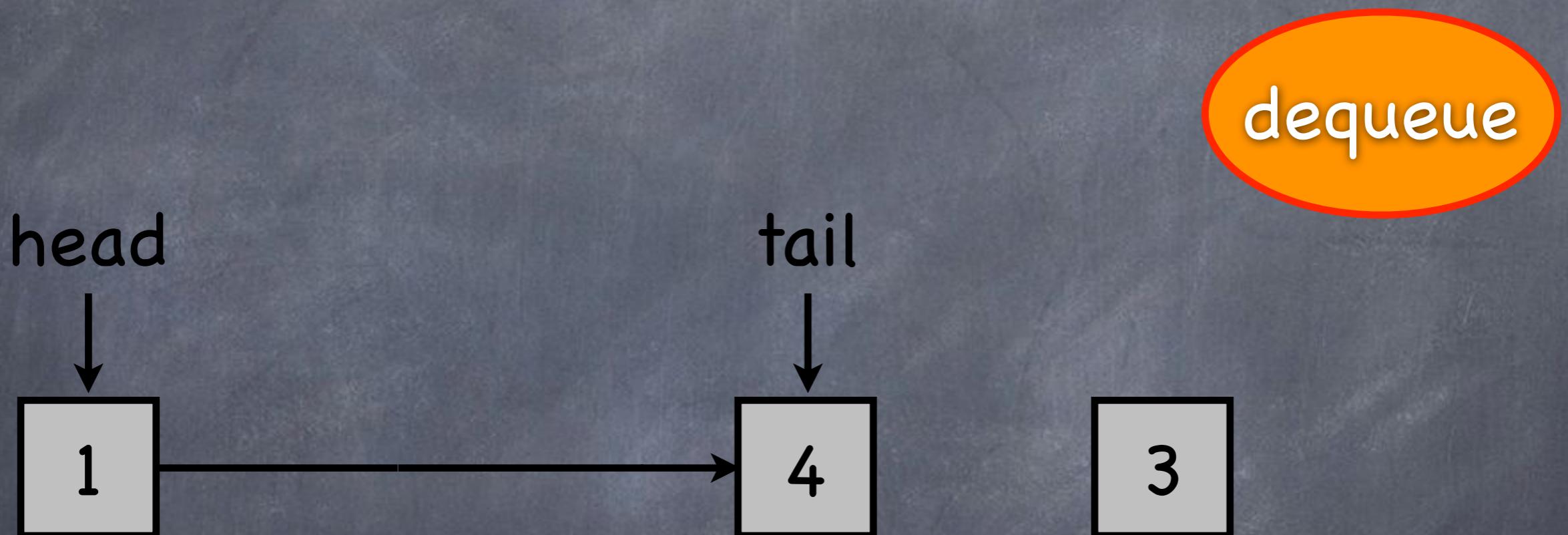
Concurrent 3-FIFO Queue (k=3)



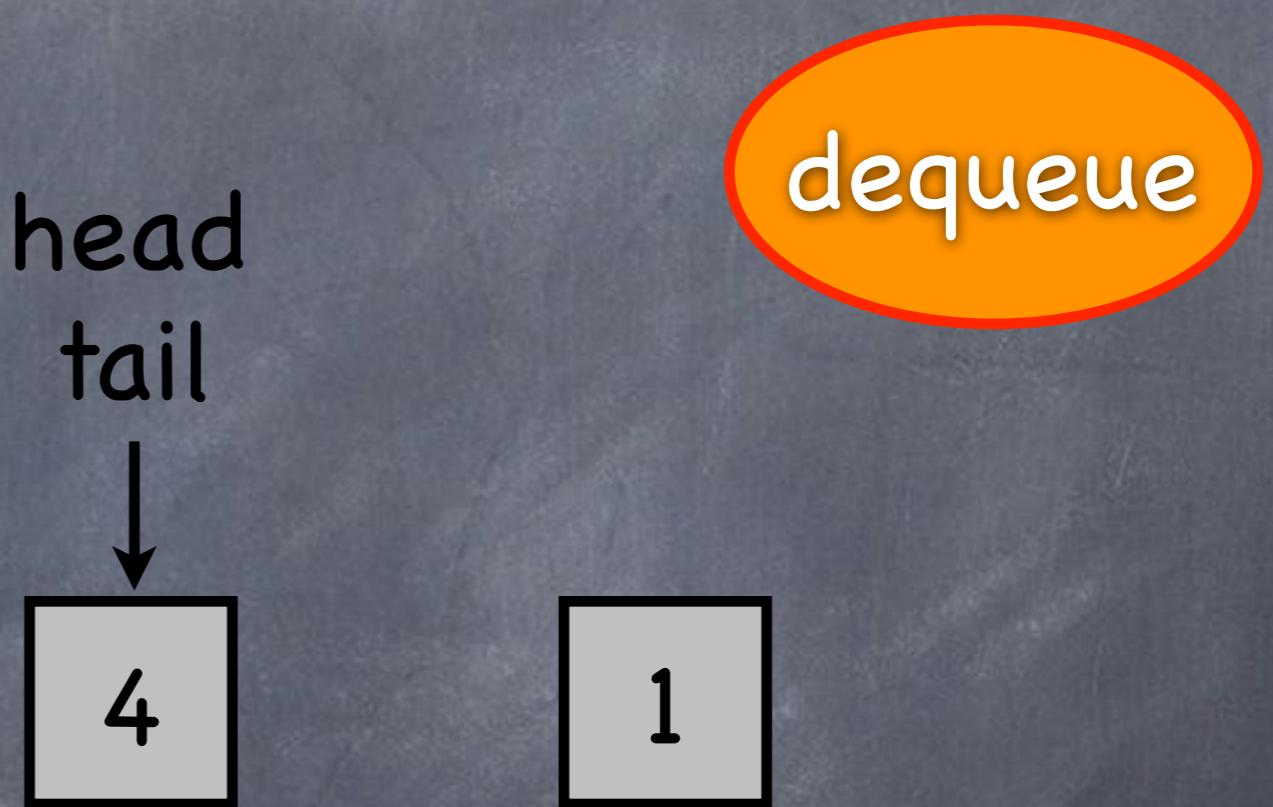
Concurrent 3-FIFO Queue (k=3)



Concurrent 3-FIFO Queue (k=3)



Concurrent 3-FIFO Queue (k=3)

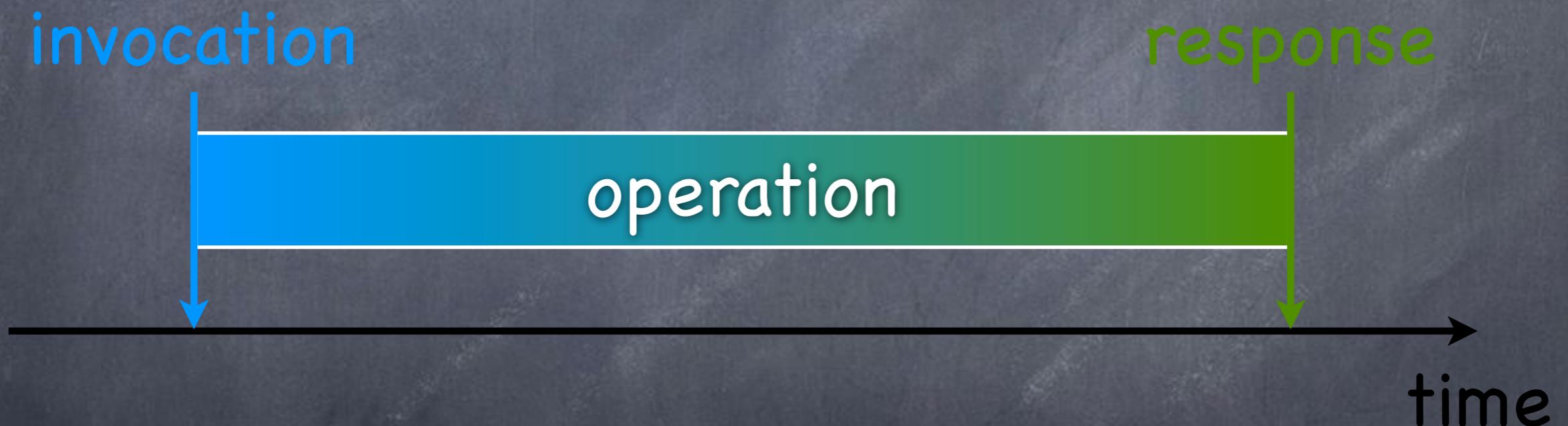


We call k
the worst-case semantical
deviation (WCSD) of
a k -FIFO queue from
a regular FIFO queue

The actual semantical deviation (ASD) is the semantical deviation of a **k**-FIFO queue when applied to a given workload

