Incorrect Systems: It’s not the Problem, It’s the Solution

Christoph Kirsch
Universität Salzburg

DREAMS Seminar, Berkeley, July 2012

Software

Software/
Hardware

Hardware

Software

Software/
Hardware

Hardware

Krishna Palem
Rice

Software

Software/
Hardware

Probabilistic or
Approximate
Computing

Krishna Palem
Rice

Software

Software/
Hardware

Rakesh Kumar
UIUC

Probabilistic or
Approximate
Computing

Krishna Palem
Rice

Software

Stochastic Processors

Rakesh Kumar
UIUC

Probabilistic or Approximate Computing

Krishna Palem
Rice

Software

Martin Rinard
MIT

Stochastic Processors

Rakesh Kumar
UIUC

Probabilistic or Approximate Computing

Krishna Palem
Rice

Program Transformation

Martin Rinard
MIT

Stochastic Processors

Rakesh Kumar
UIUC

Probabilistic or Approximate Computing

Krishna Palem
Rice

Program Transformation

1. memory leaks
2. addressing errors
3. infinite loops

Stochastic Processors

Rakesh Kumar
UIUC

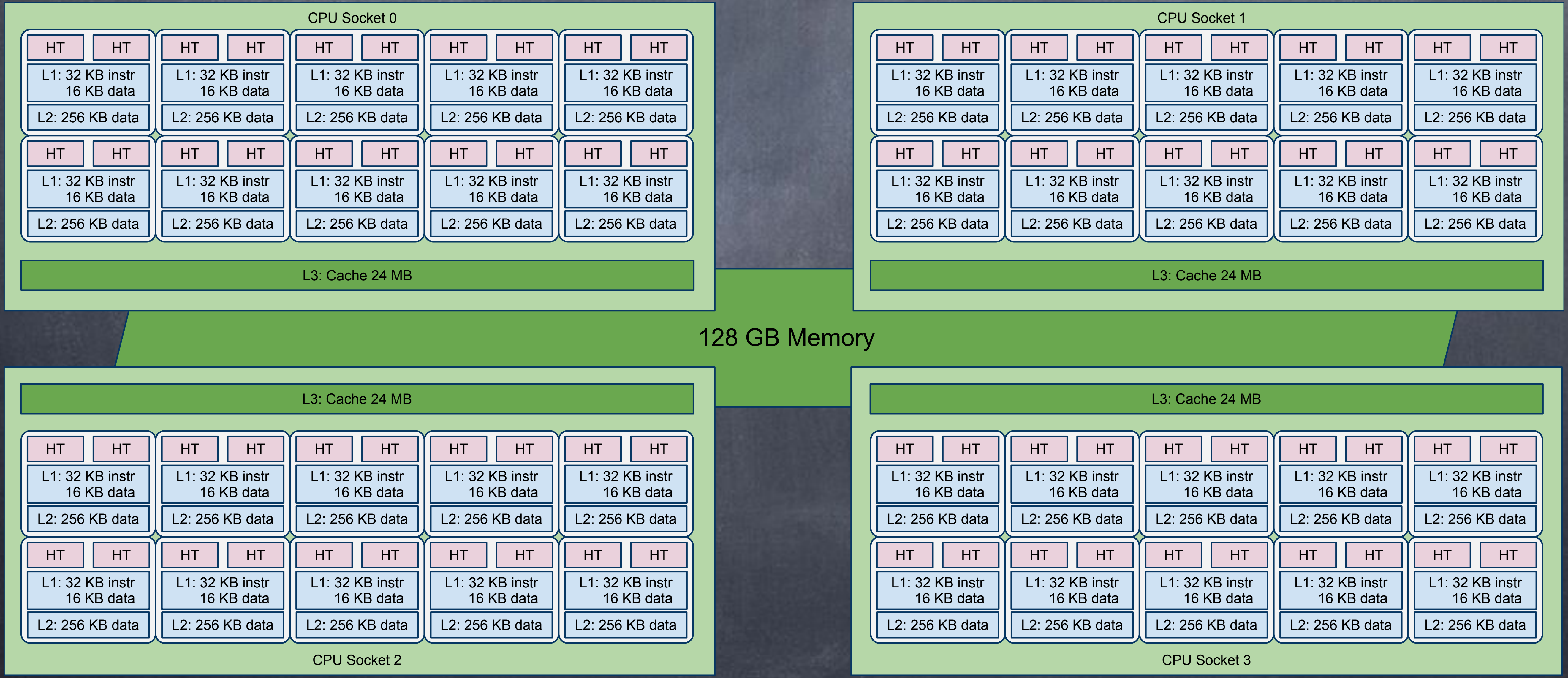
Probabilistic or Approximate Computing

Krishna Palem
Rice

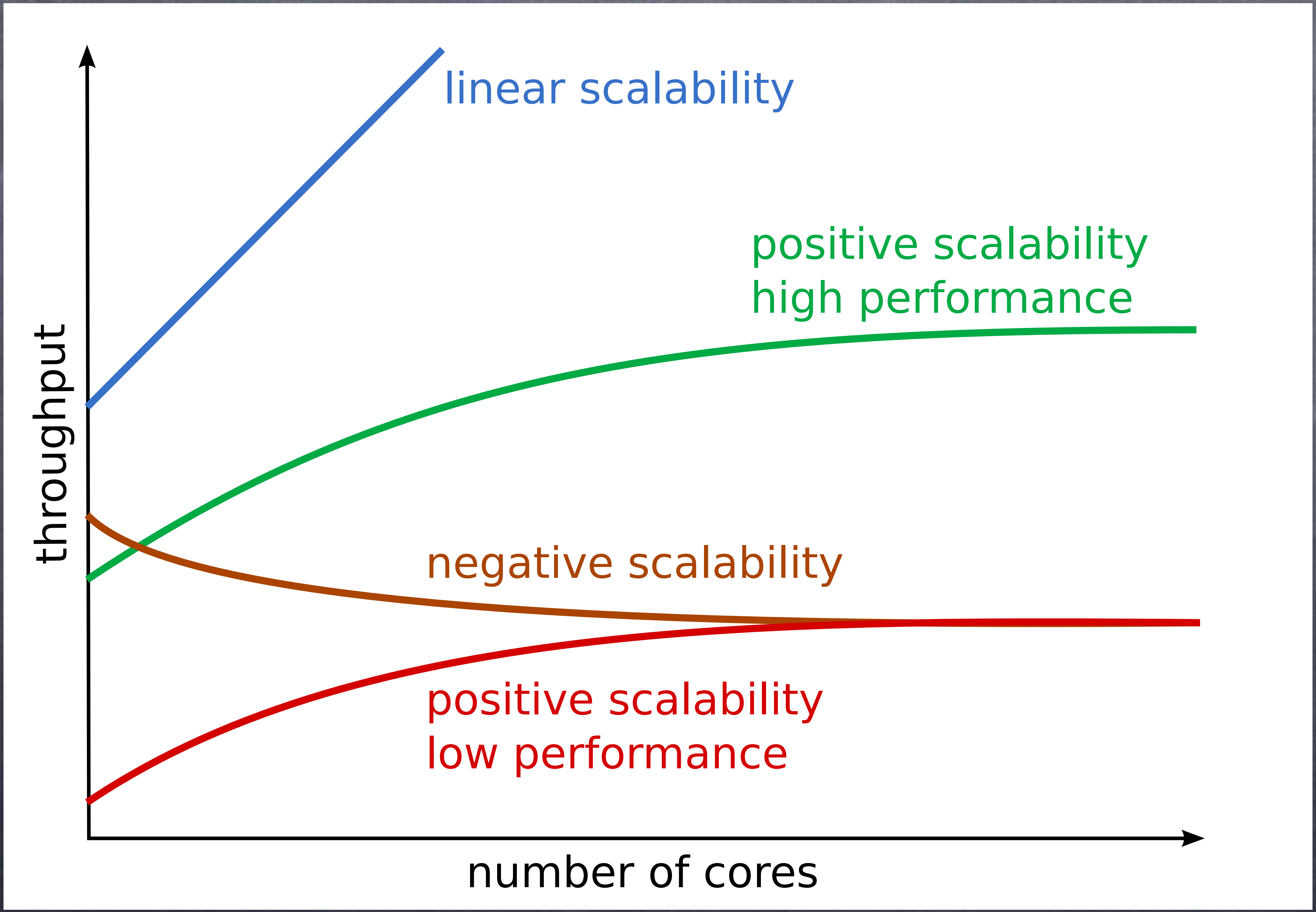
Metrics of Correctness in Systems Engineering

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

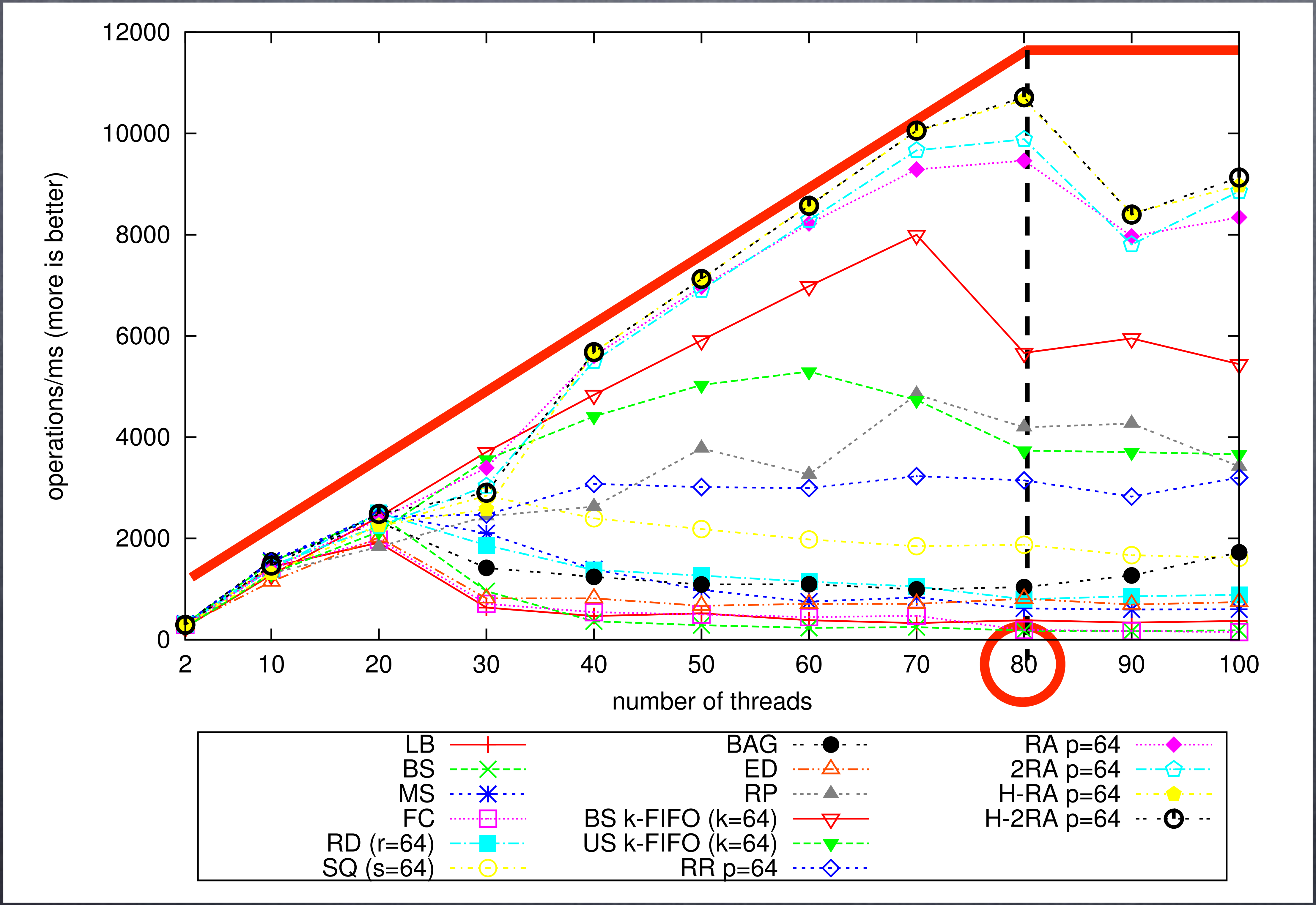
4 processors x 10 cores x 2 hardware threads = 80 hardware threads