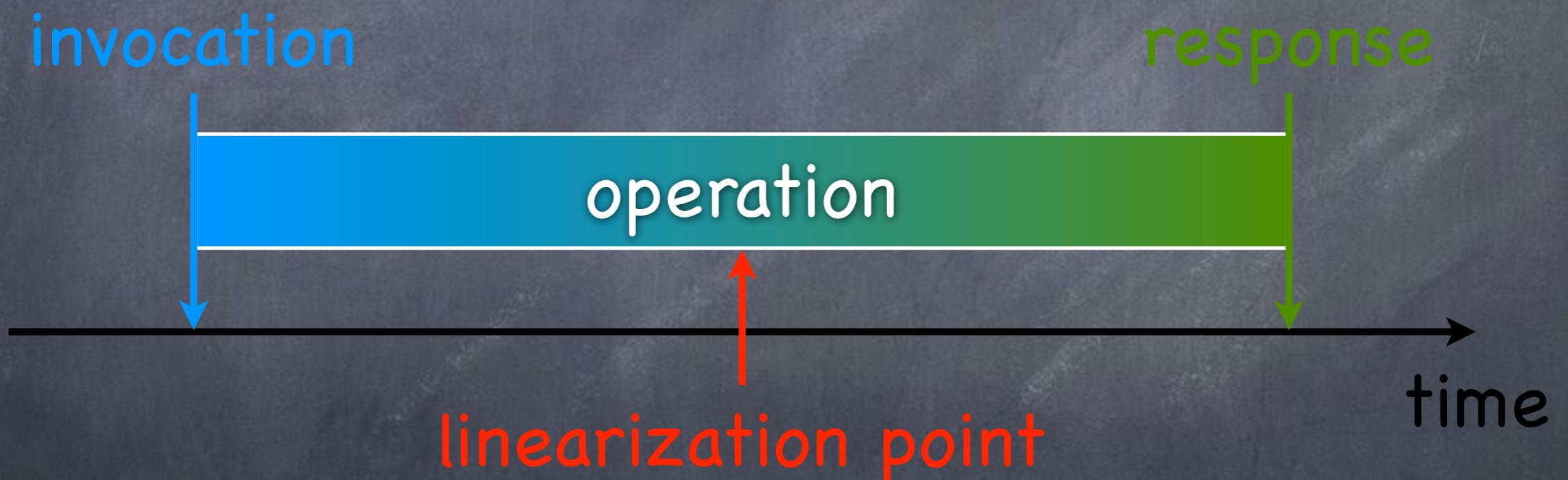
Execution History

Sequence of Time-Stamped Invocation and Response Events

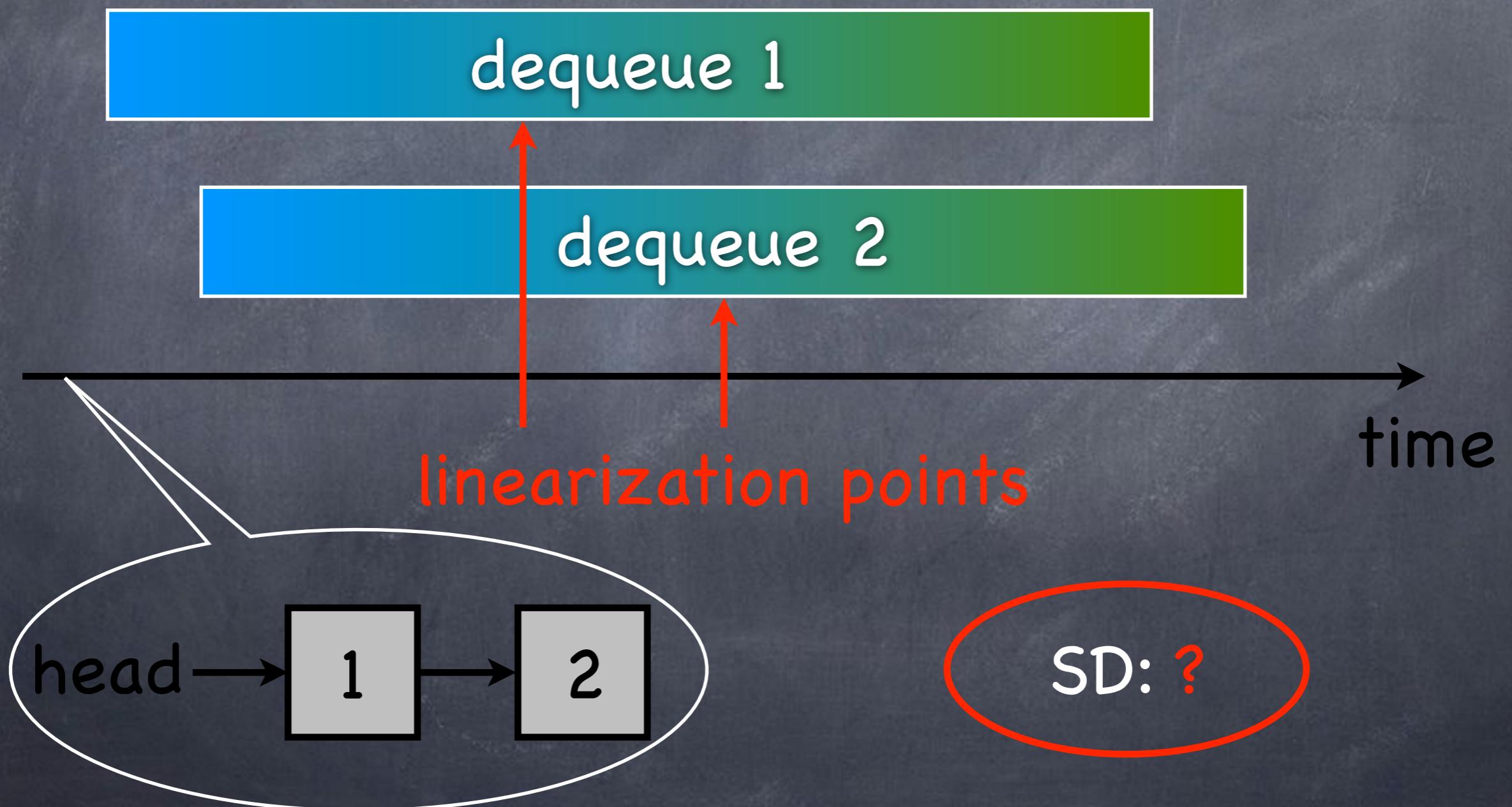


Execution History

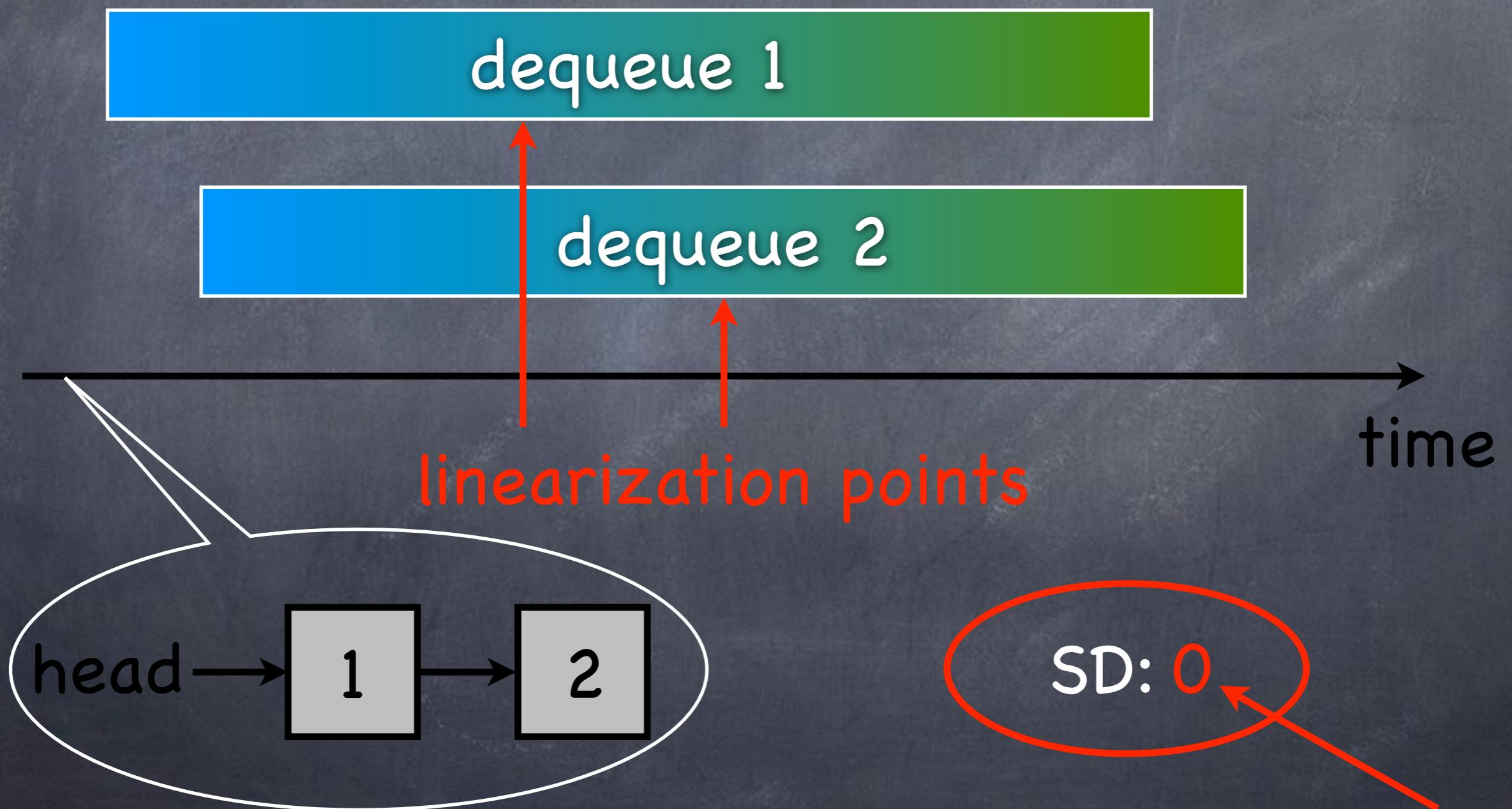
Sequence of Time-Stamped Invocation and Response Events



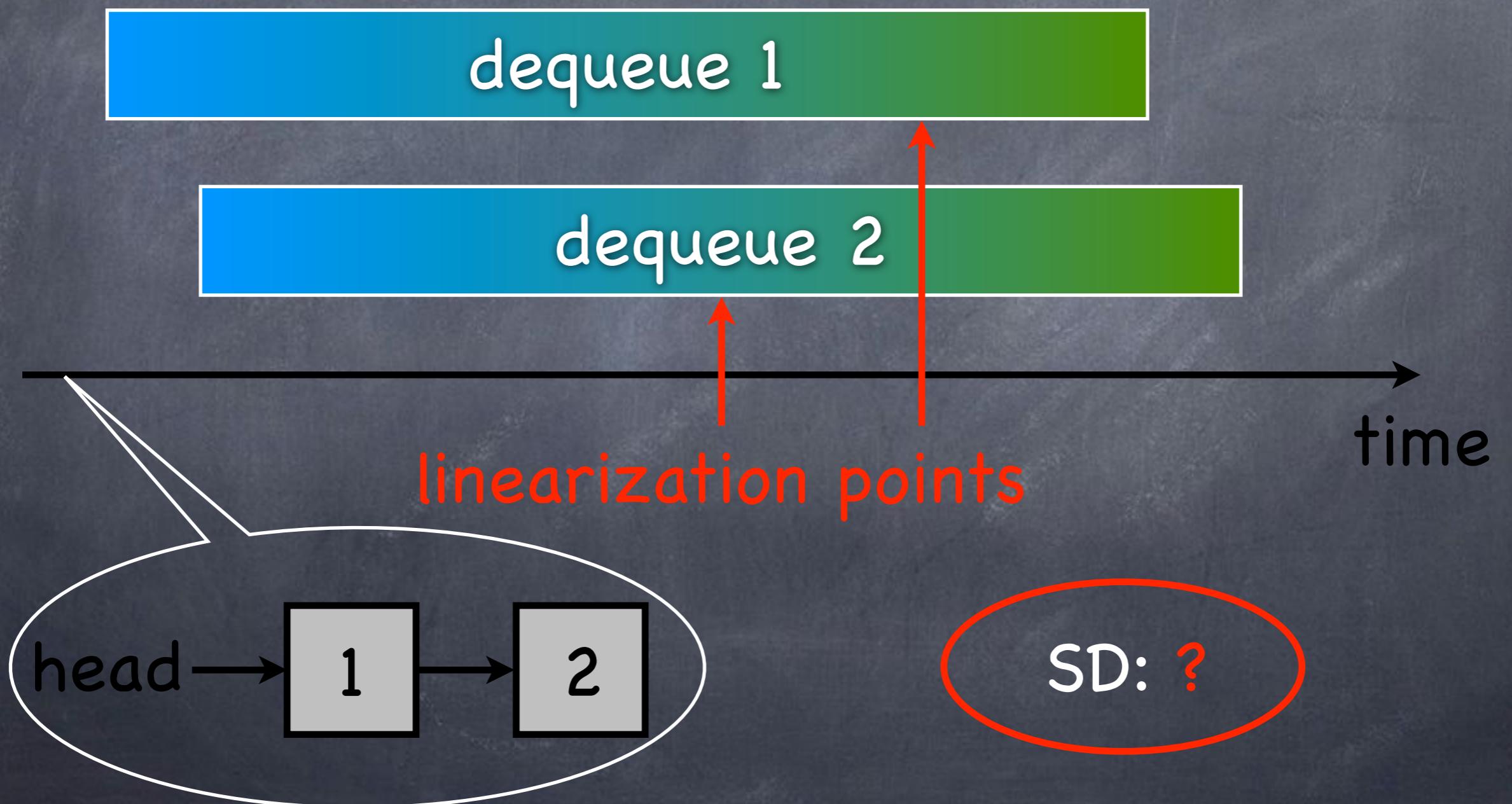
Measuring Semantical Deviation (SD)