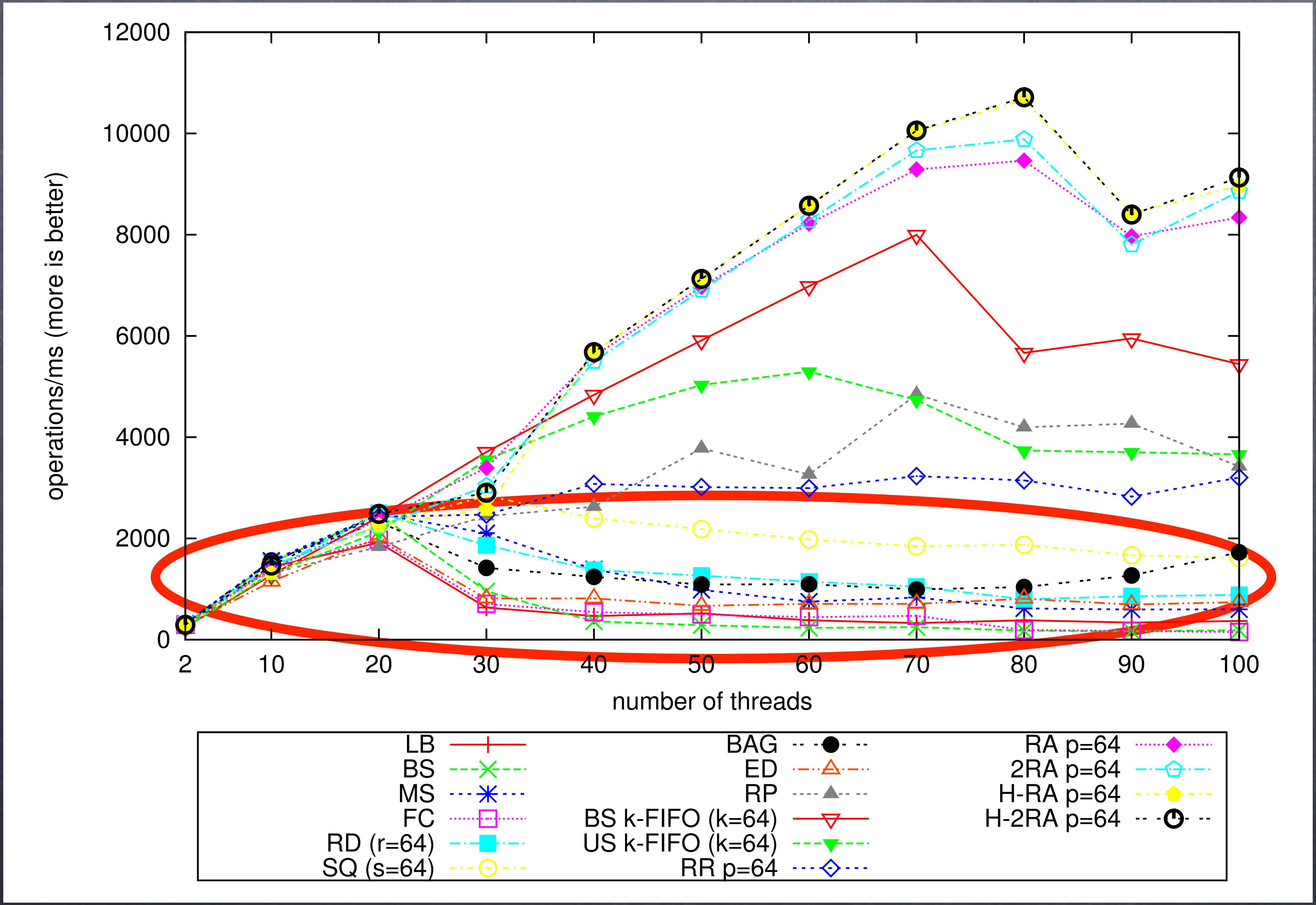
Performance & Scalability



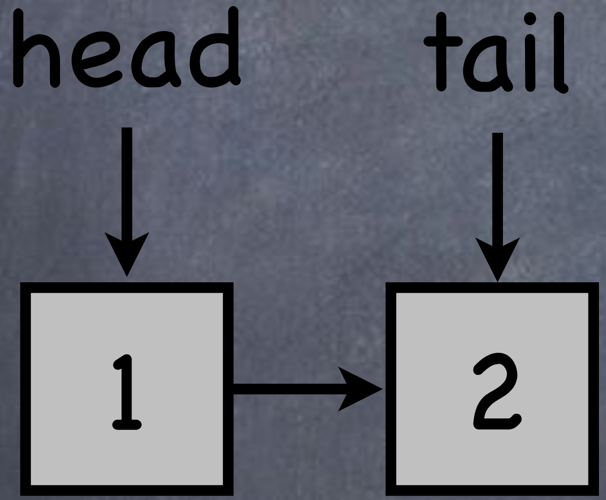
Ideal 80-Thread Performance



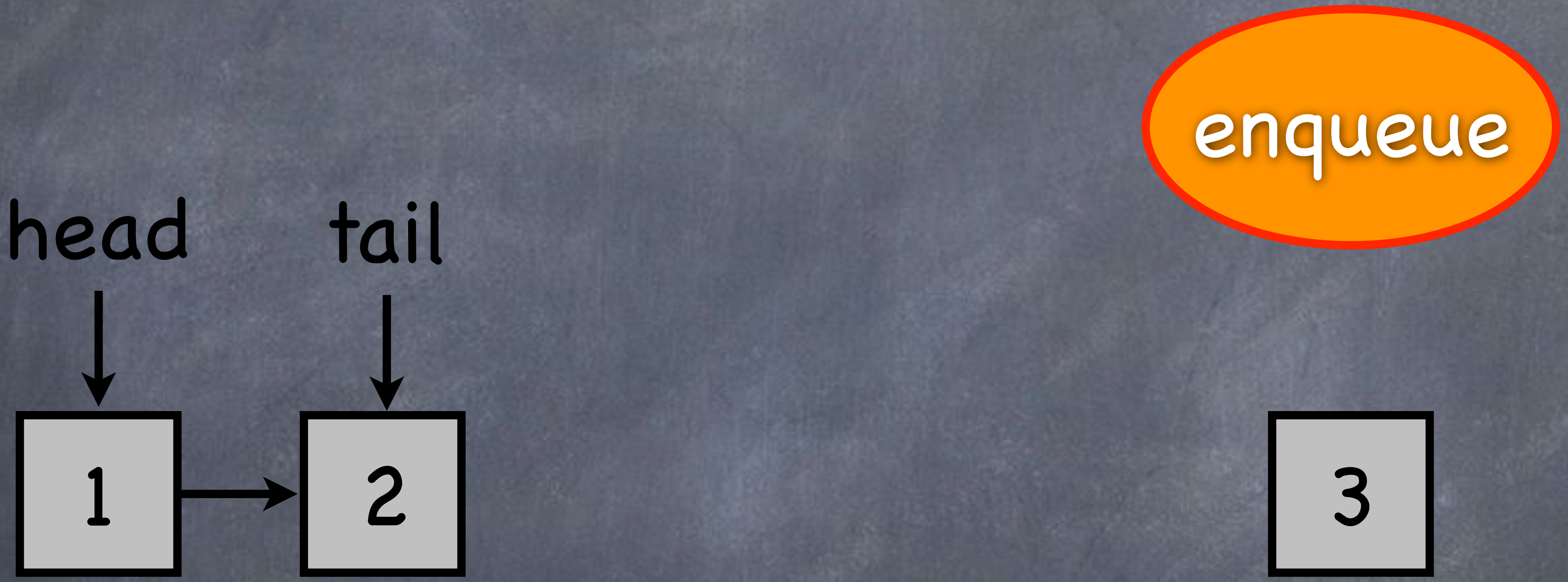
Regular FIFO Queues



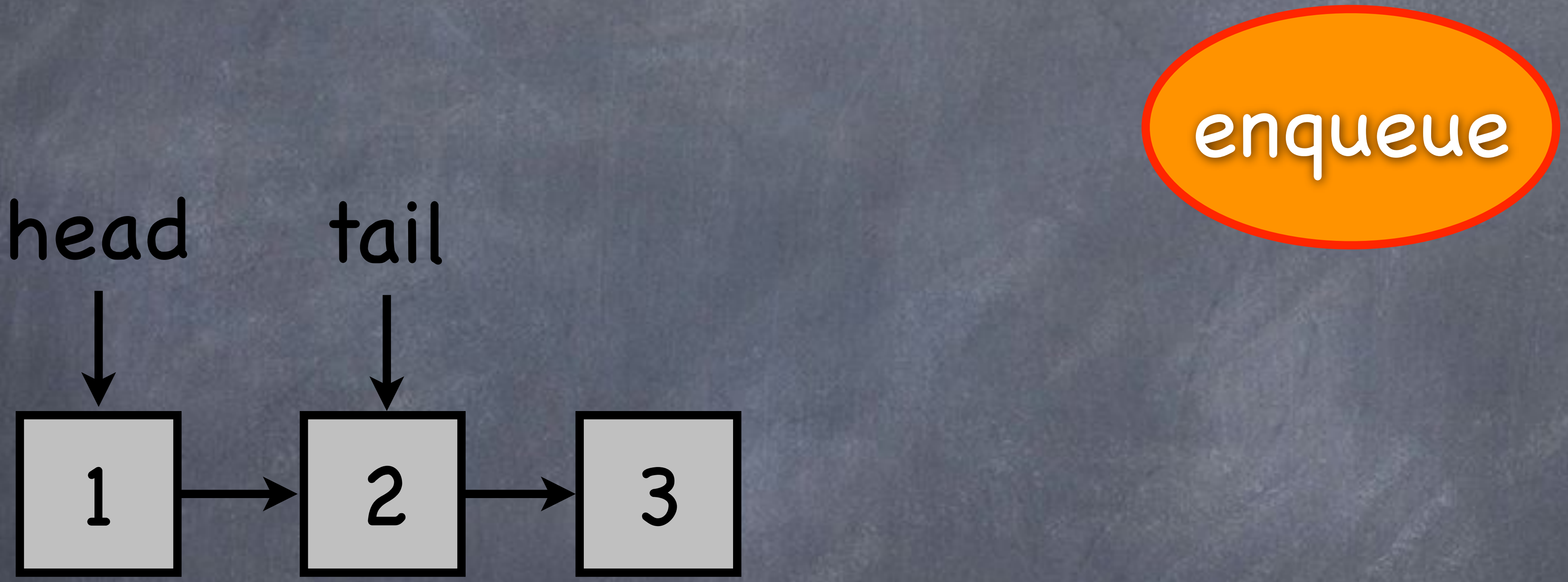
Concurrent First-in-First-out (FIFO) Queue



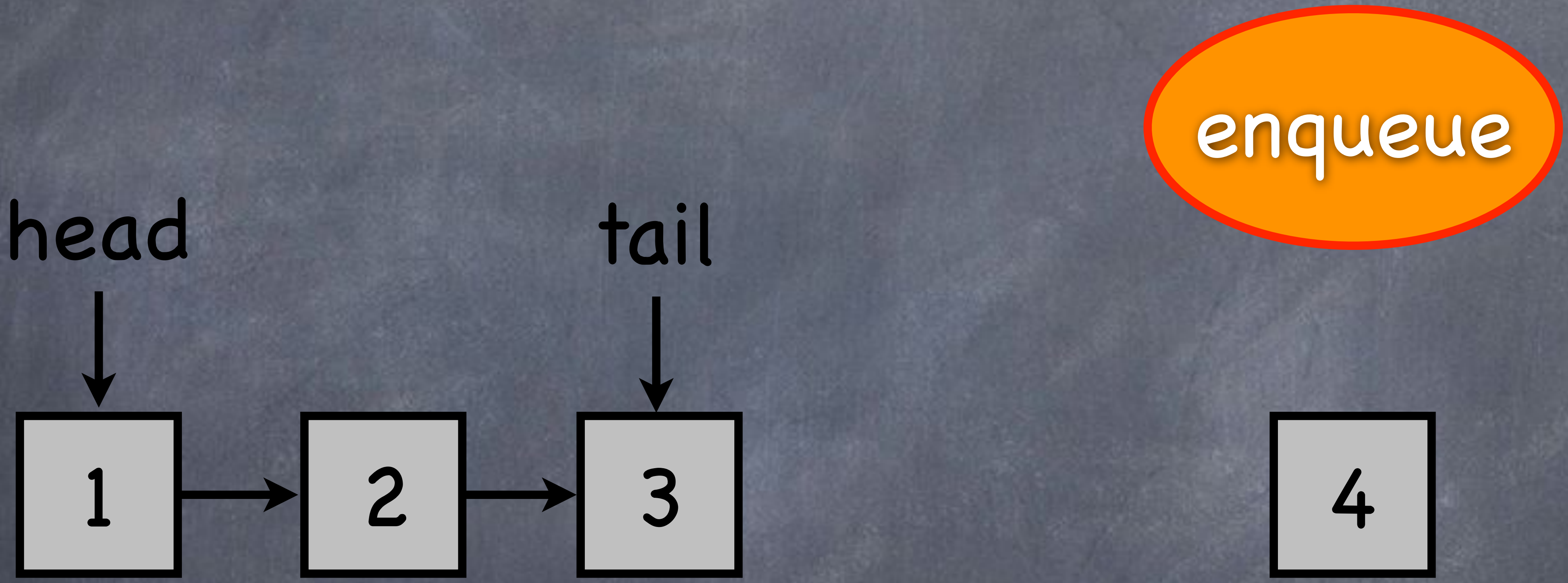
Concurrent First-in-First-out (FIFO) Queue



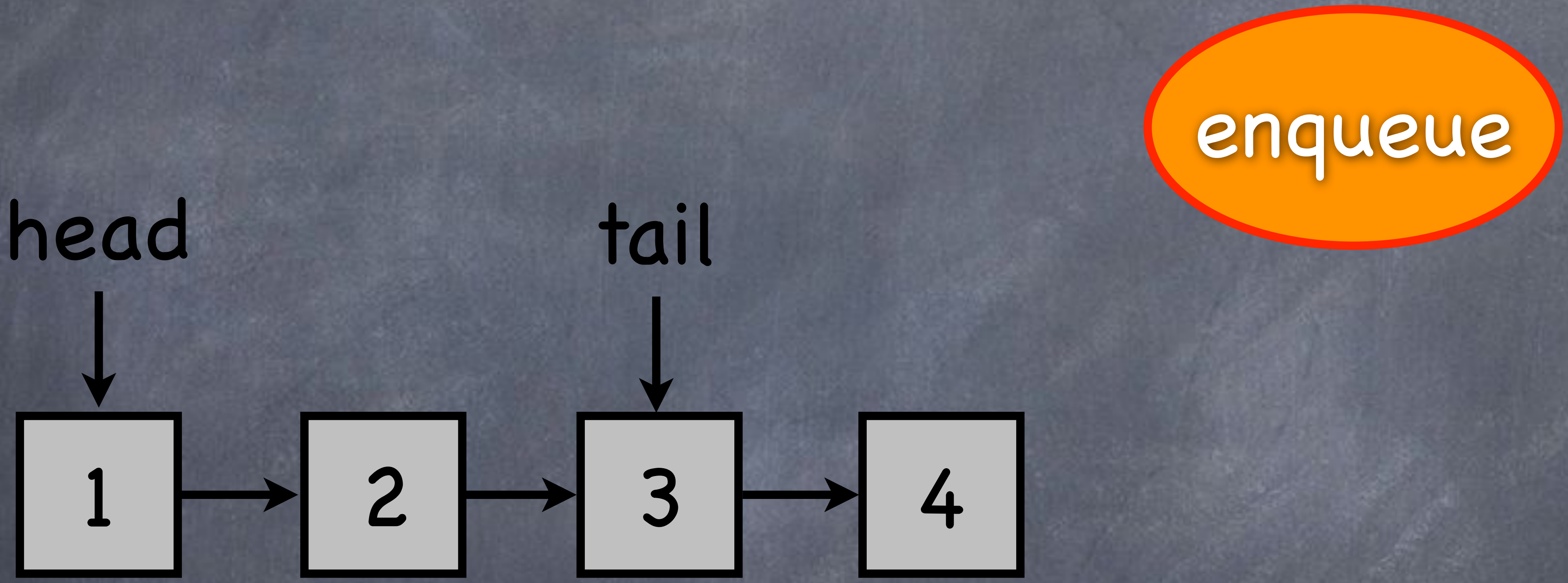
Concurrent First-in-First-out (FIFO) Queue



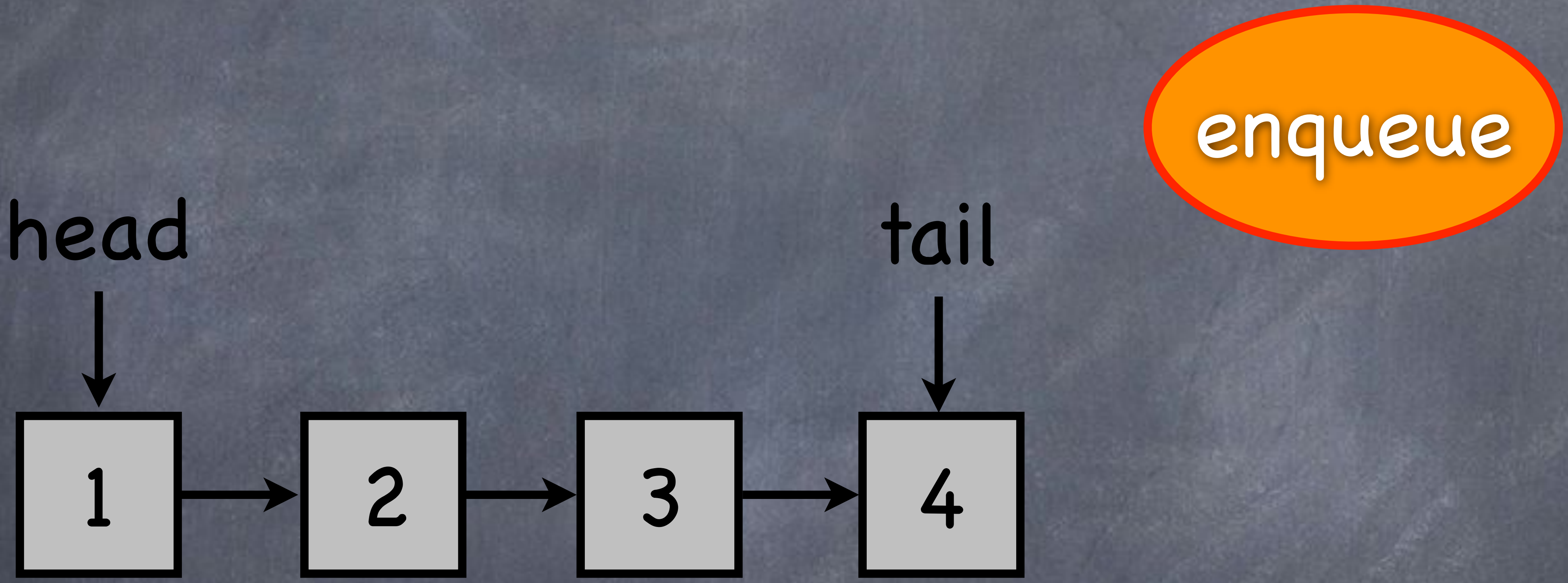
Concurrent First-in-First-out (FIFO) Queue



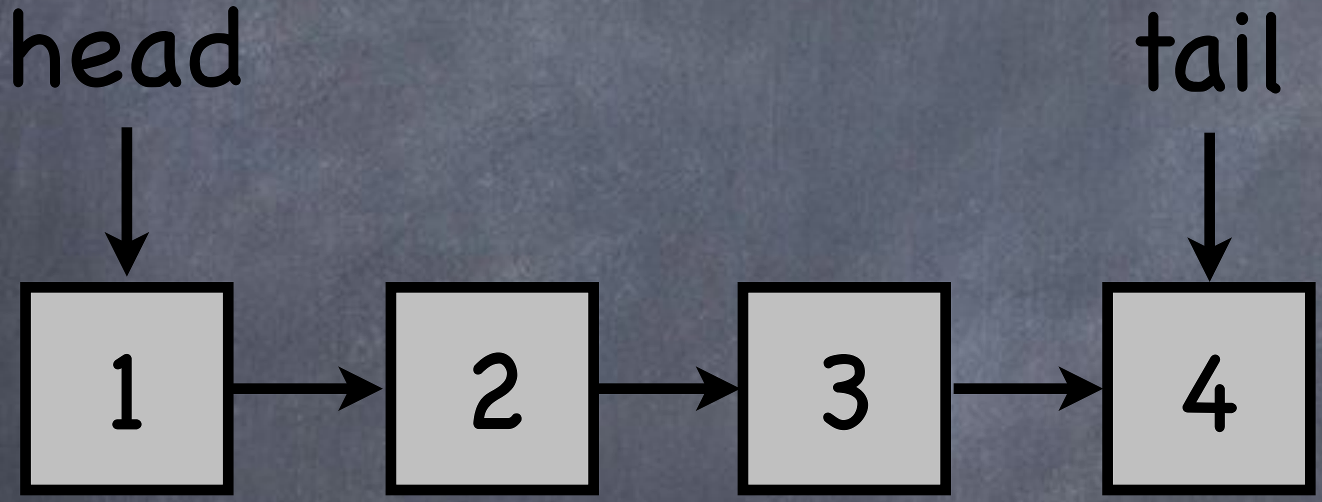
Concurrent First-in-First-out (FIFO) Queue



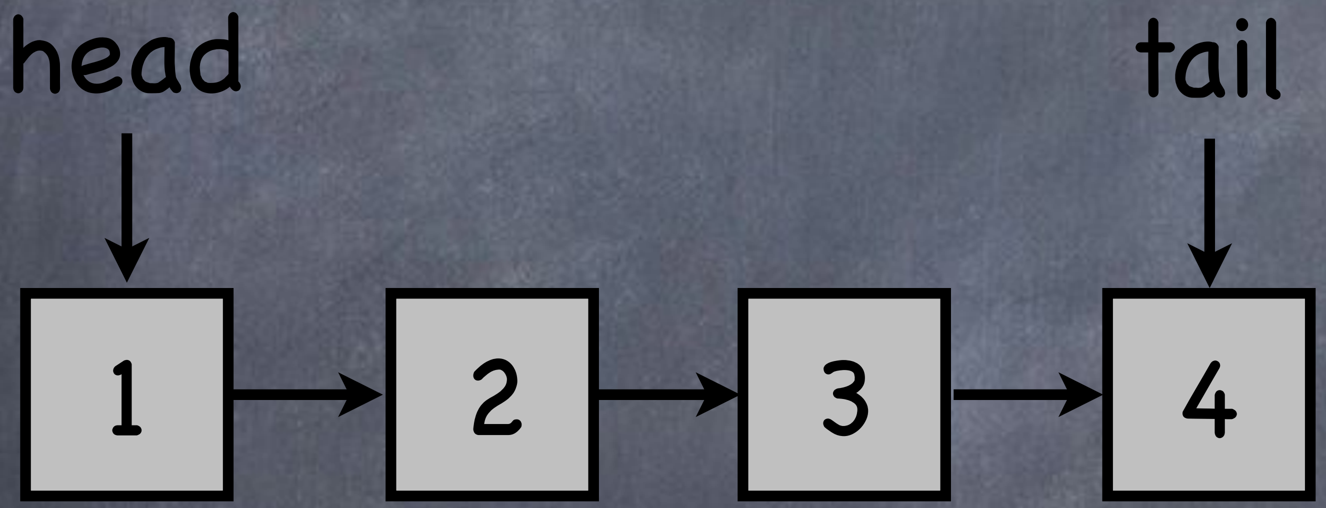
Concurrent First-in-First-out (FIFO) Queue



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

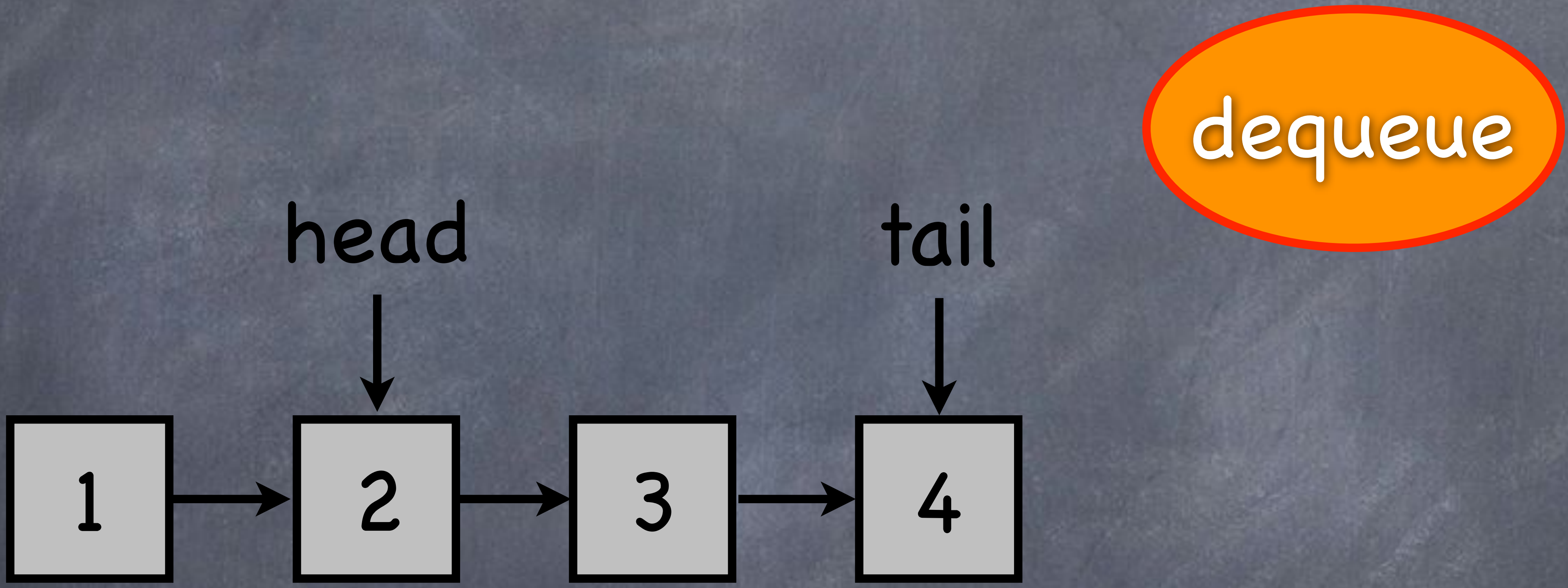


Concurrent First-in-First-out (FIFO) Queue

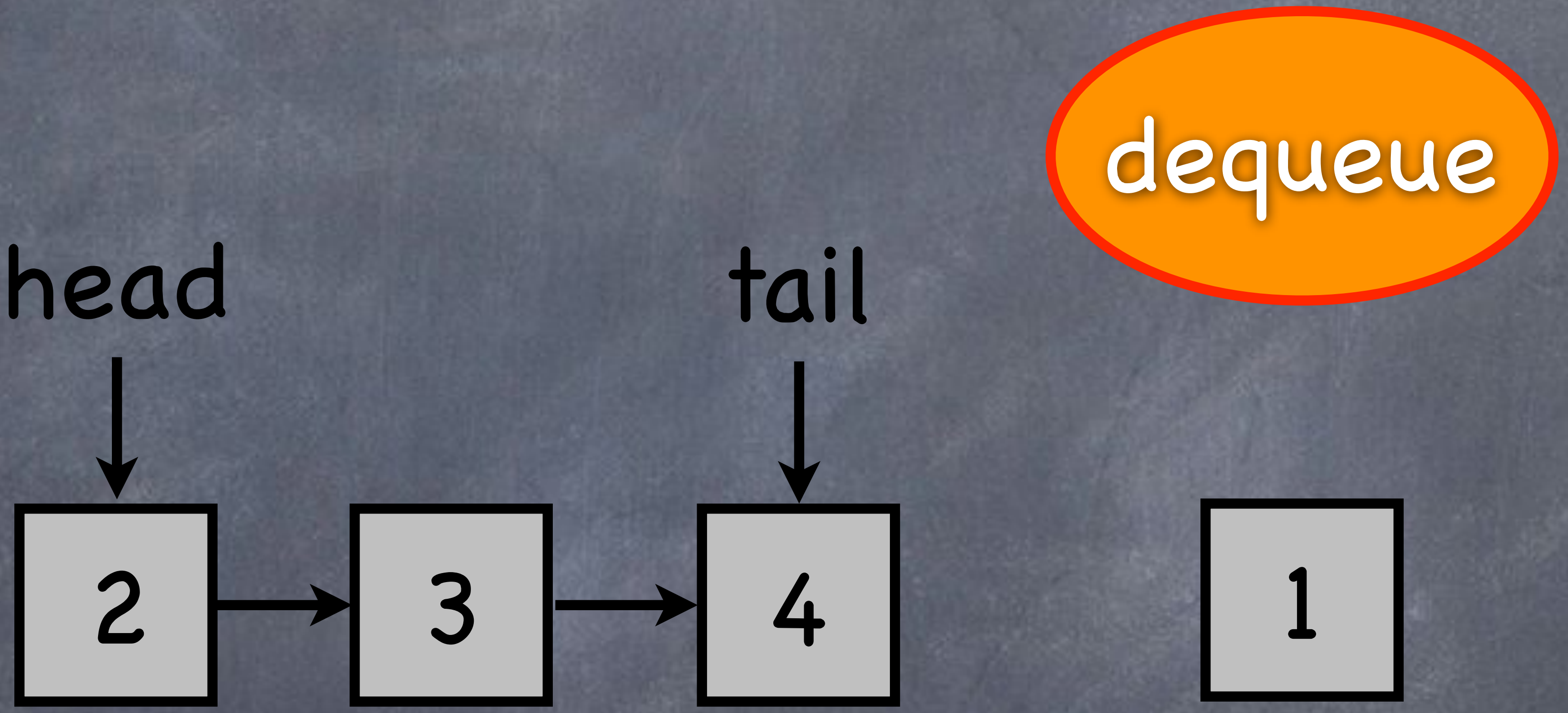


dequeue

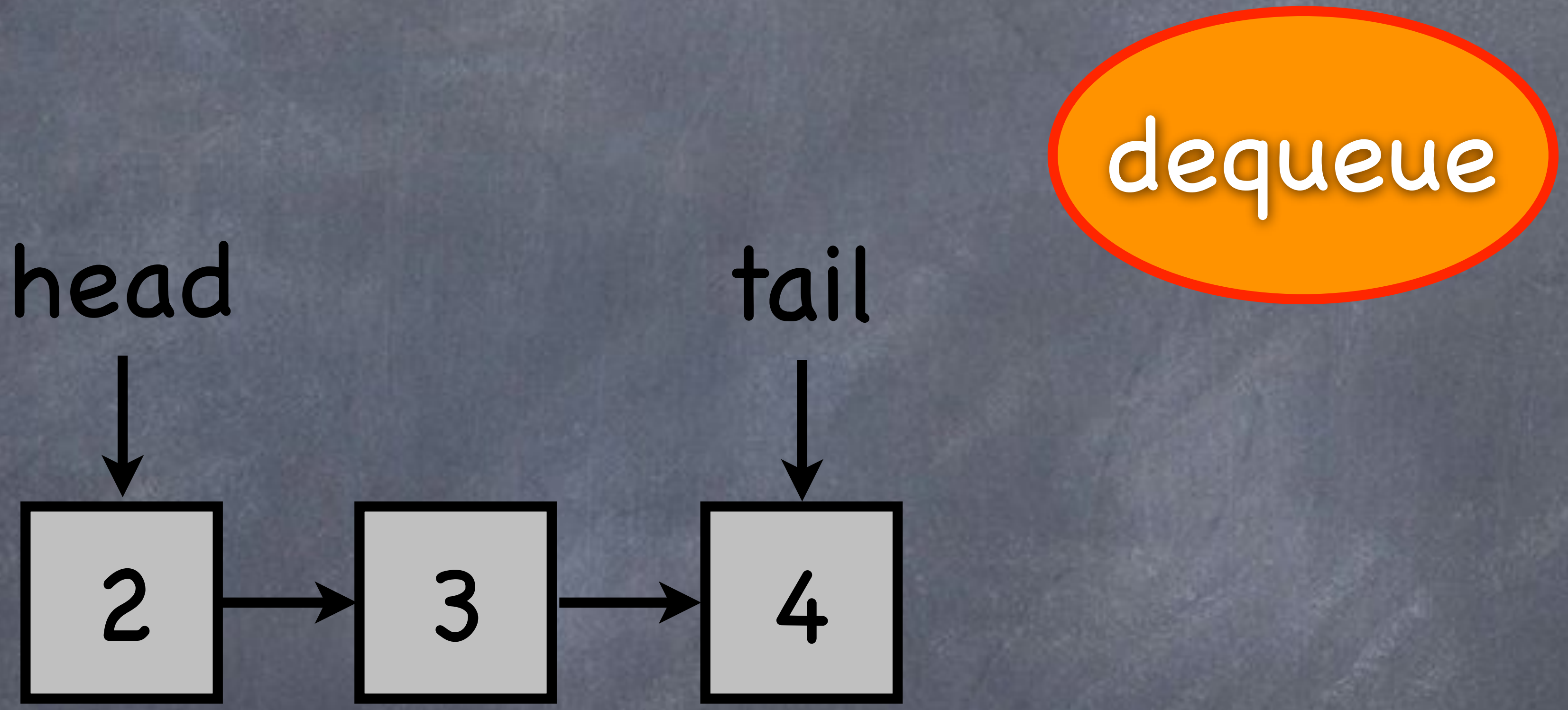
Concurrent First-in-First-out (FIFO) Queue



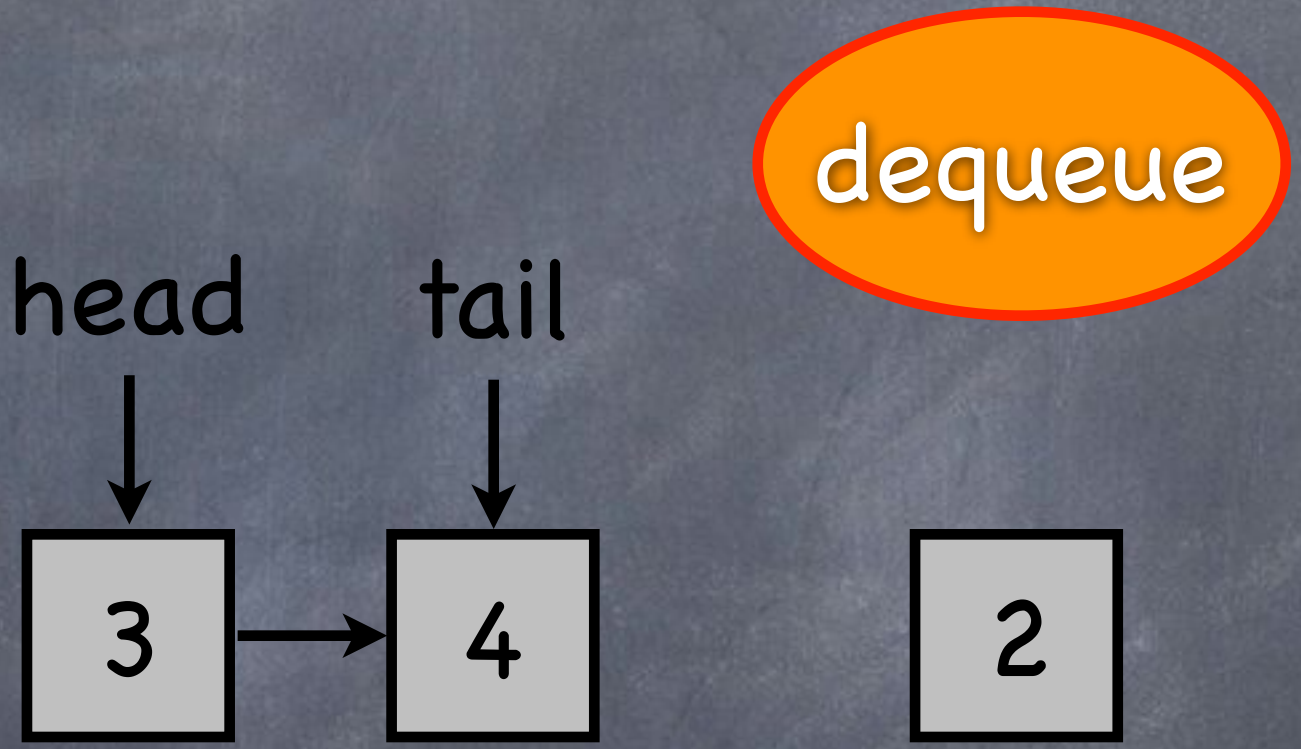
Concurrent First-in-First-out (FIFO) Queue