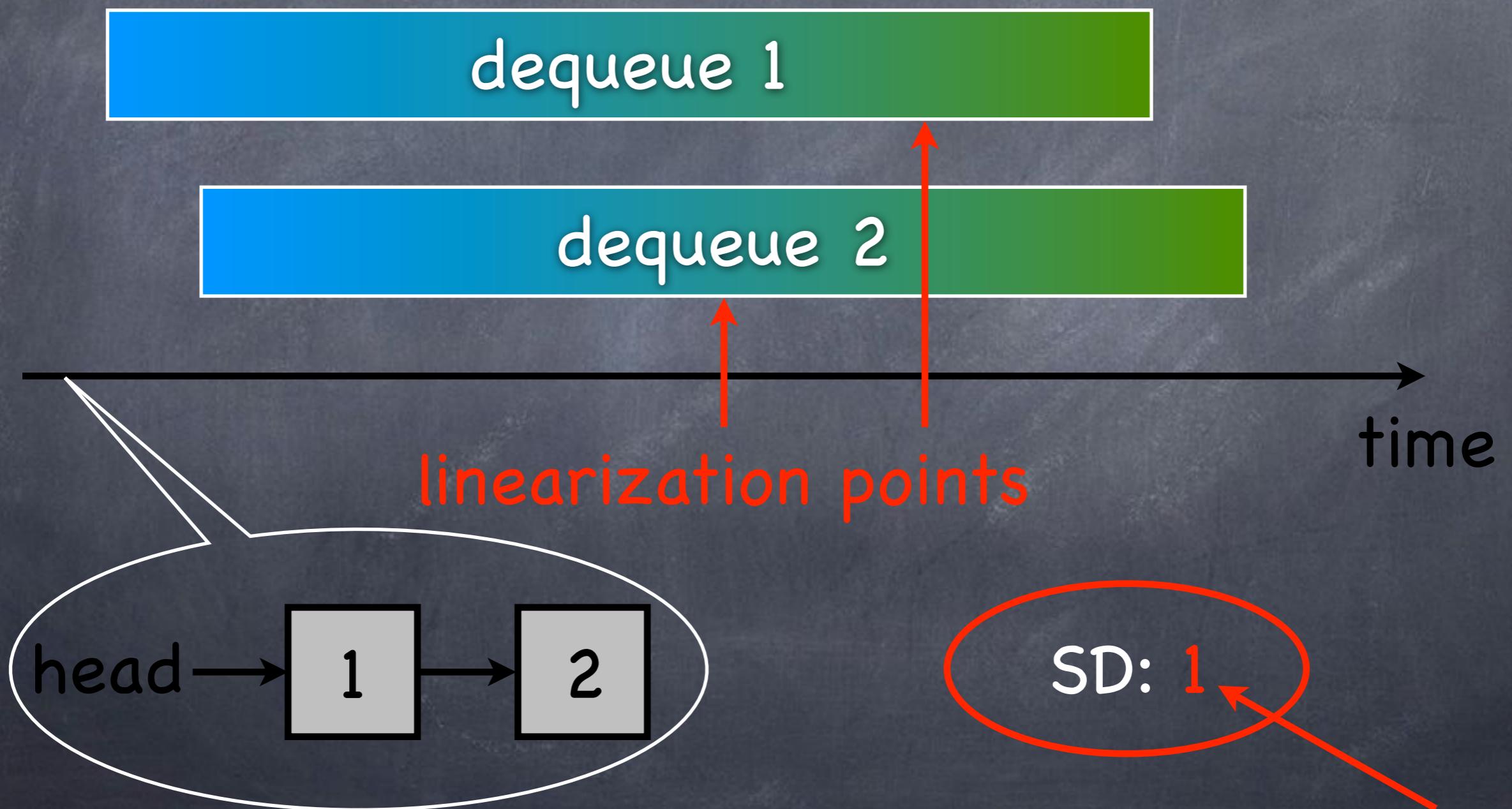
Measuring Semantical Deviation (SD)



Measuring Semantical Deviation (SD)

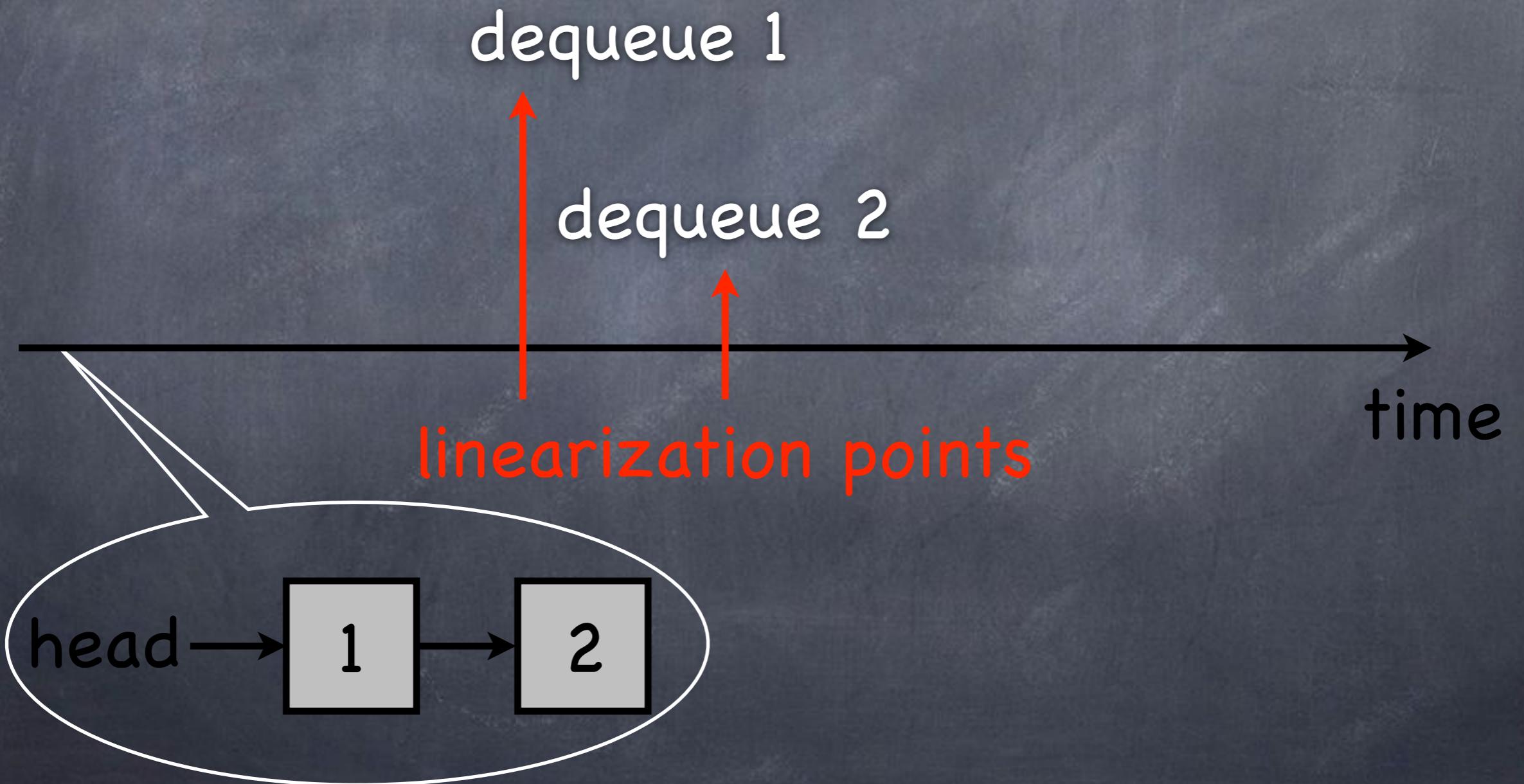


Measuring Semantical Deviation (SD)



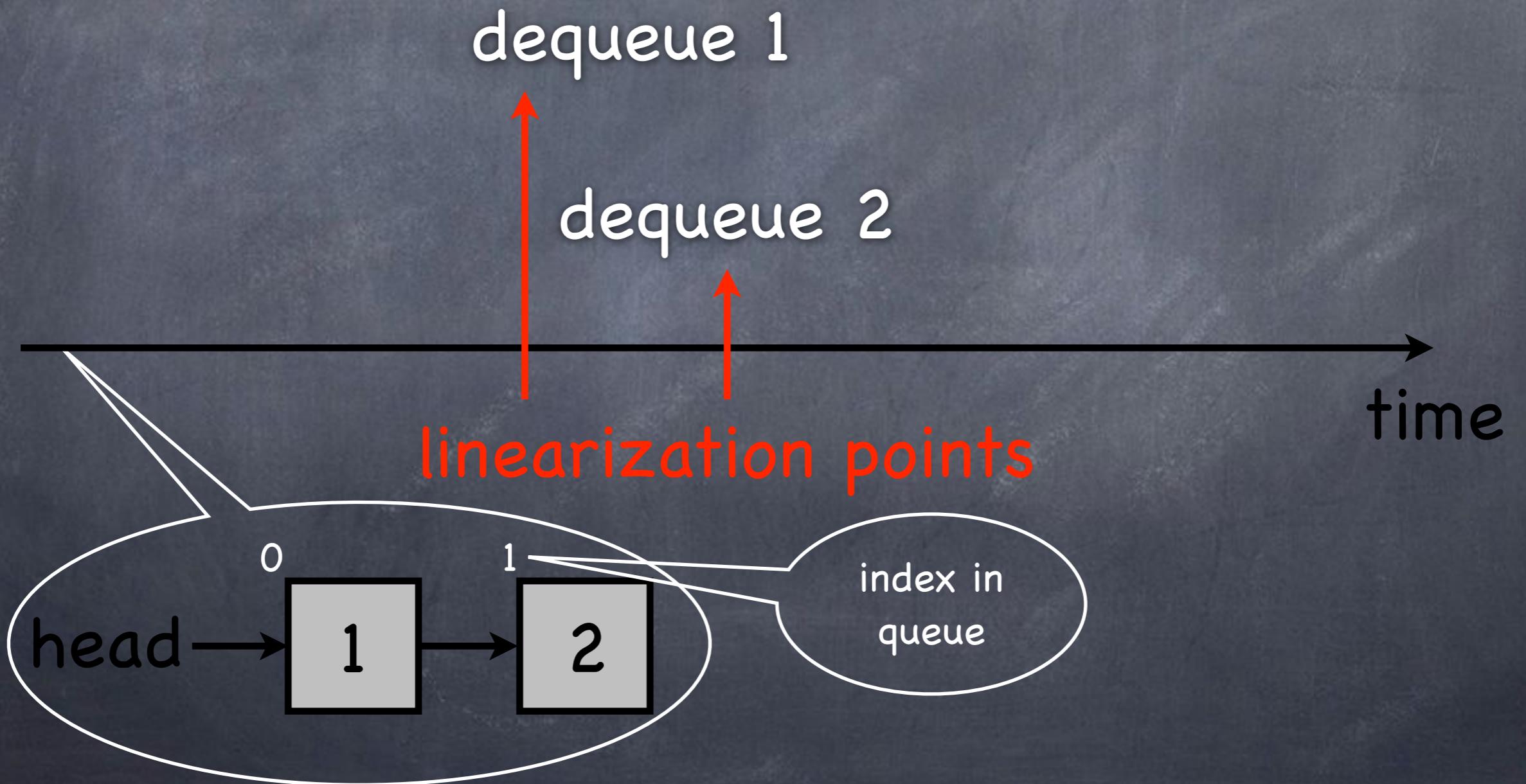
Sequential History

Sequence of Operations (Linearization Points)



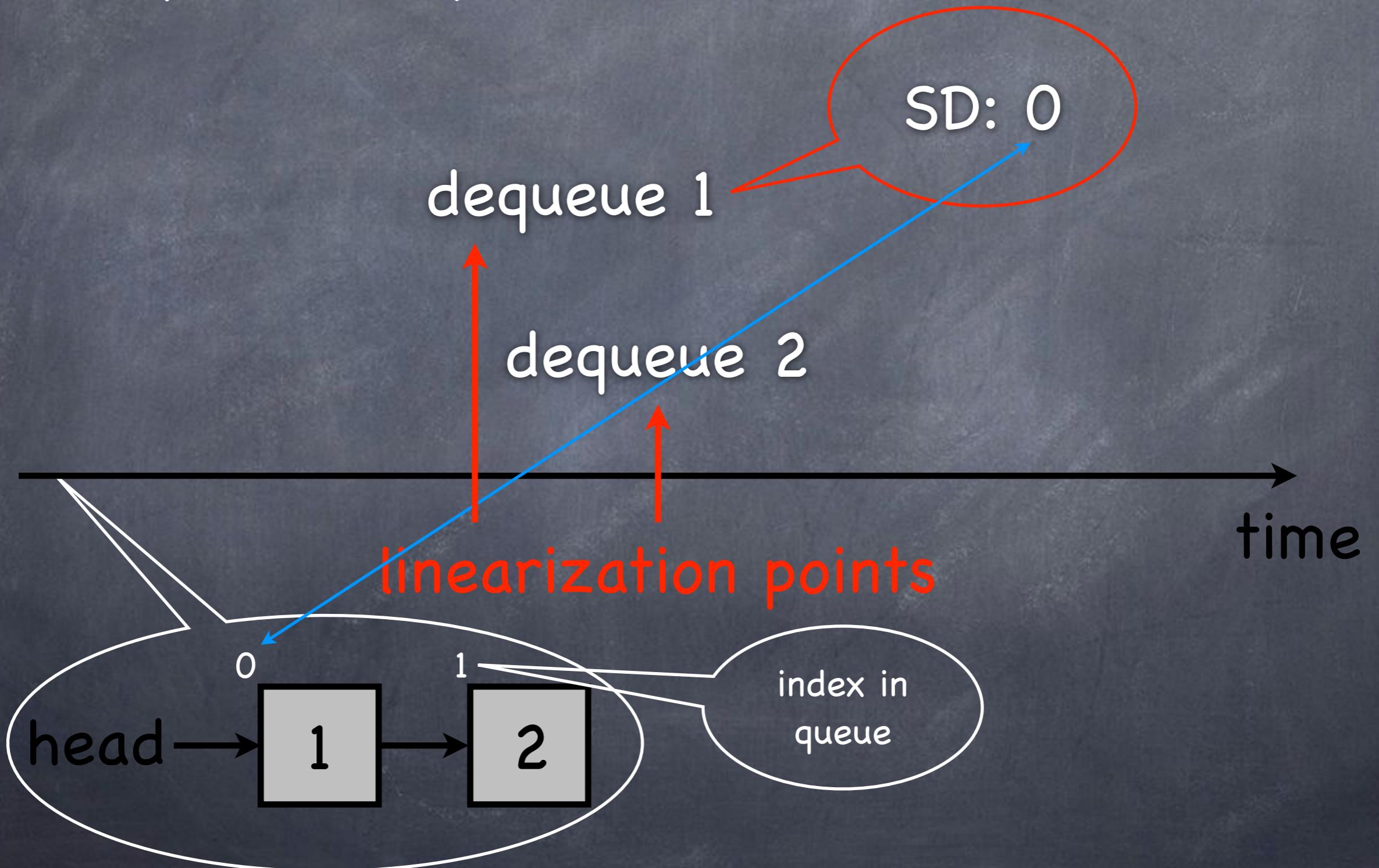
Sequential History

Sequence of Operations (Linearization Points)



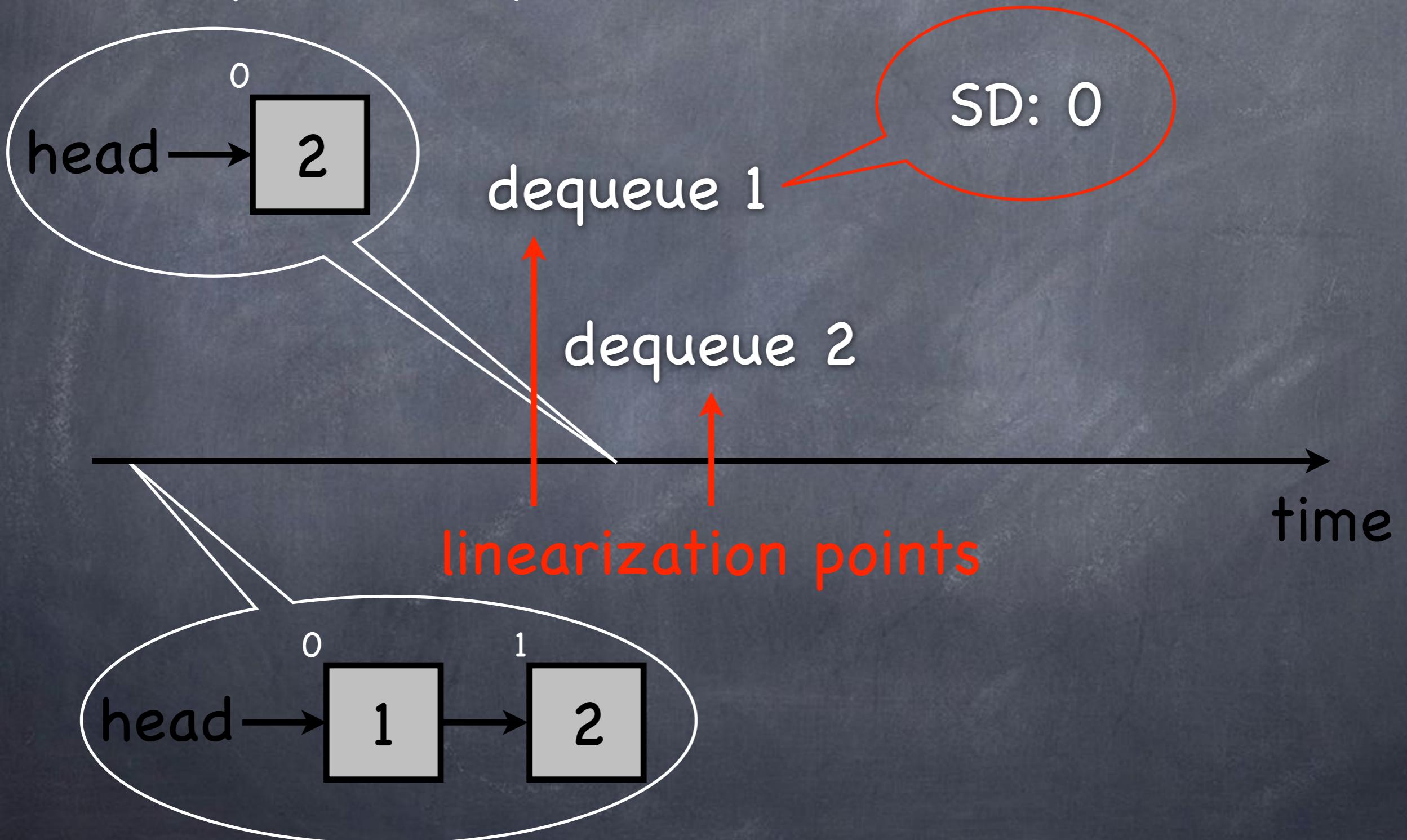
Sequential History

Sequence of Operations (Linearization Points)



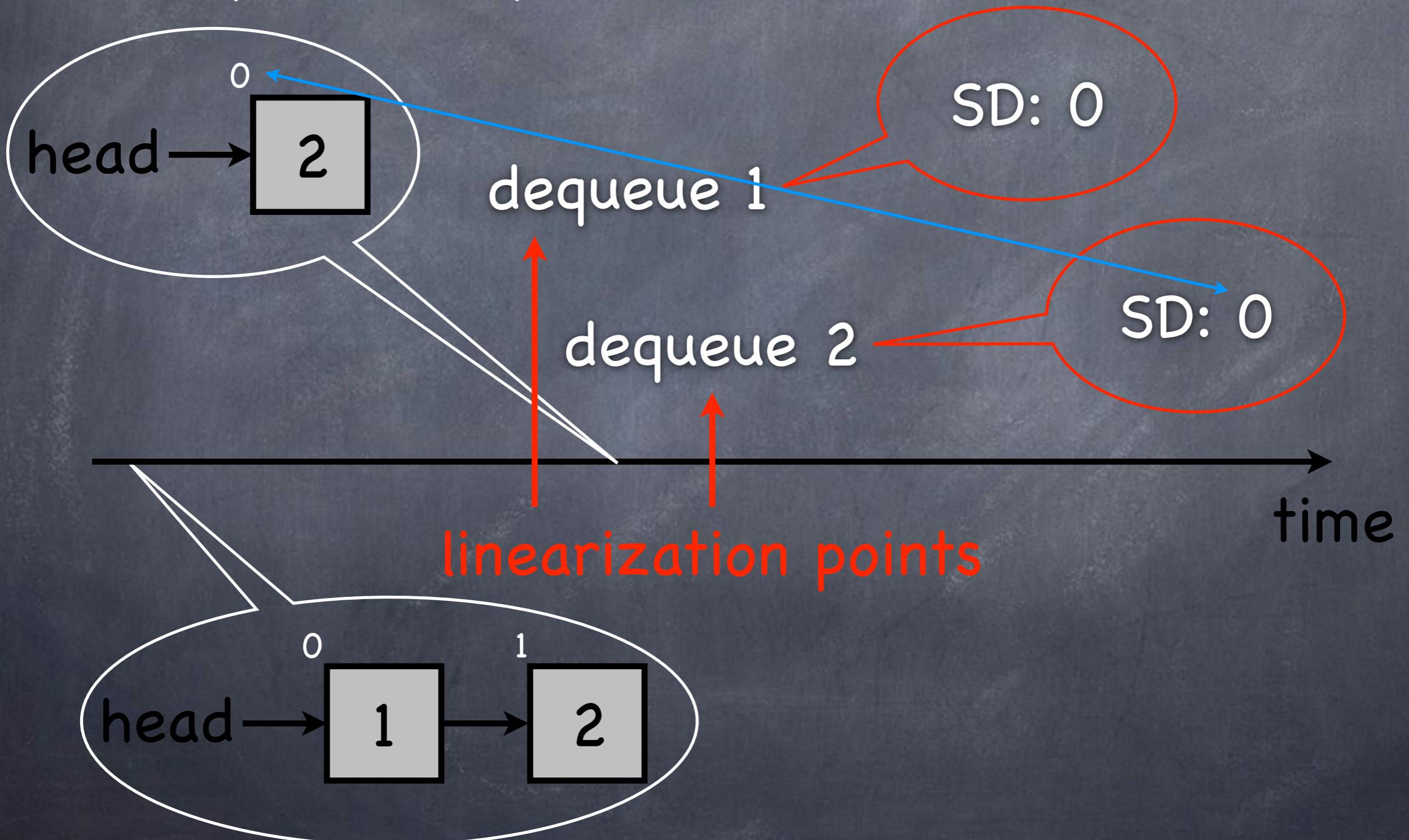
Sequential History

Sequence of Operations (Linearization Points)