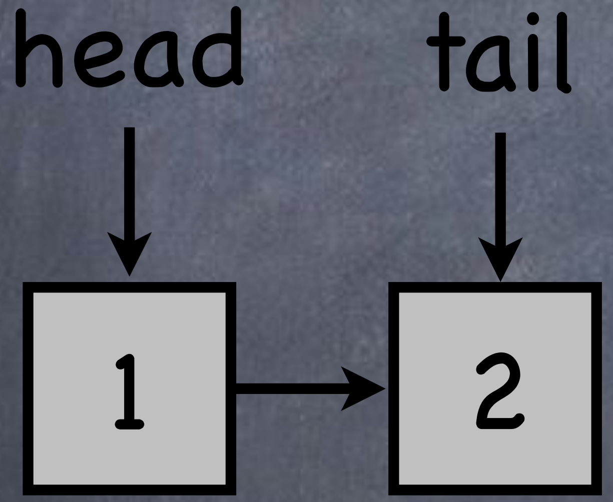
Concurrent First-in-First-out (FIFO) Queue



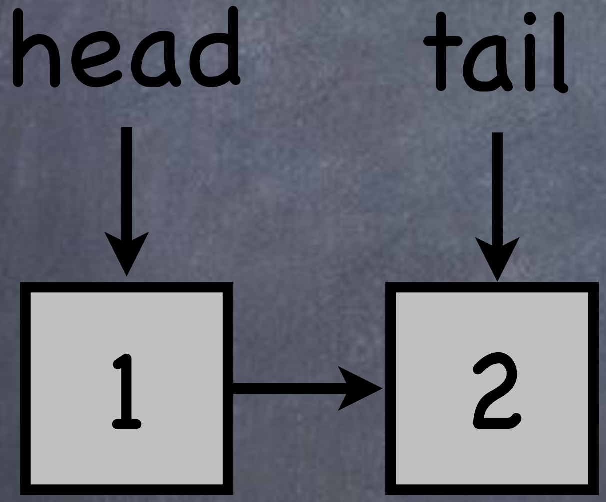
Concurrent First-in-First-out (FIFO) Queue