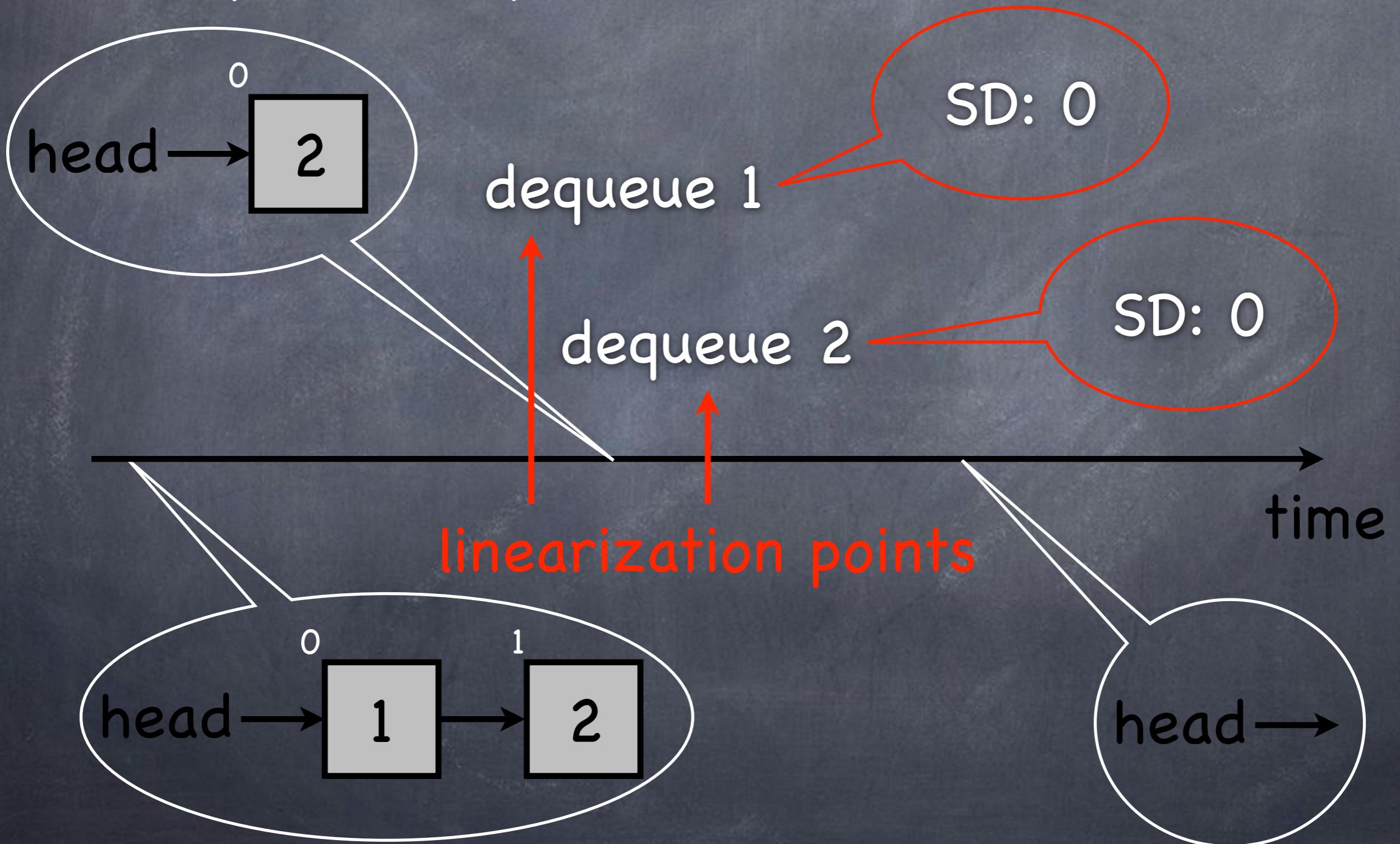
Sequential History

Sequence of Operations (Linearization Points)



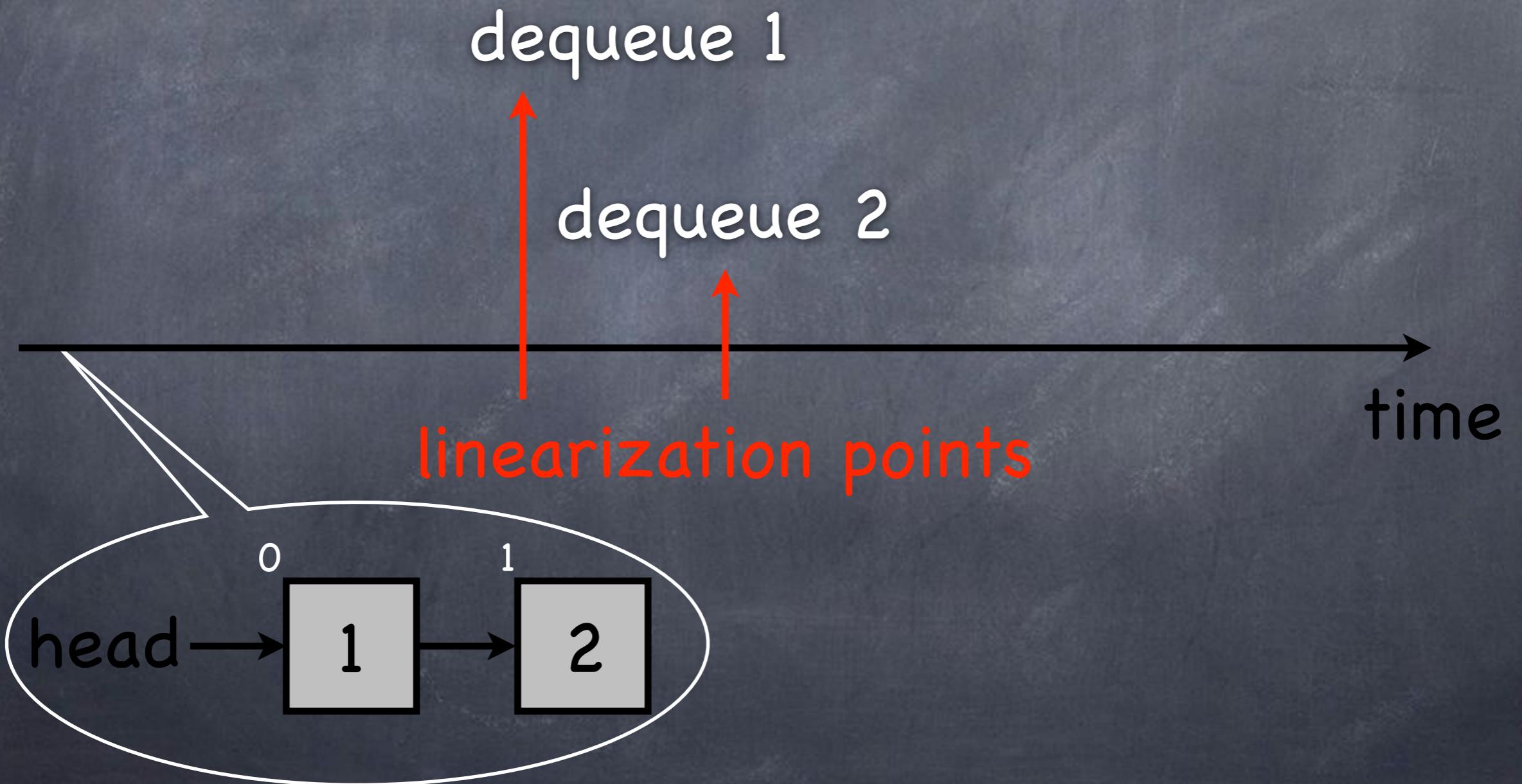
Sequential History

Sequence of Operations (Linearization Points)



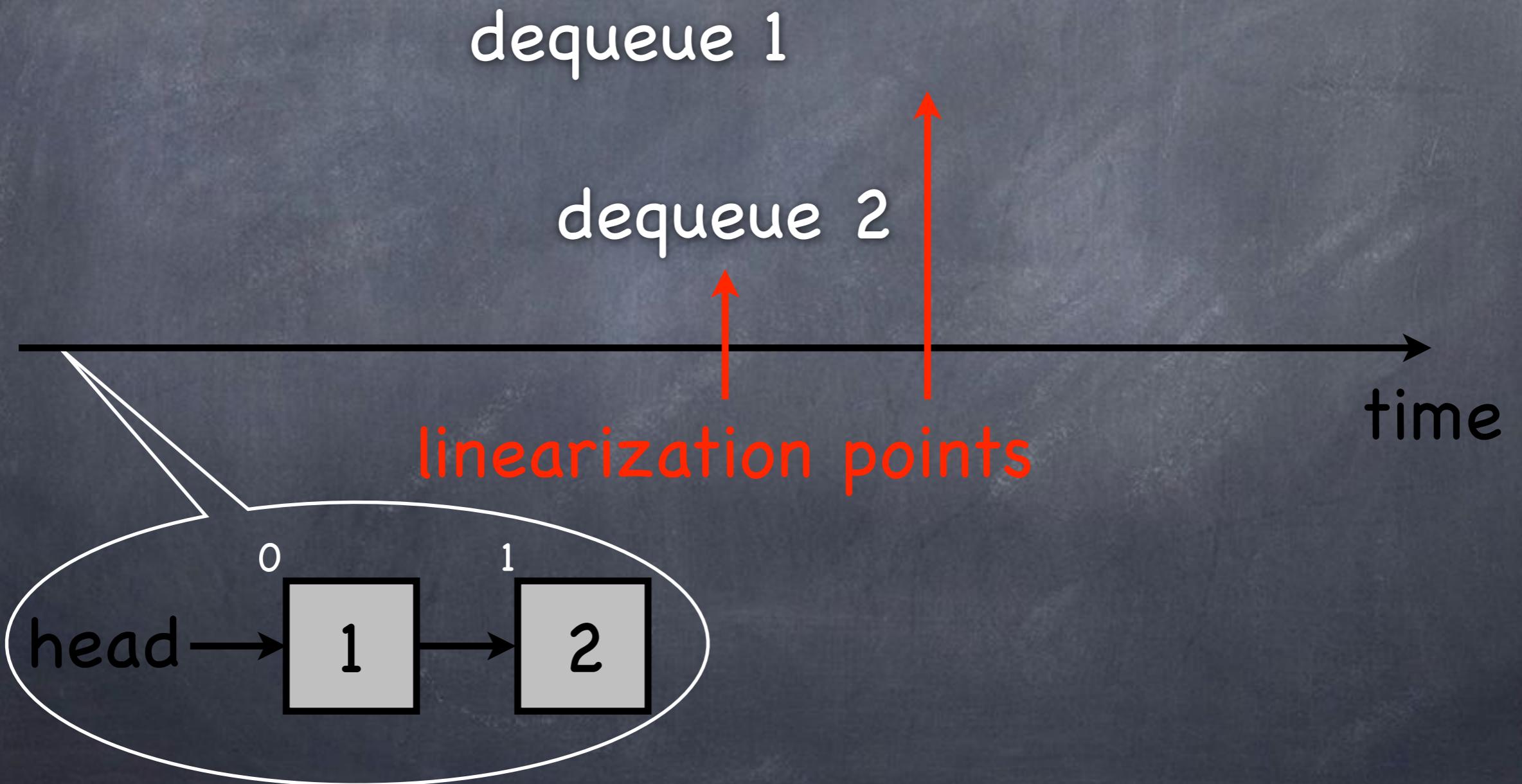
Sequential History II

Sequence of Operations (Linearization Points)



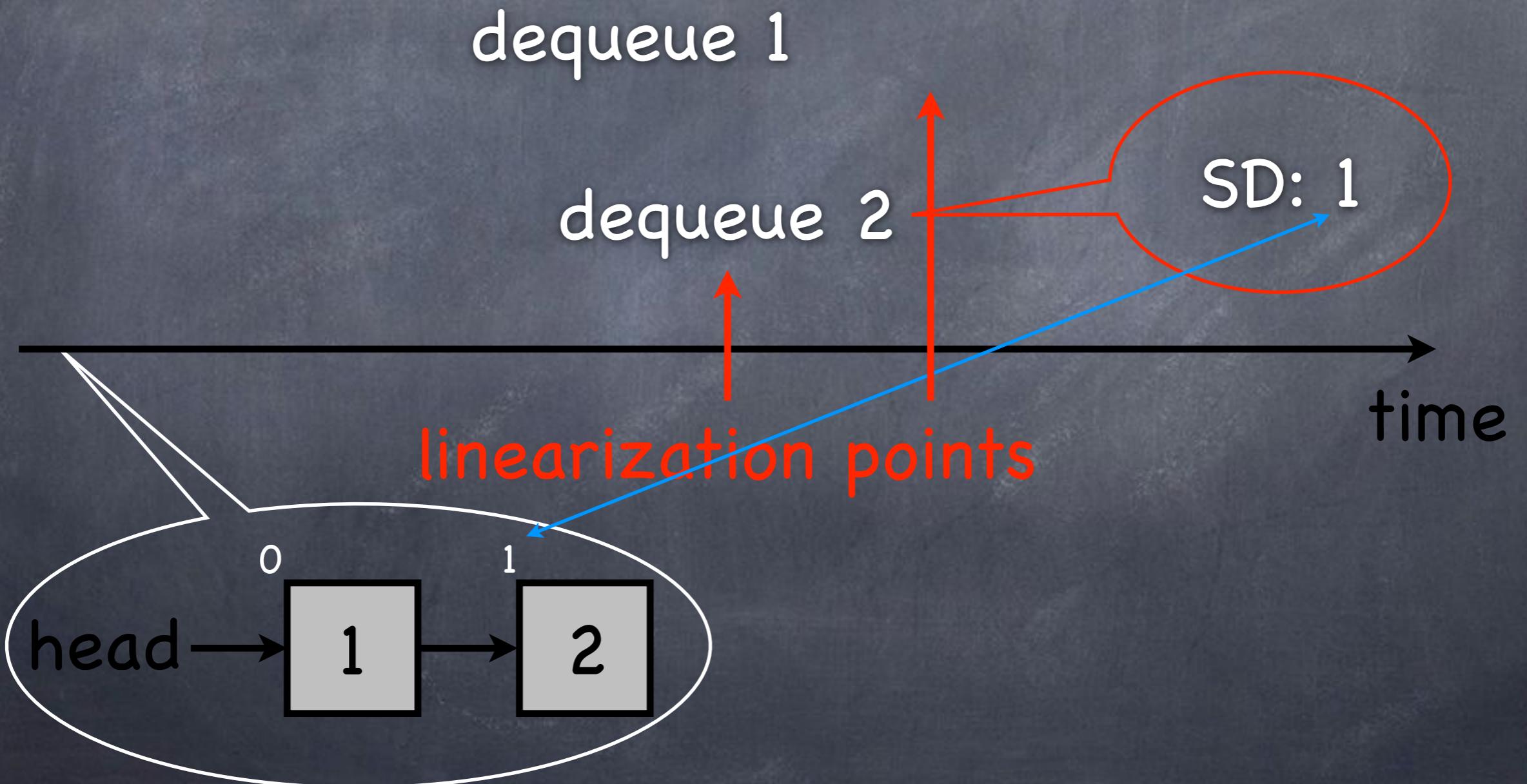
Sequential History II

Sequence of Operations (Linearization Points)



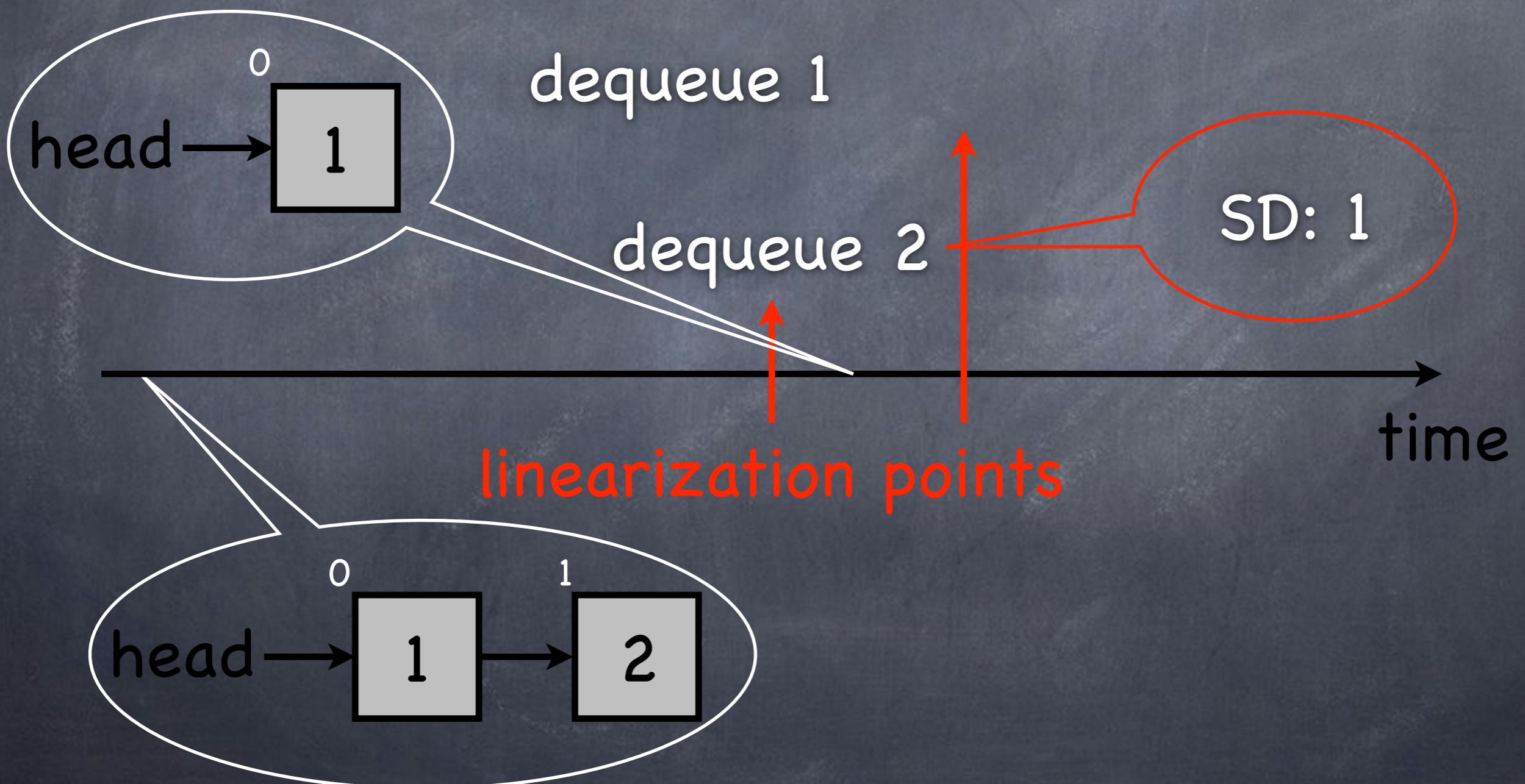
Sequential History II

Sequence of Operations (Linearization Points)



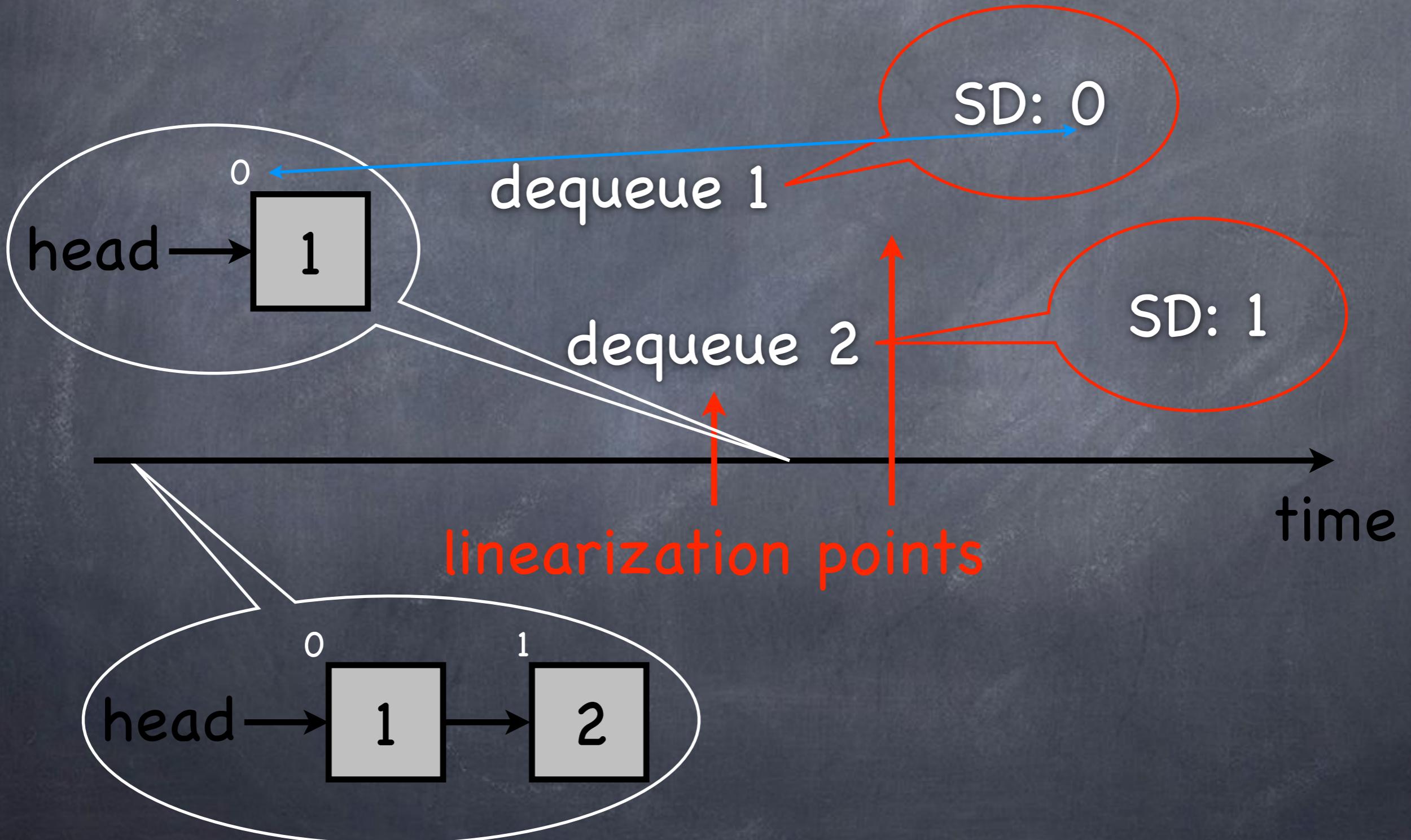
Sequential History II

Sequence of Operations (Linearization Points)