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

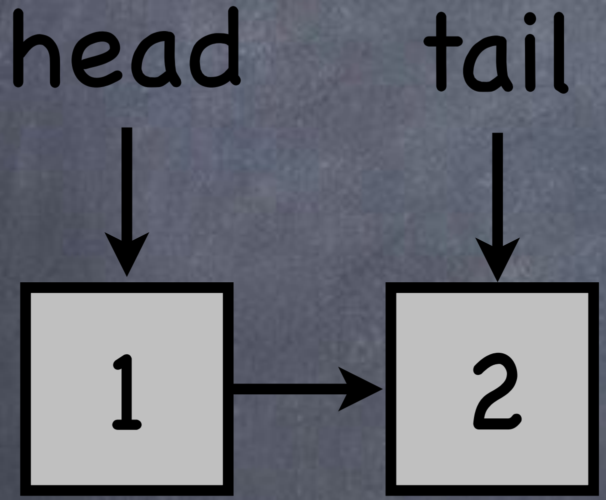


Concurrent First-in-First-out (FIFO) Queue



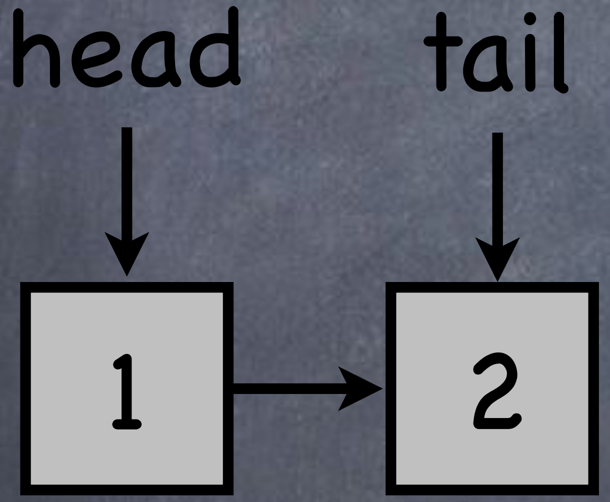
-> 1 lock

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



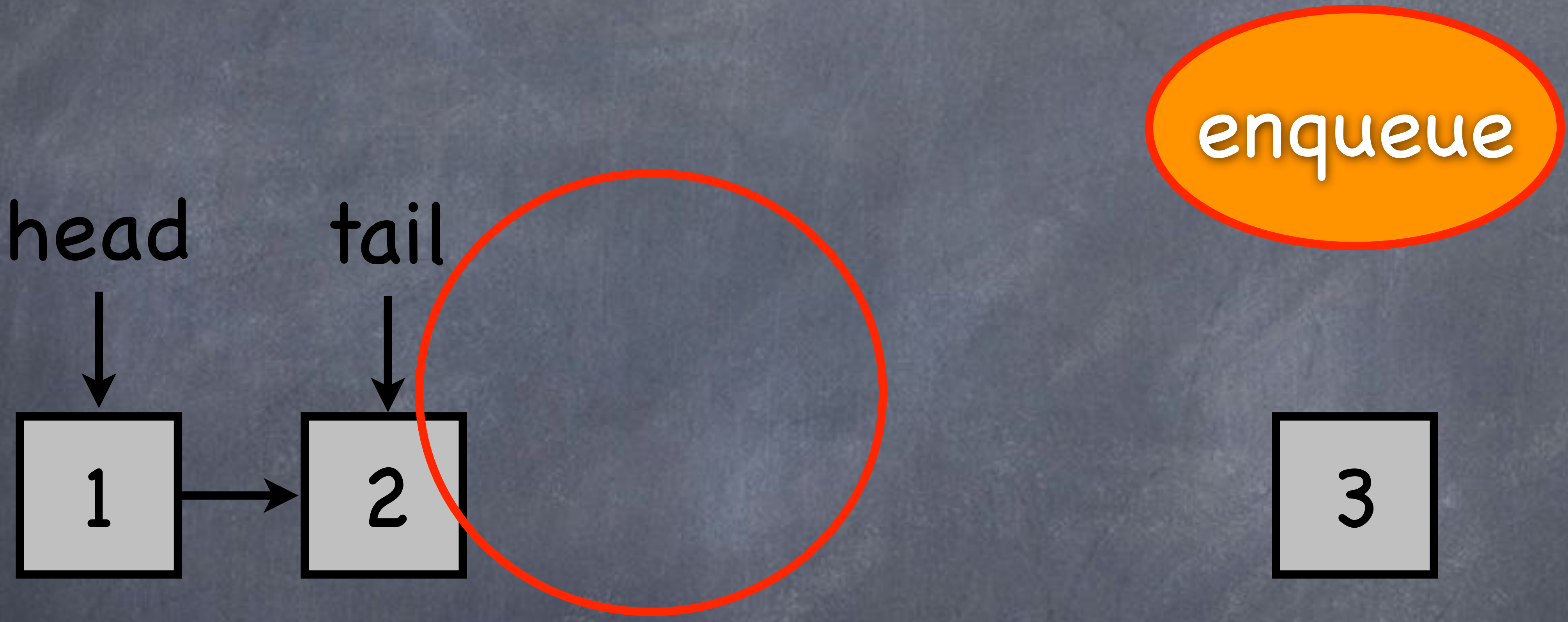
-> 1 lock -> 2 locks

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



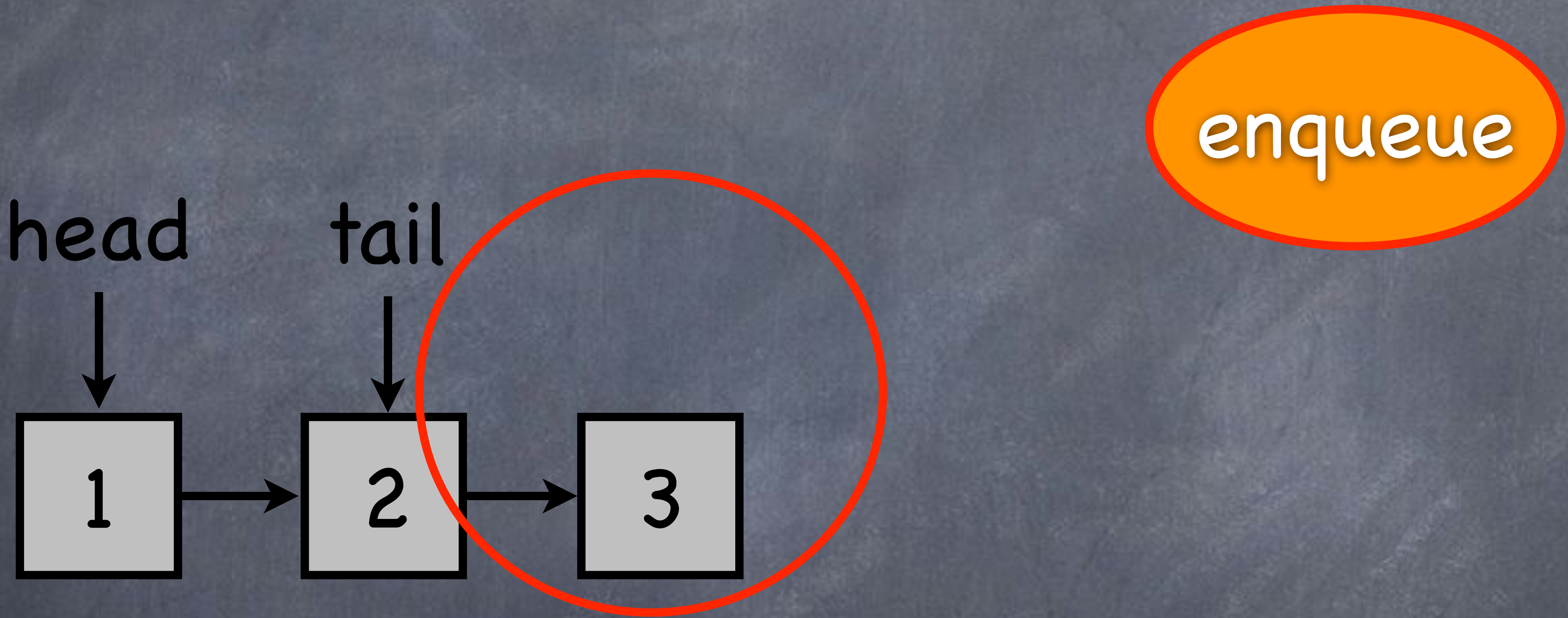
-> 1 lock -> 2 locks -> 0 locks

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



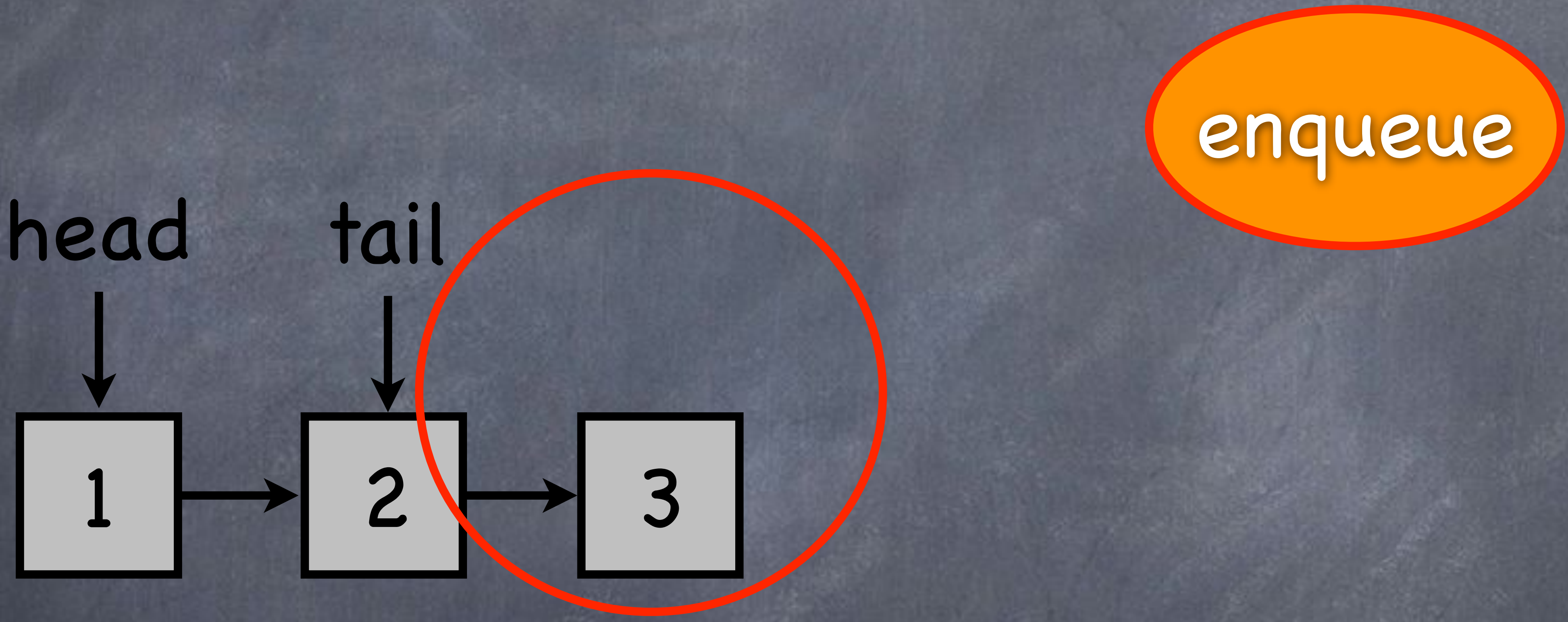
-> 1 lock -> 2 locks -> 0 locks

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



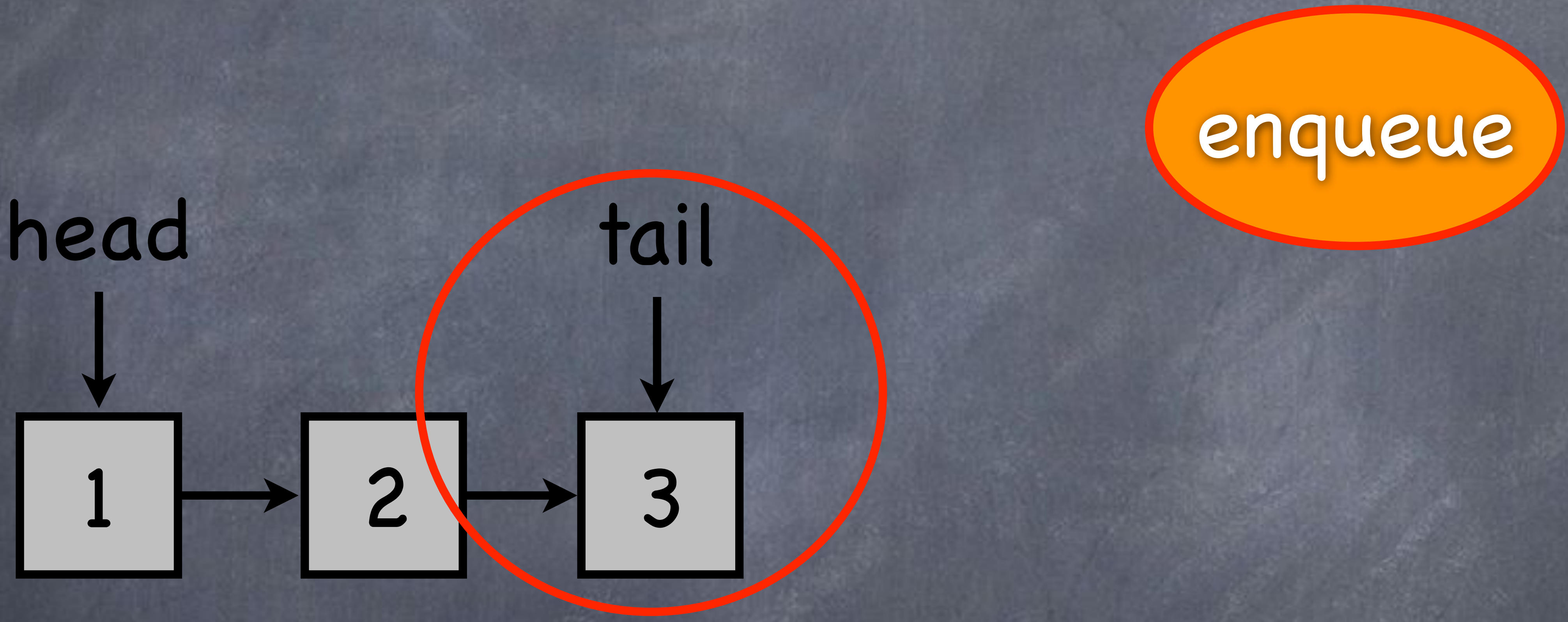
-> 1 lock -> 2 locks -> 0 locks

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



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

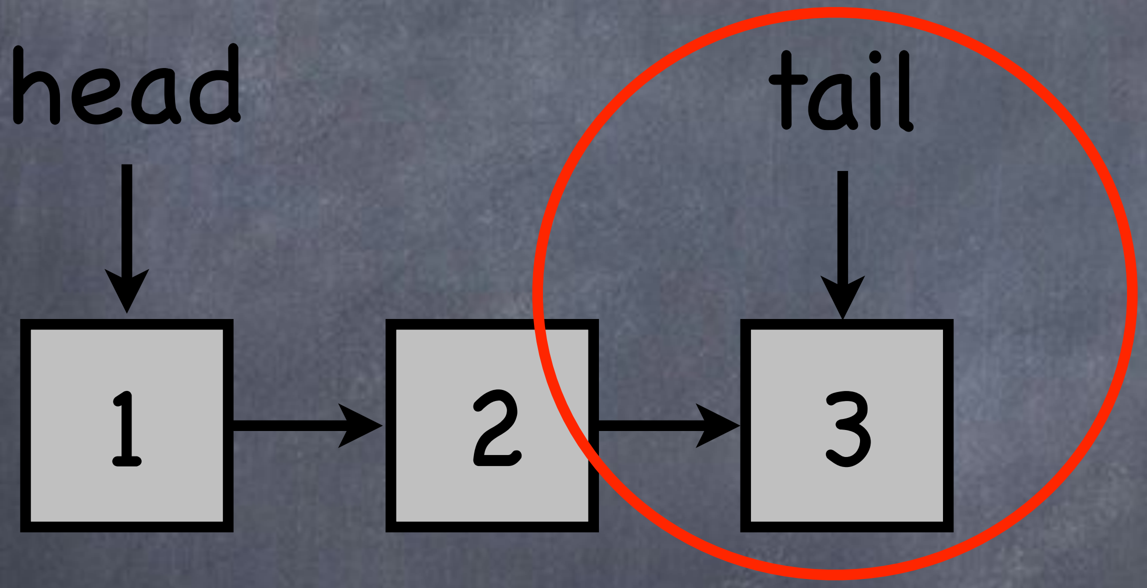
Concurrent First-in-First-out (FIFO) Queue



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

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

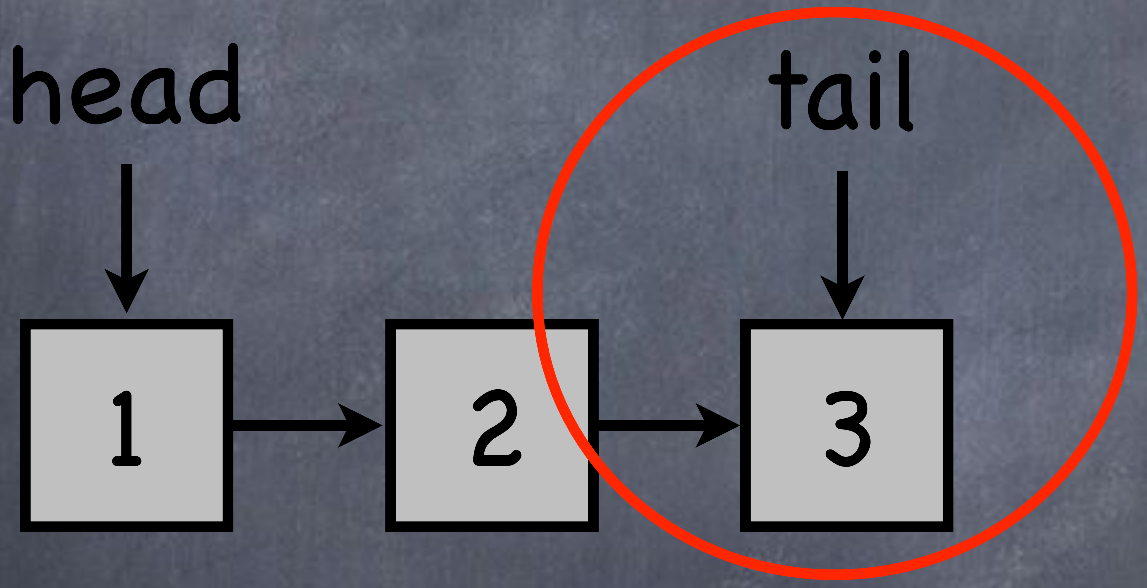
enqueue



[Michael,Scott'96]

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

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

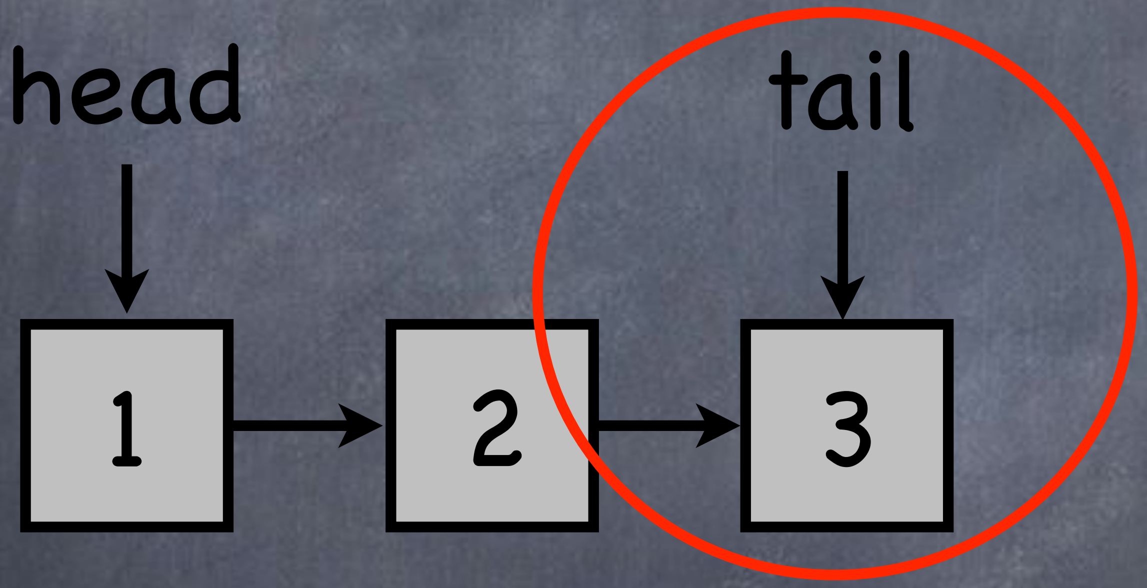


enqueue

[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

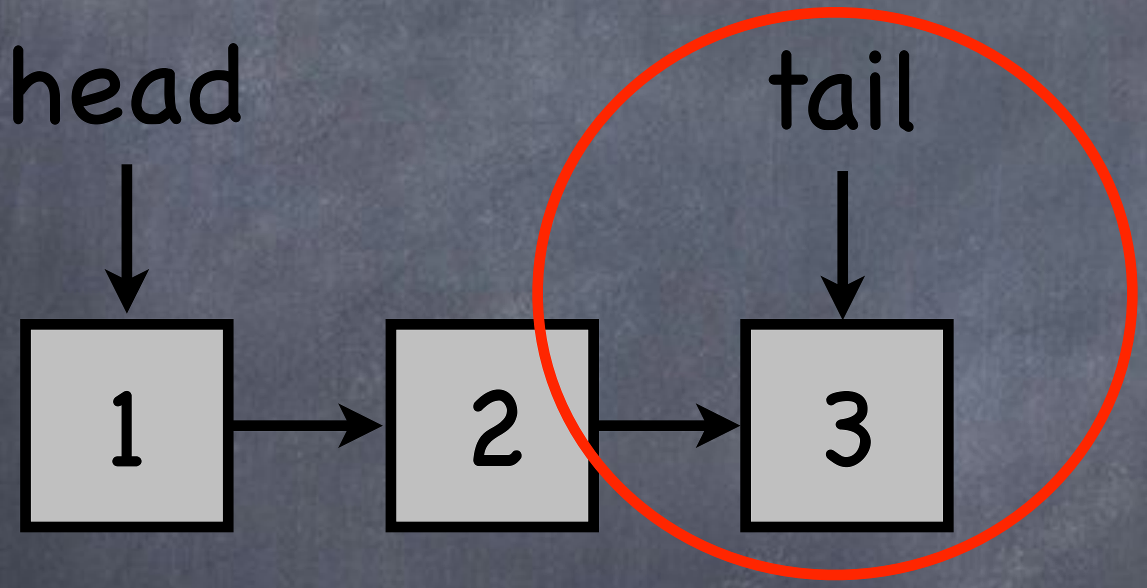


enqueue

[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

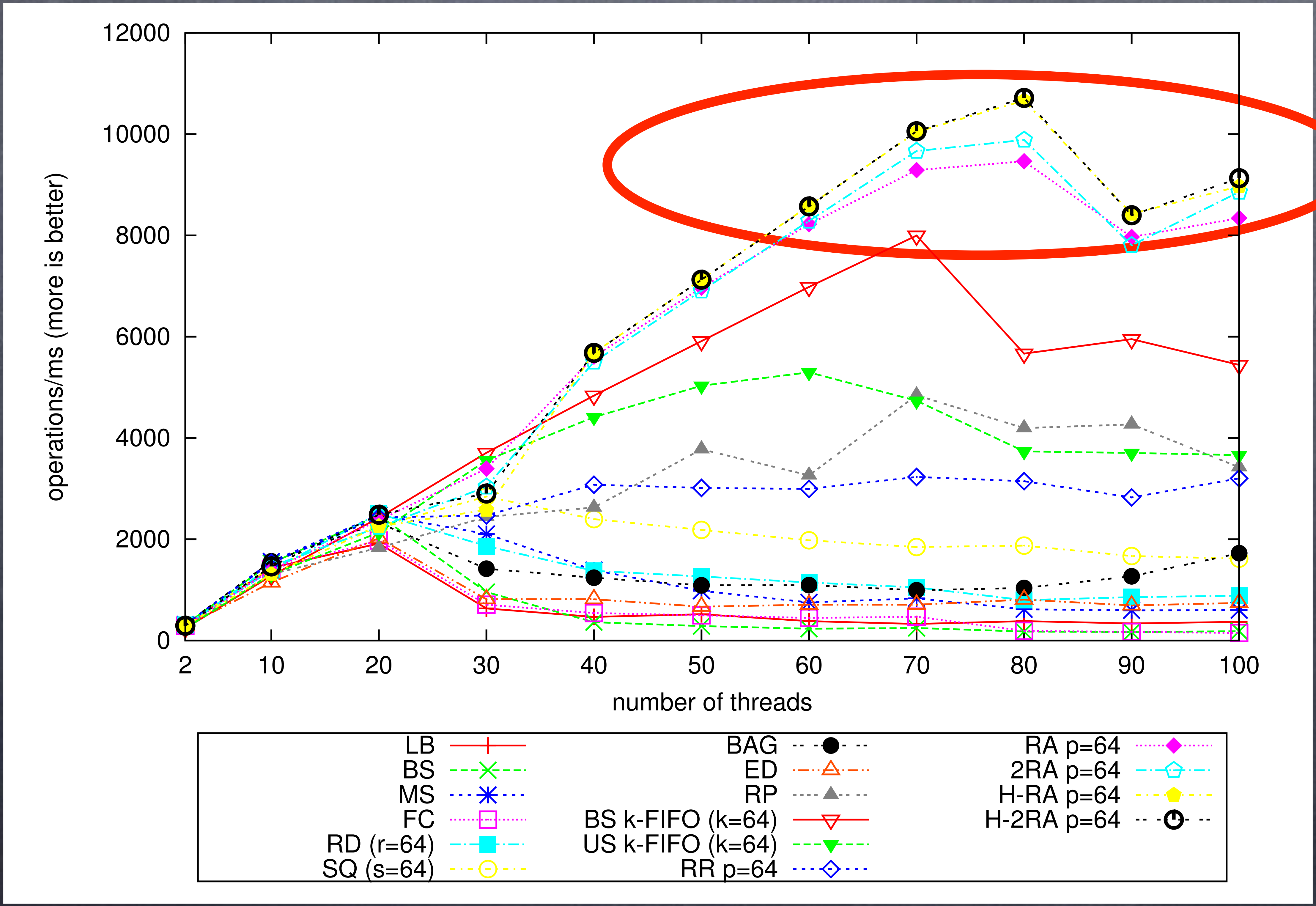


enqueue

[Michael,Scott'96]

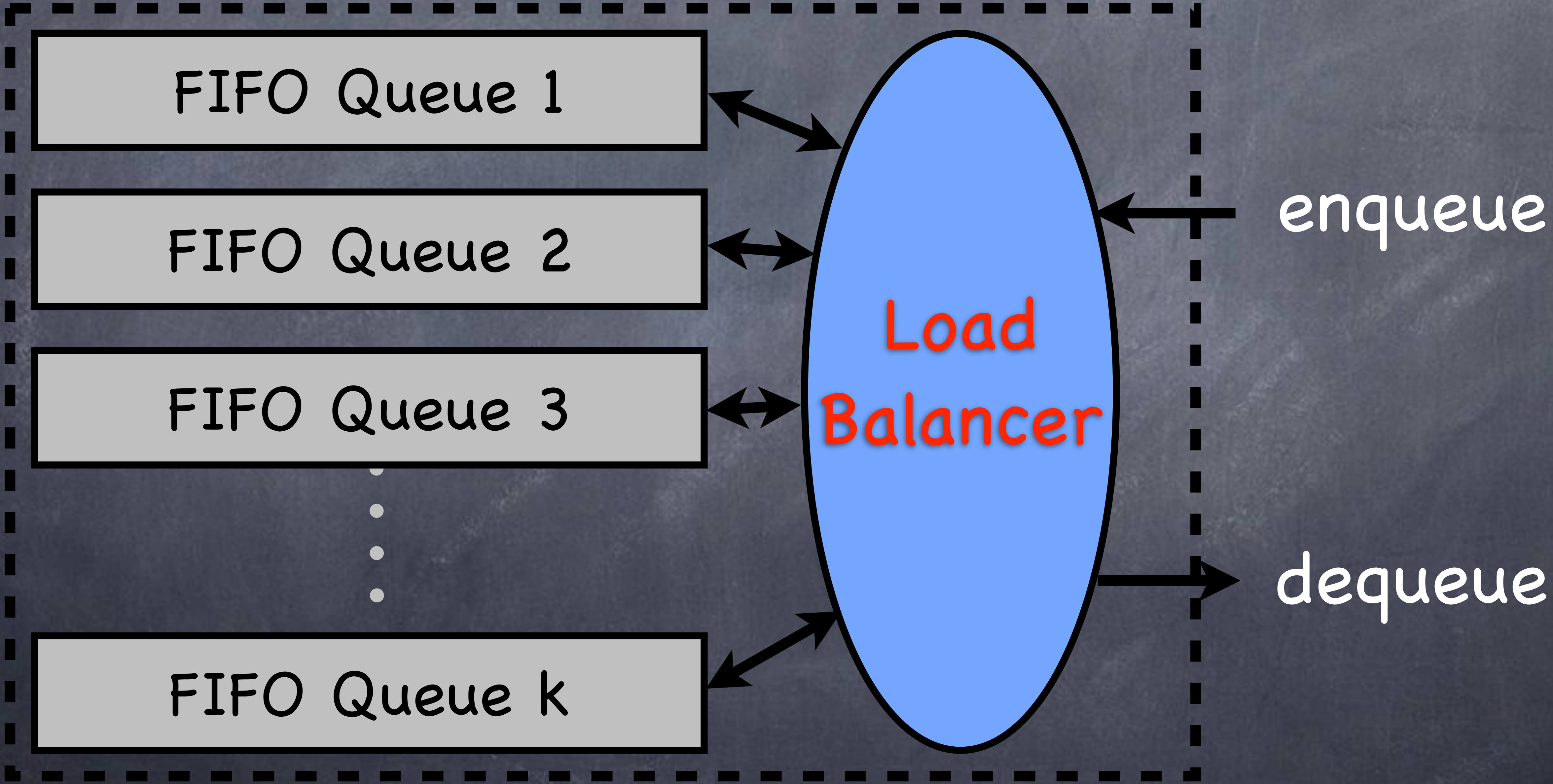
-> 1 lock -> 2 locks -> 0 locks -> compare & swap
-> lock-based vs. lock-free vs. wait-free?
-> memory contention on head and tail pointers!
-> and on next pointers!

Distributed Queues

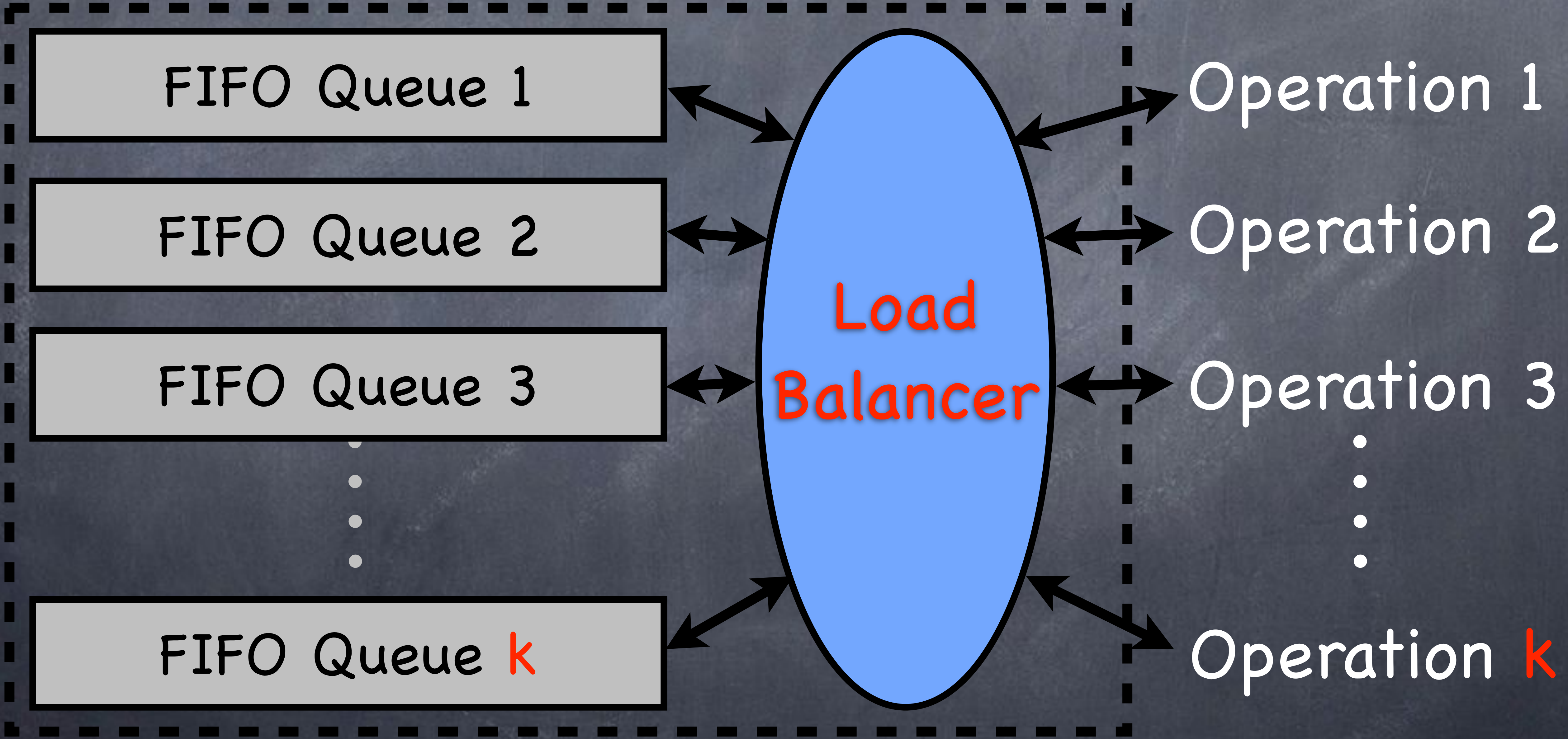


Distributed Queues

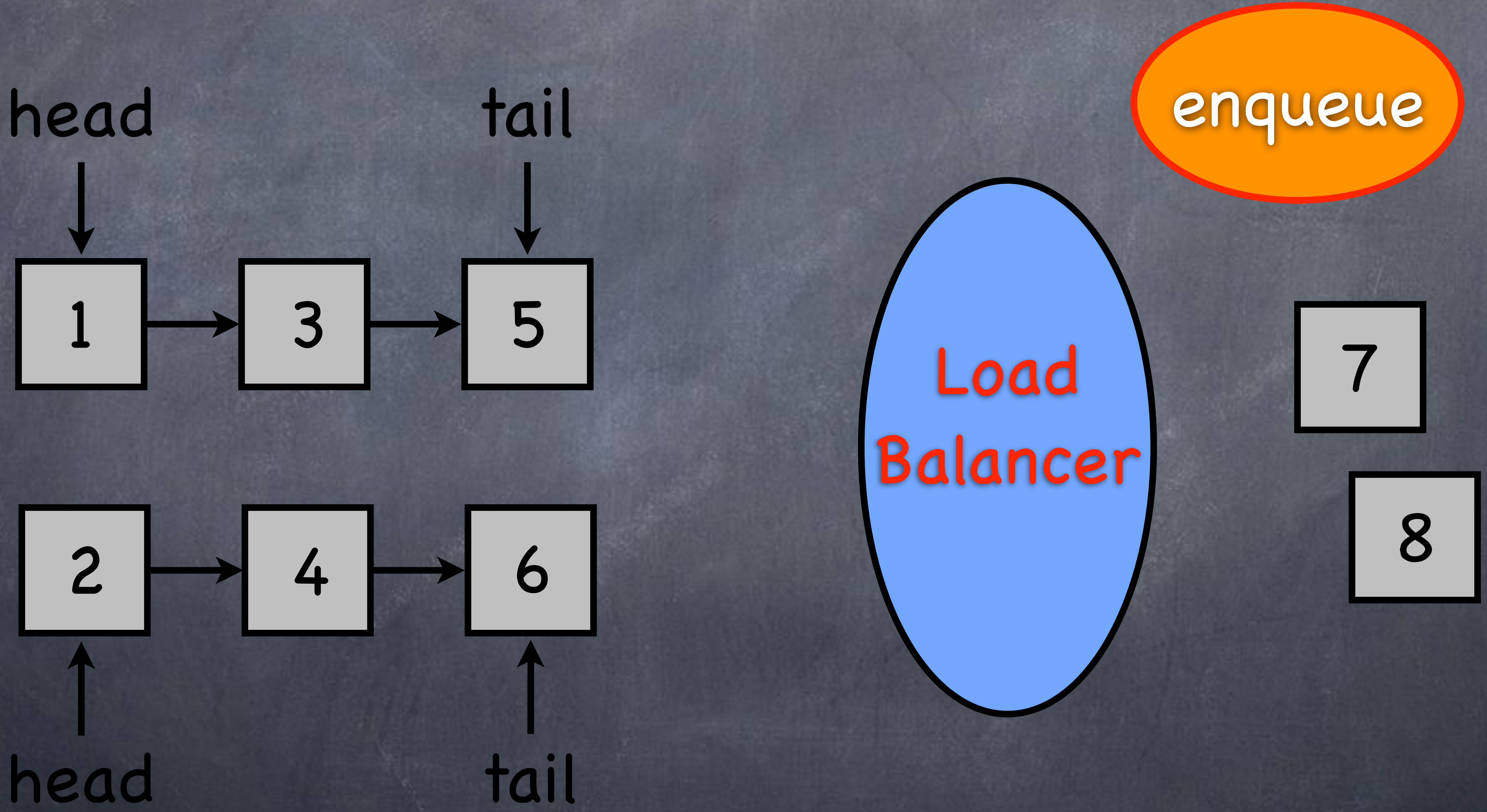
[_,Payer,Röck,Sokolova'12]



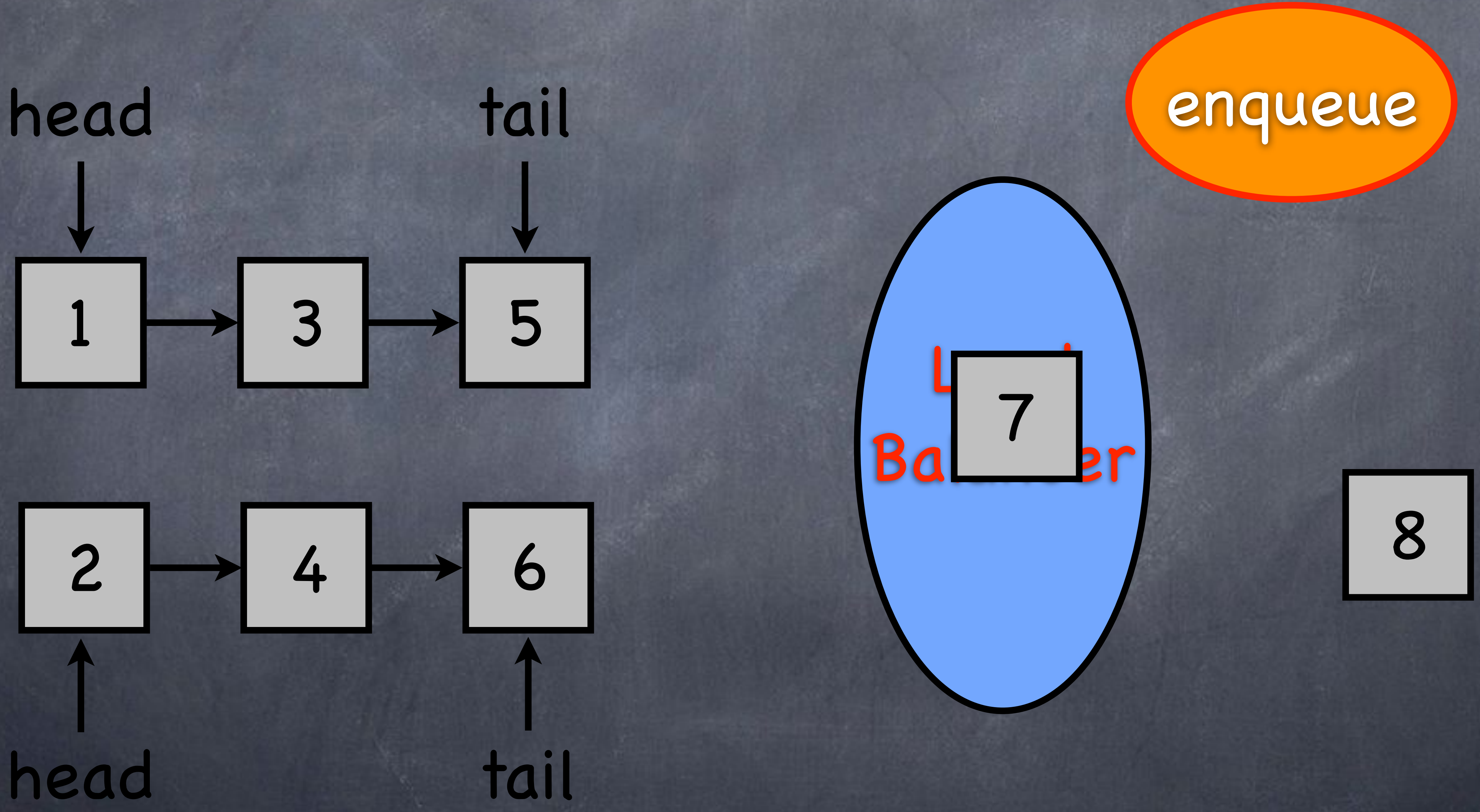
Up to k Parallel Enqueues and k Parallel Dequeues