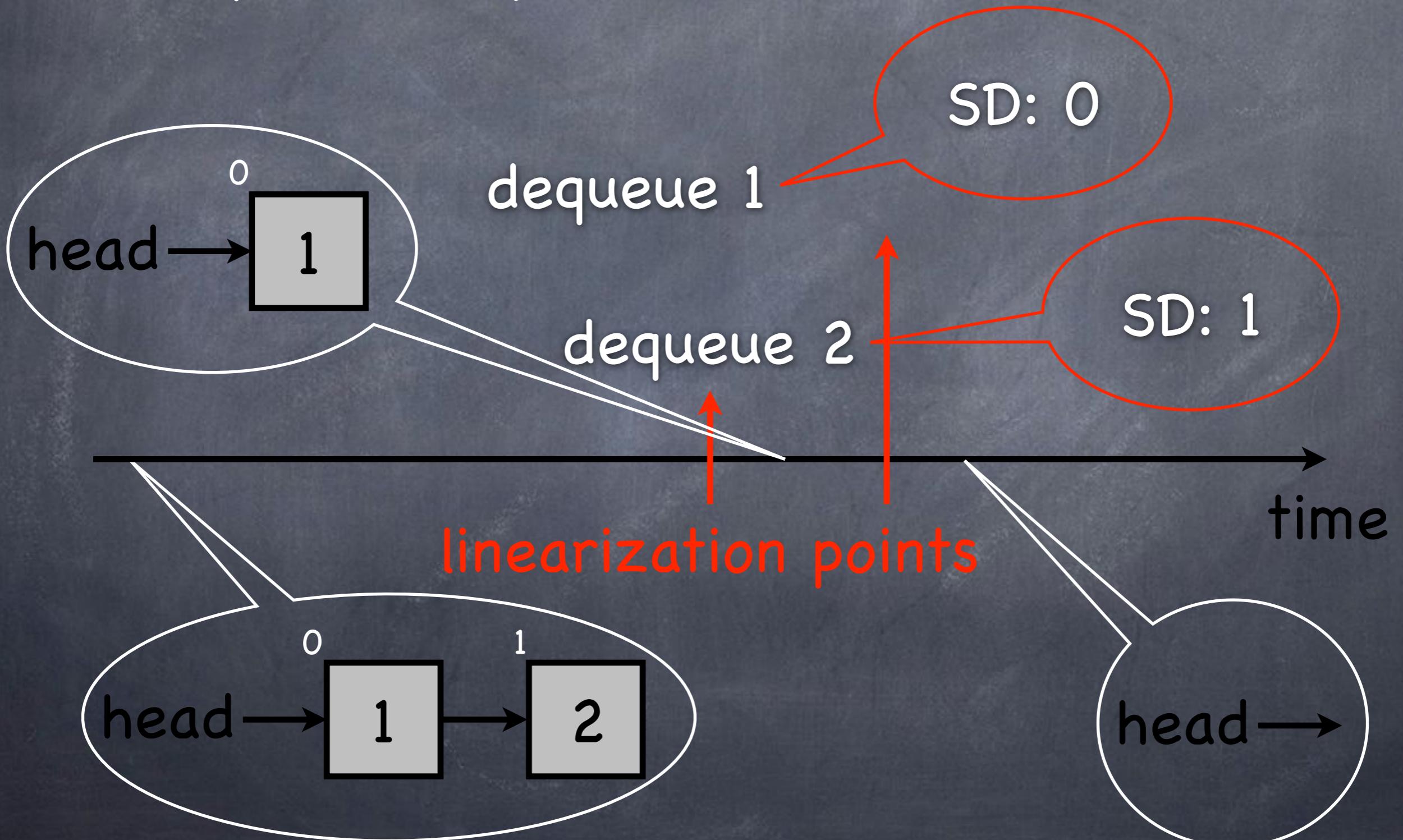
Sequential History II

Sequence of Operations (Linearization Points)

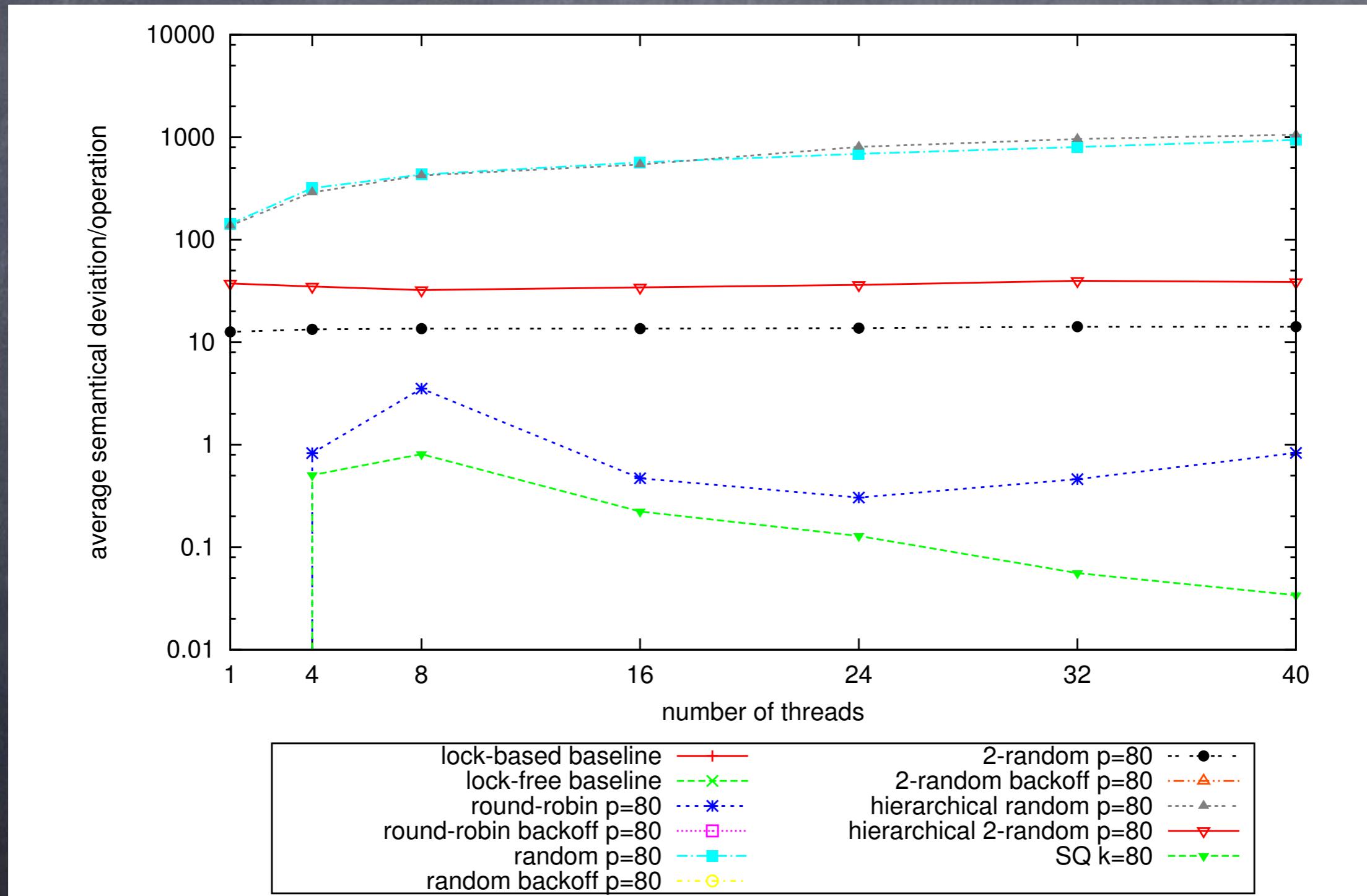


Sequential History II

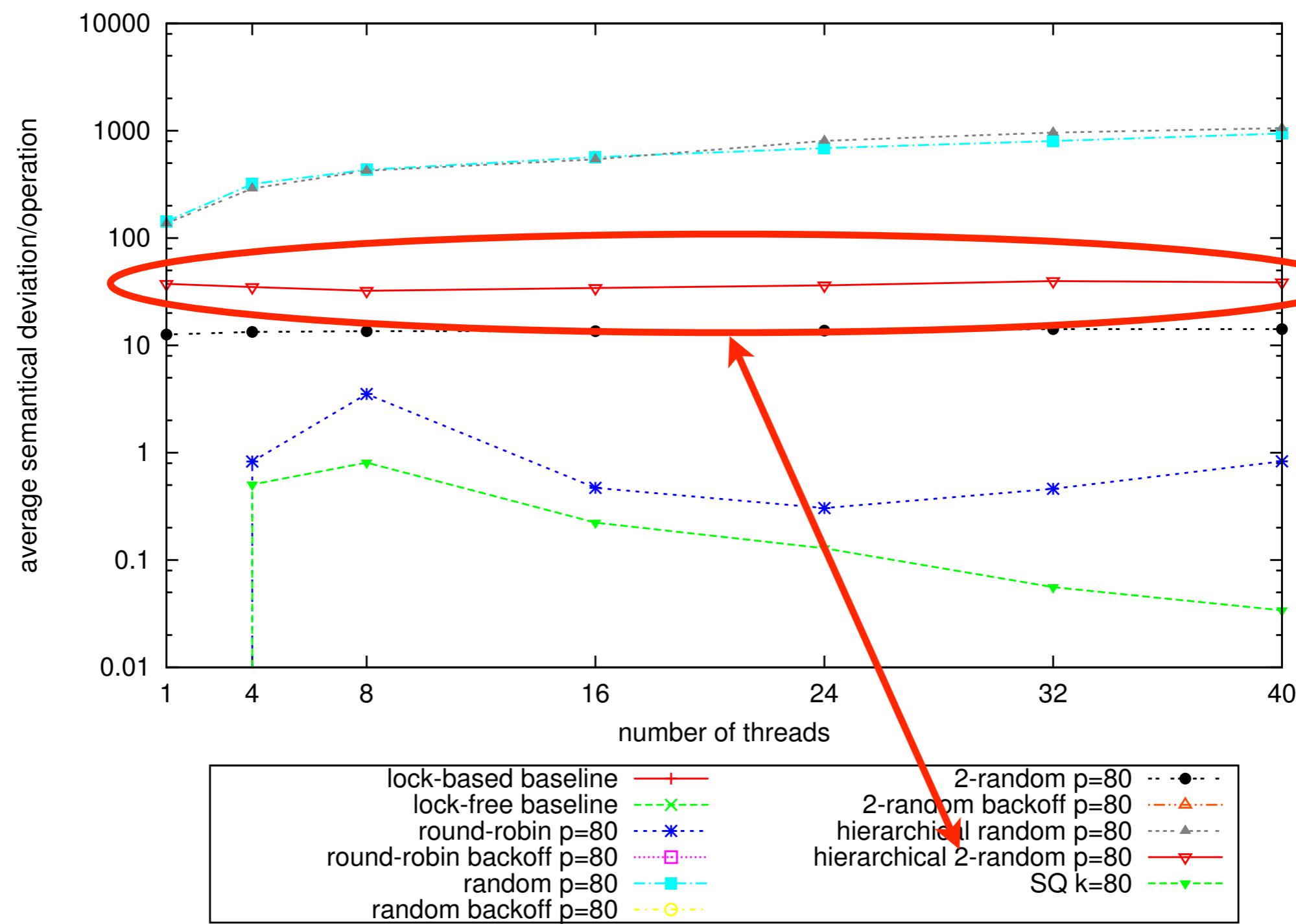
Sequence of Operations (Linearization Points)