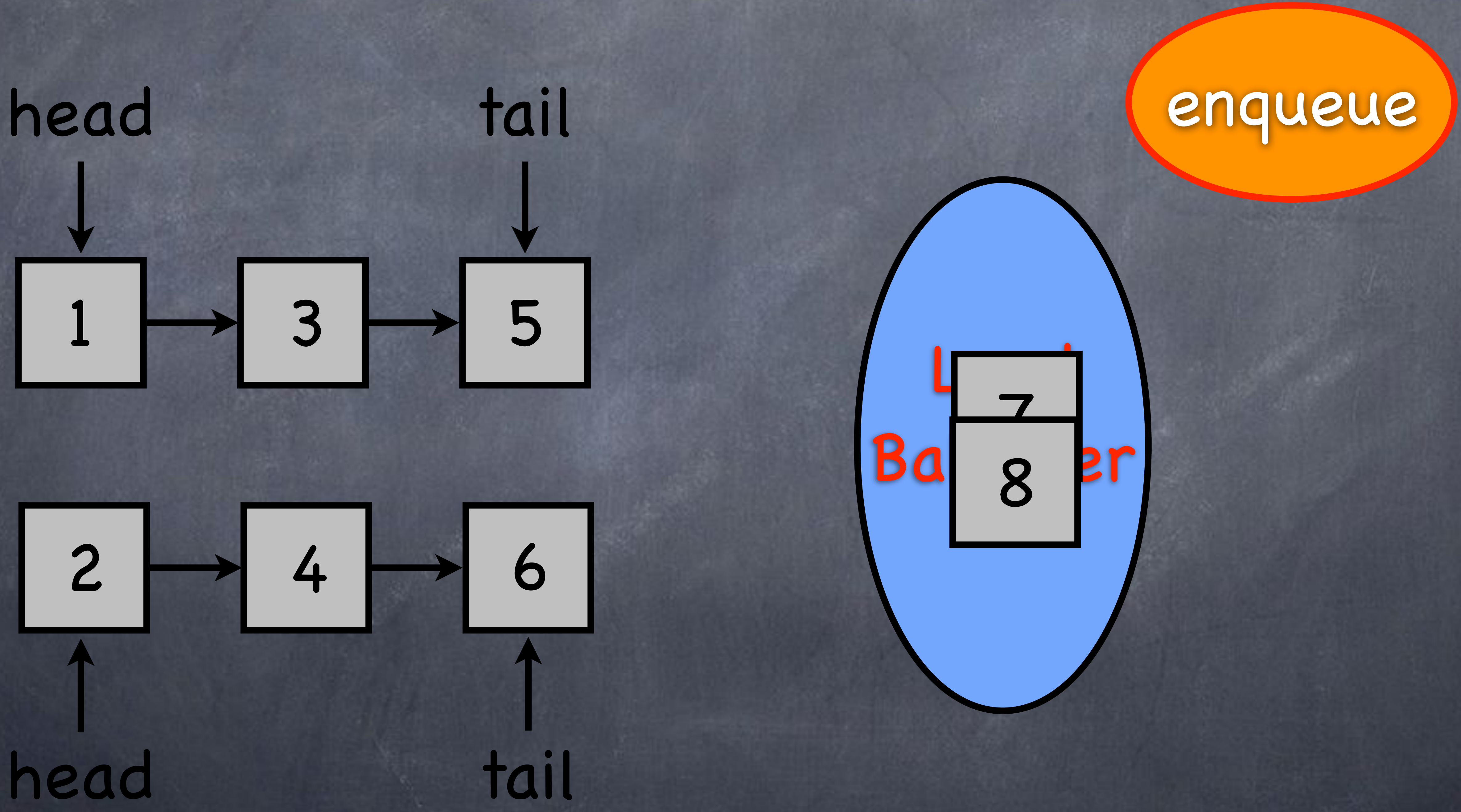
Load Balancing



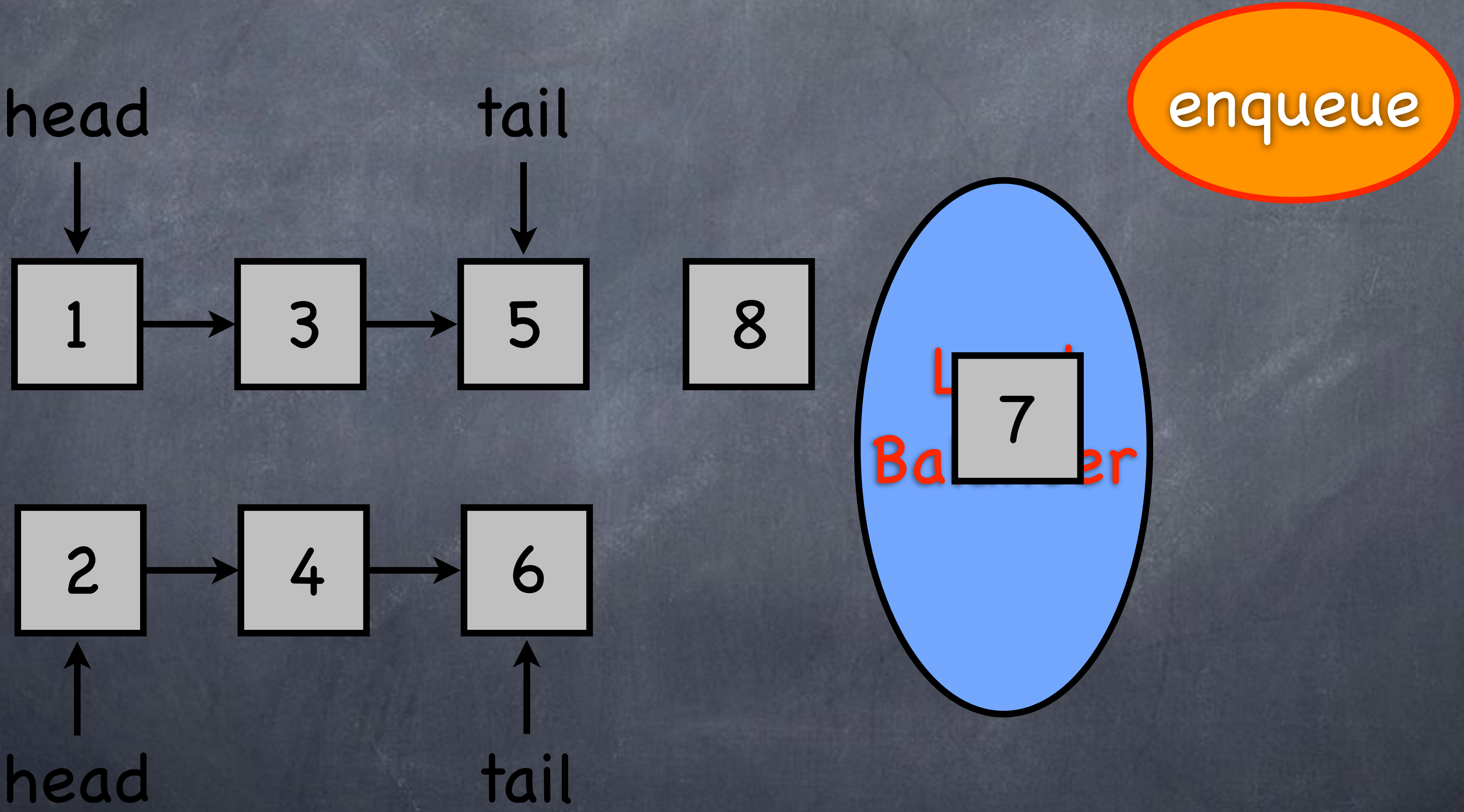
Load Balancing



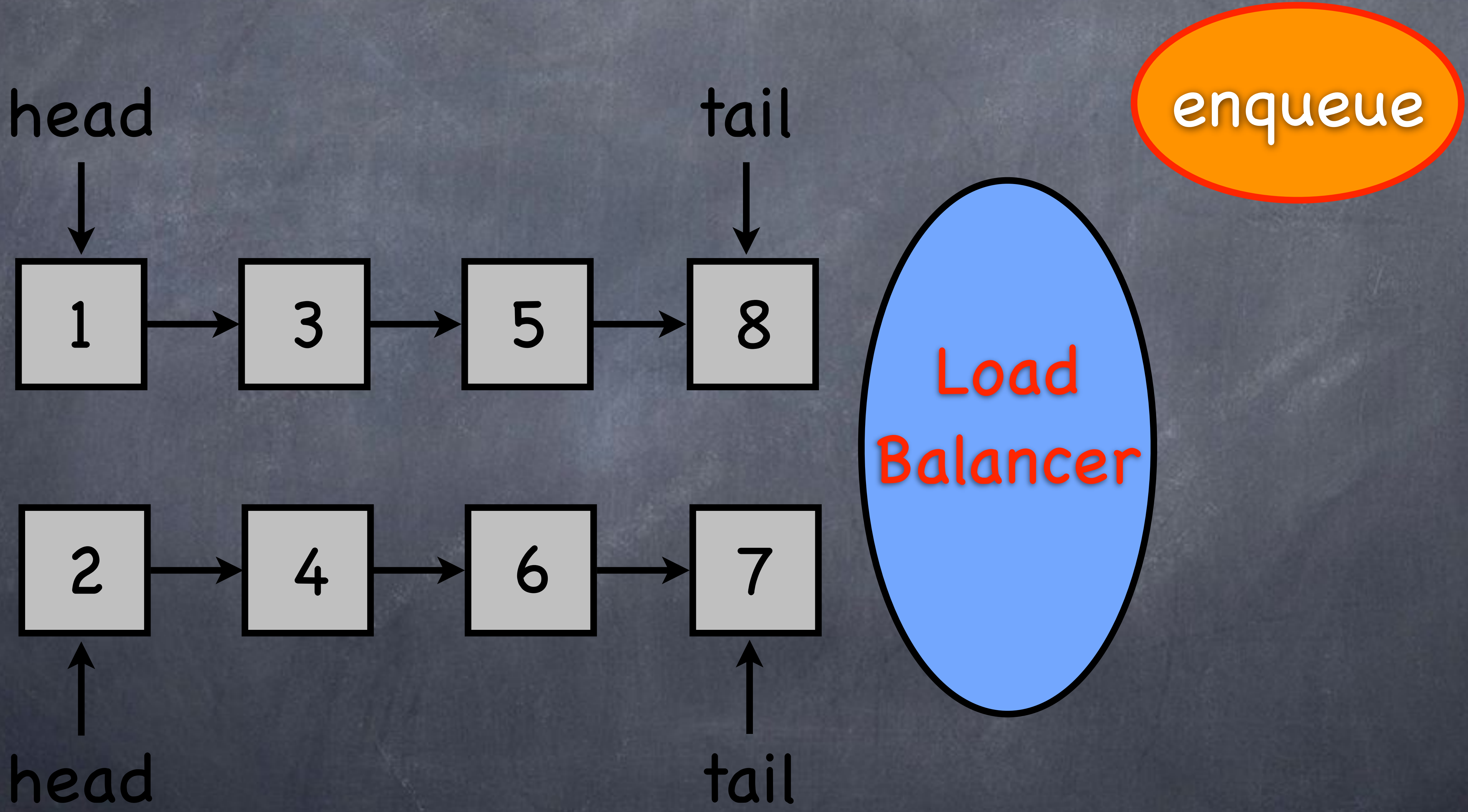
Load Balancing



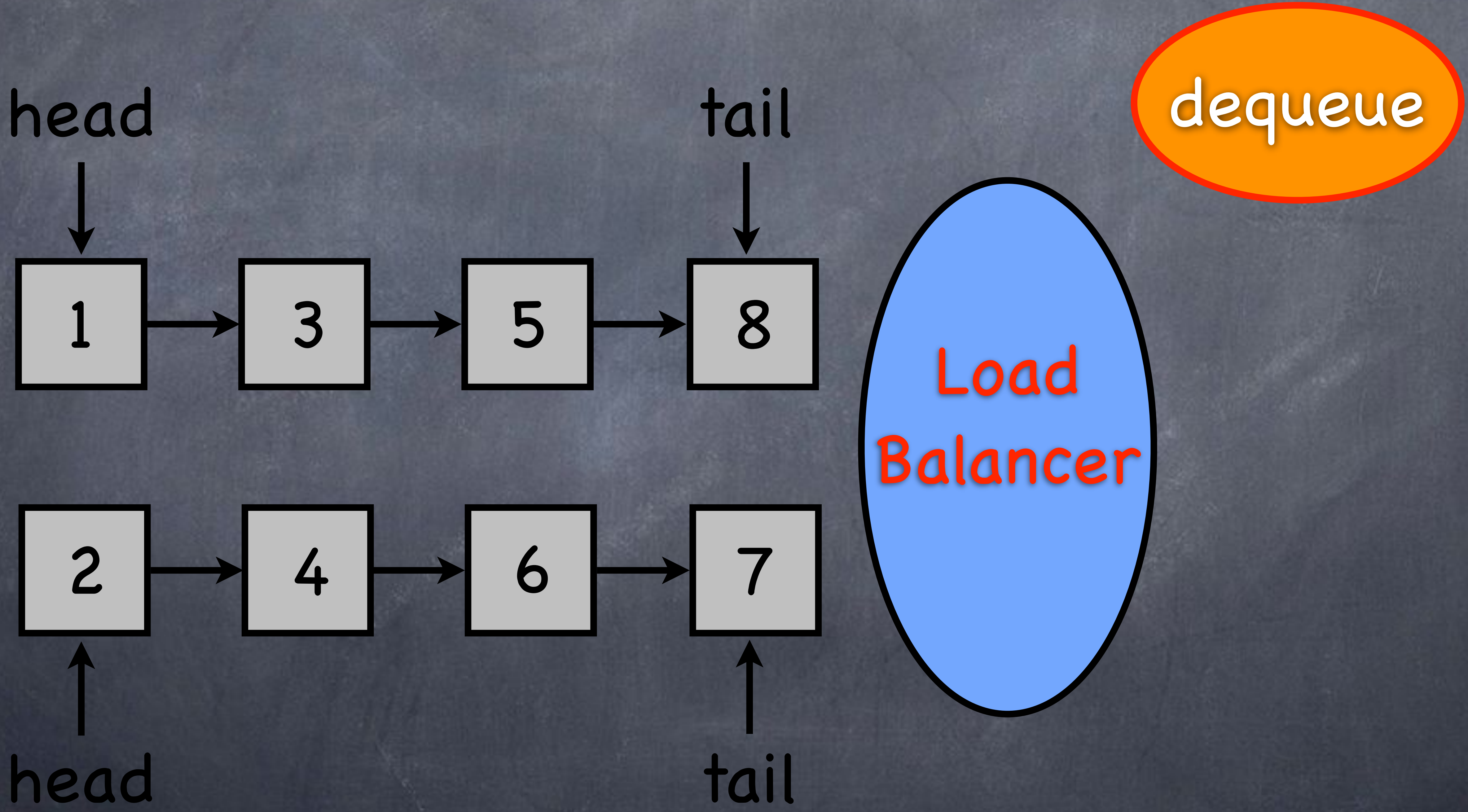
Load Balancing



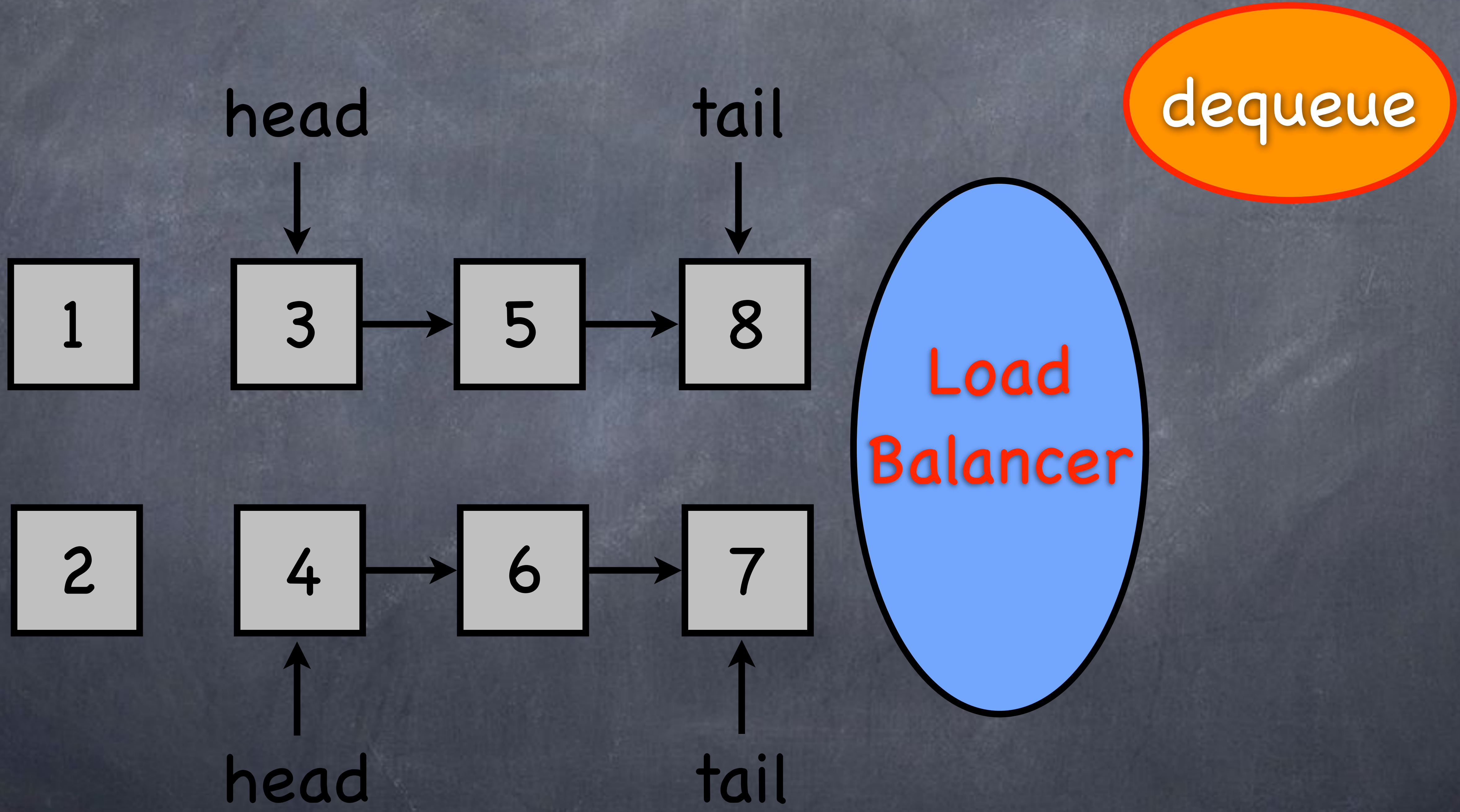