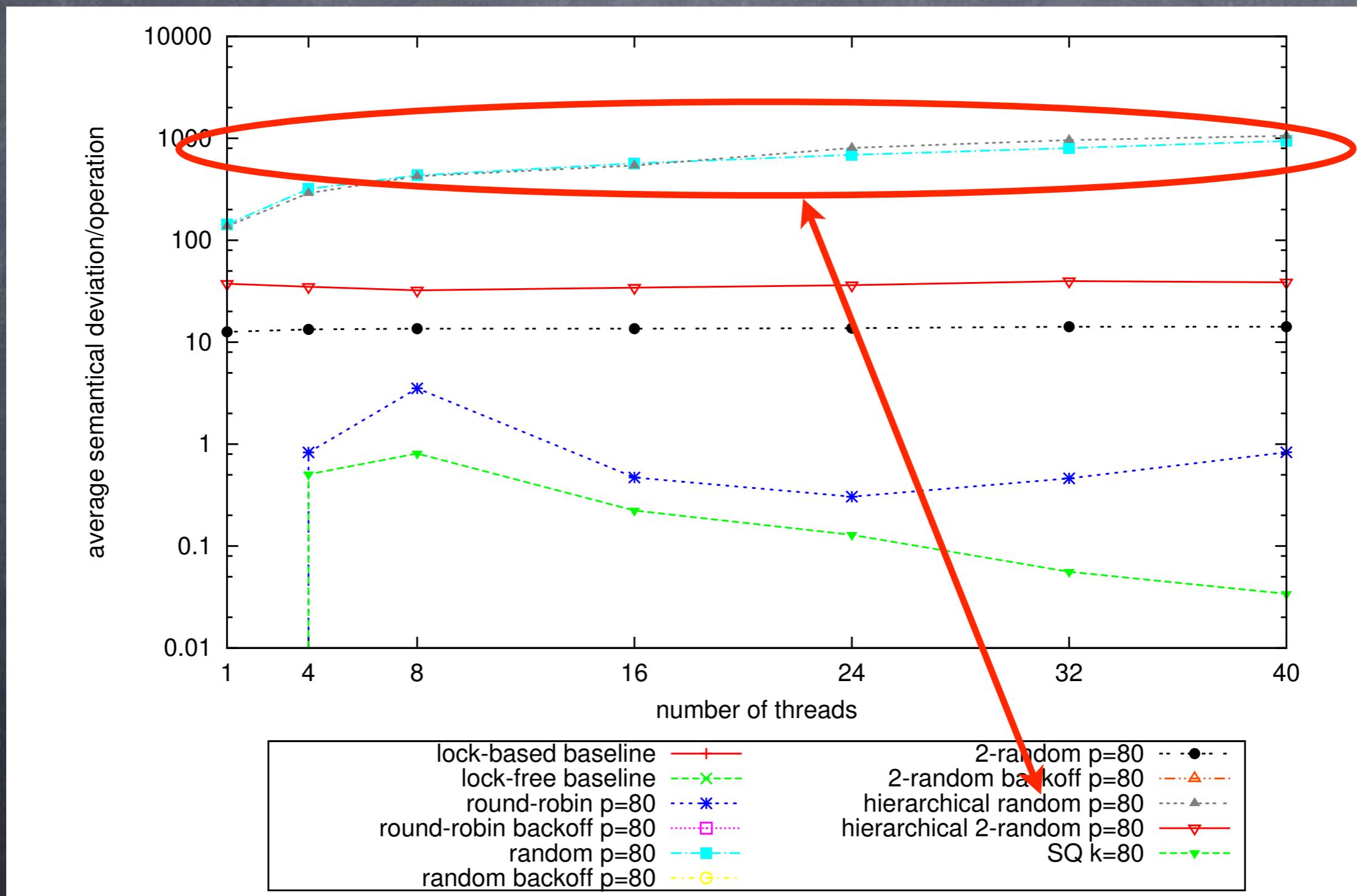
Average Semantical Deviation



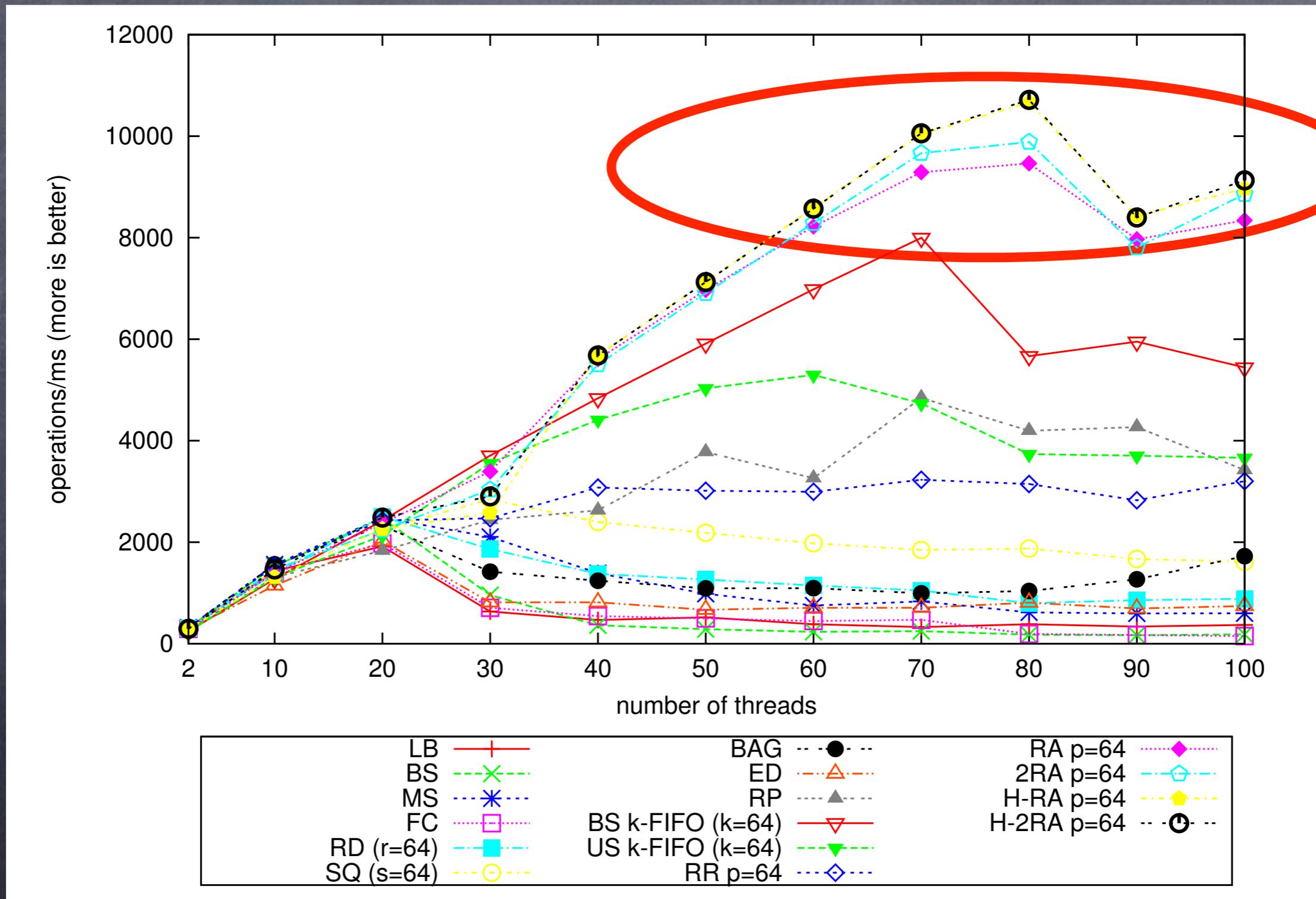
Here k may be around 40
on average: **best tradeoff**



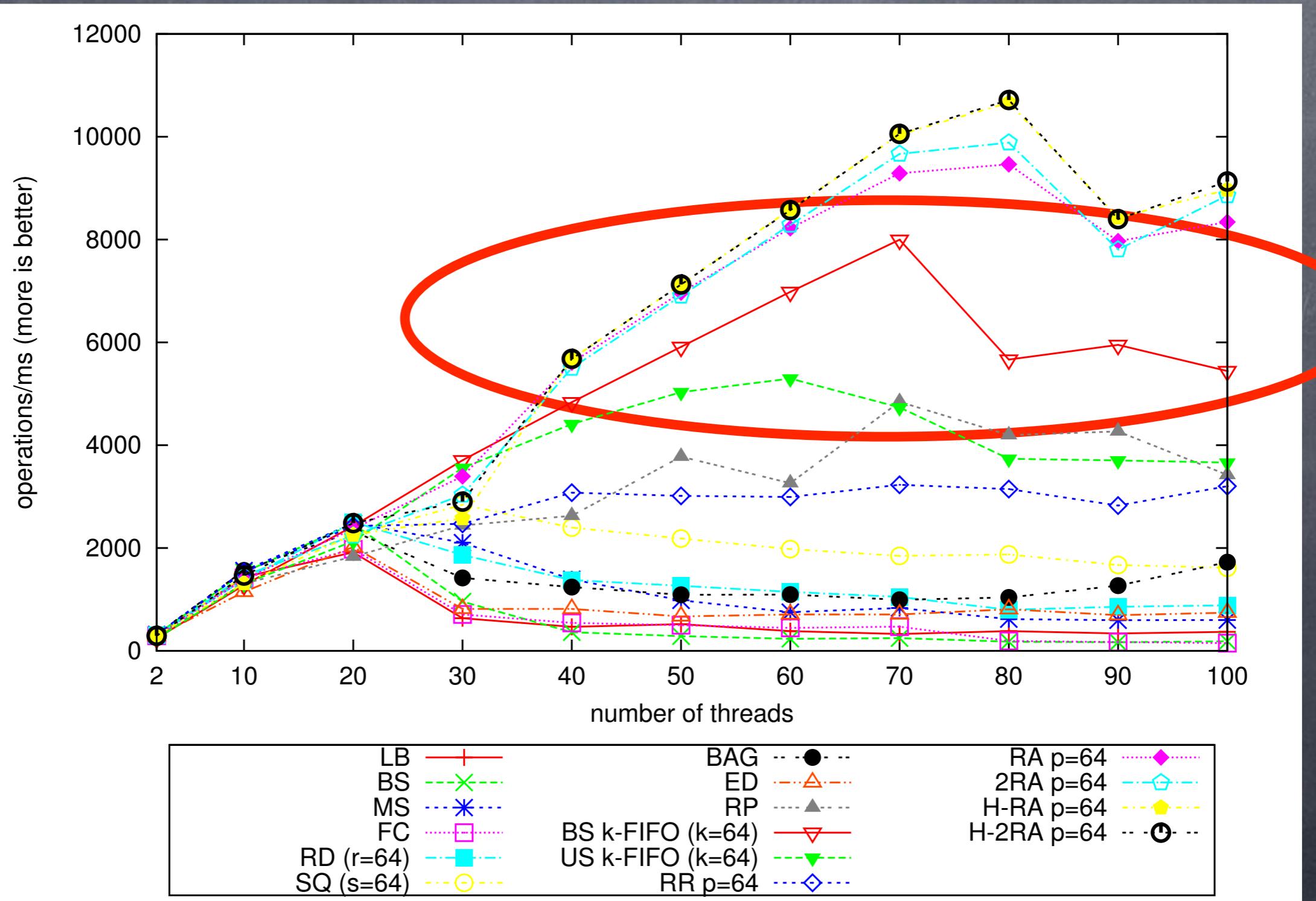
Whereas here κ is one order
of magnitude bigger w/o gain



Random vs. d-Random



Segmented Queues



Back to Correctness?

Questions?