Load Balancing



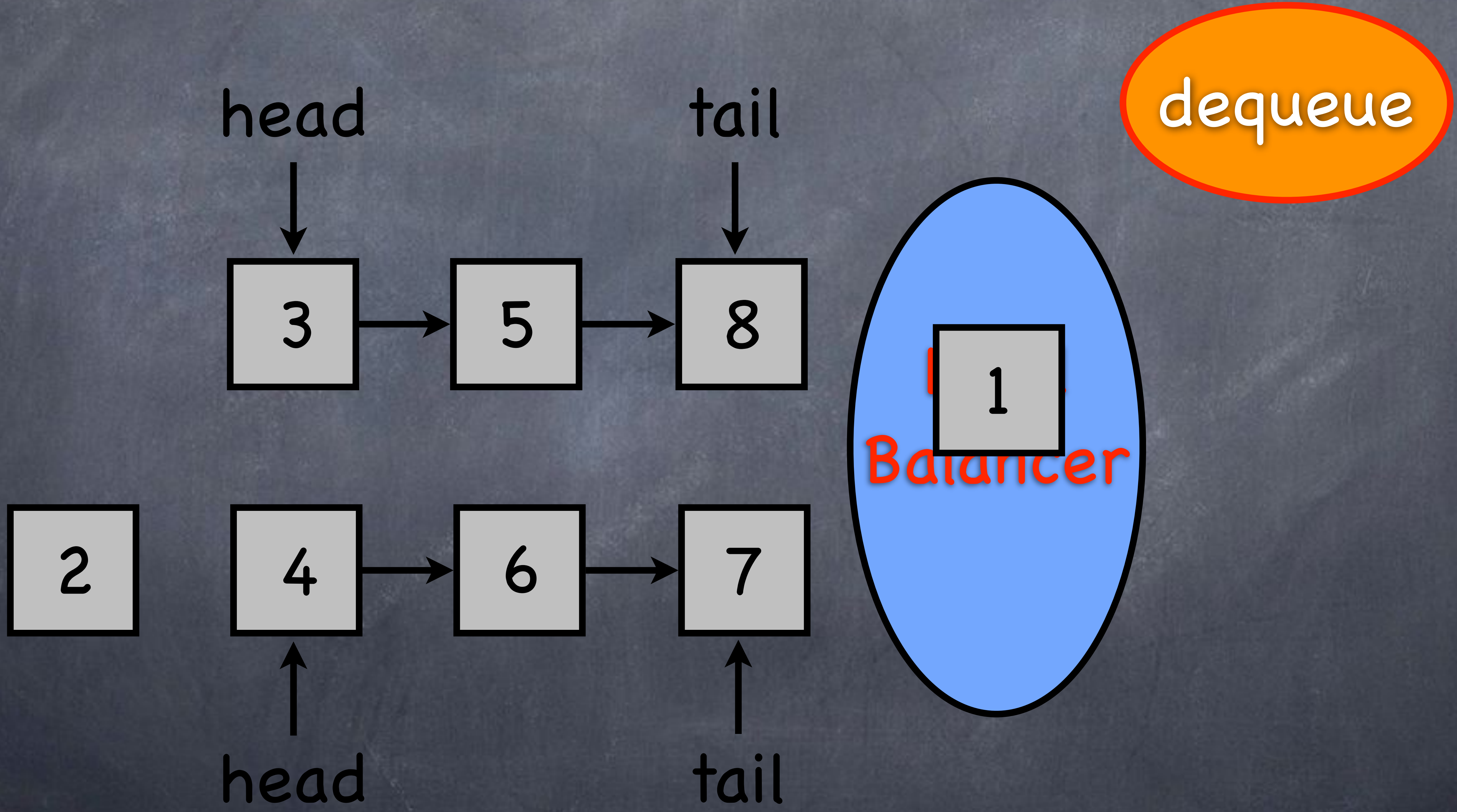
Load Balancing



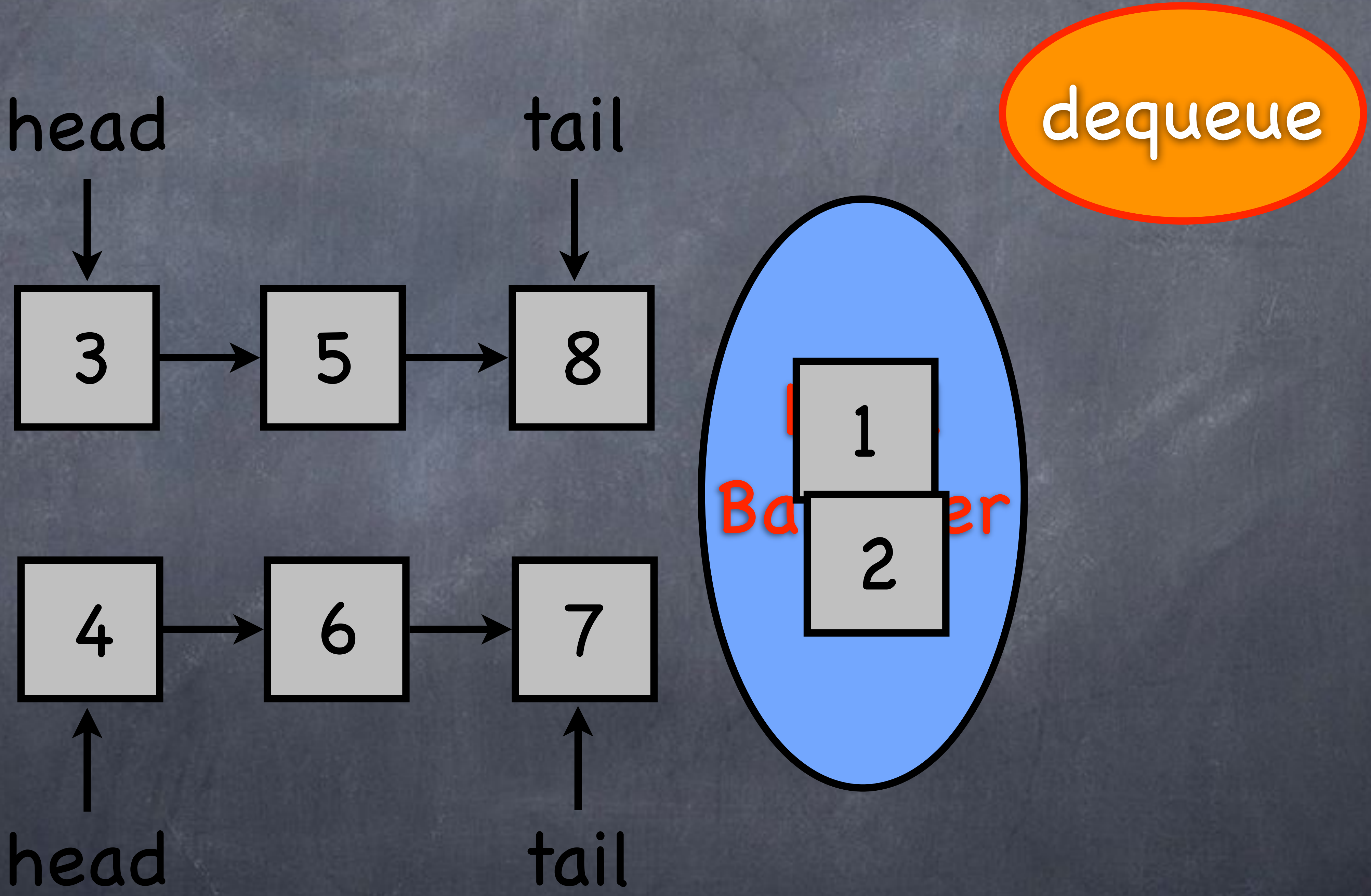
Load Balancing



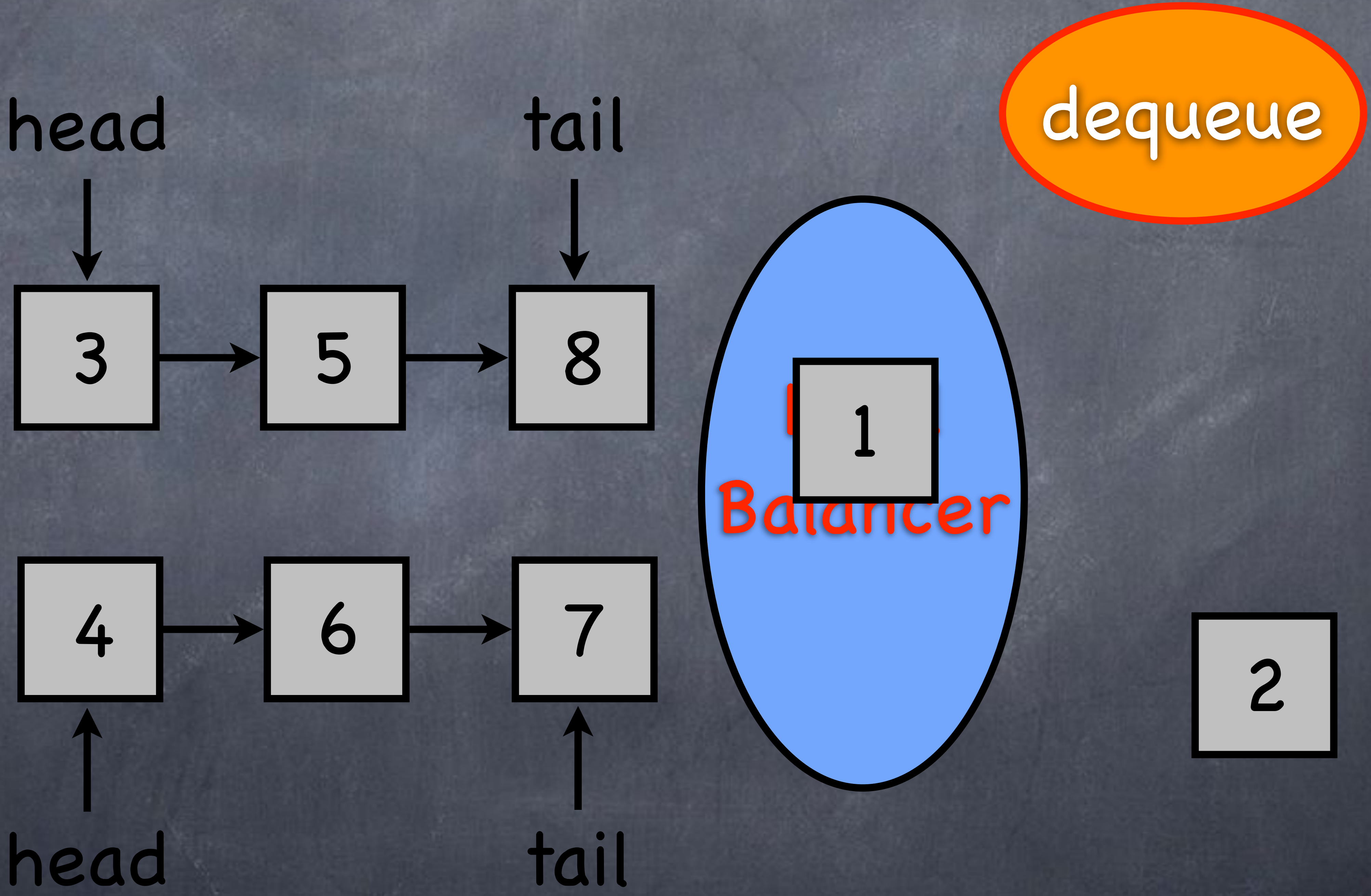
Load Balancing



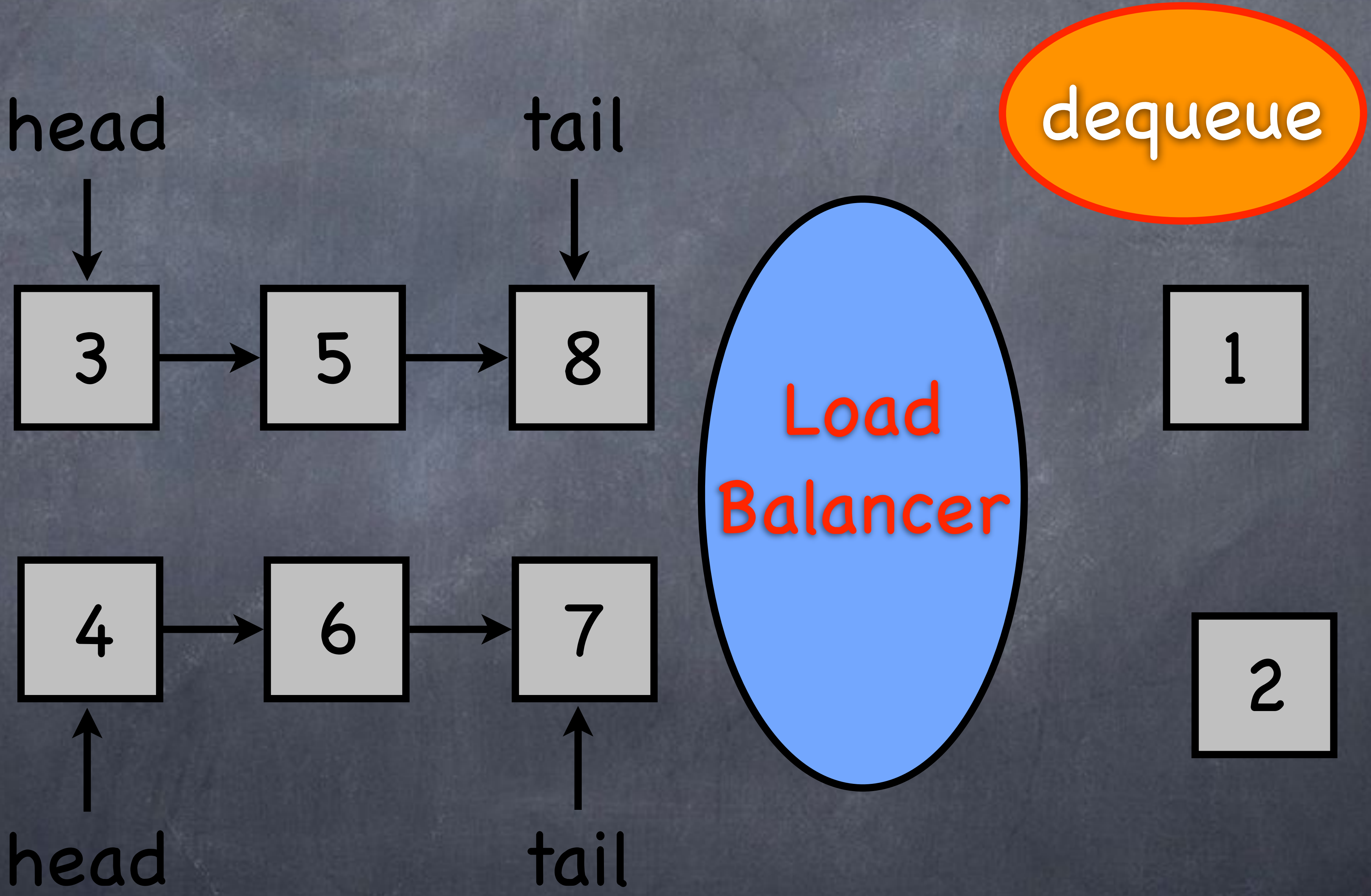
Load Balancing



Load Balancing

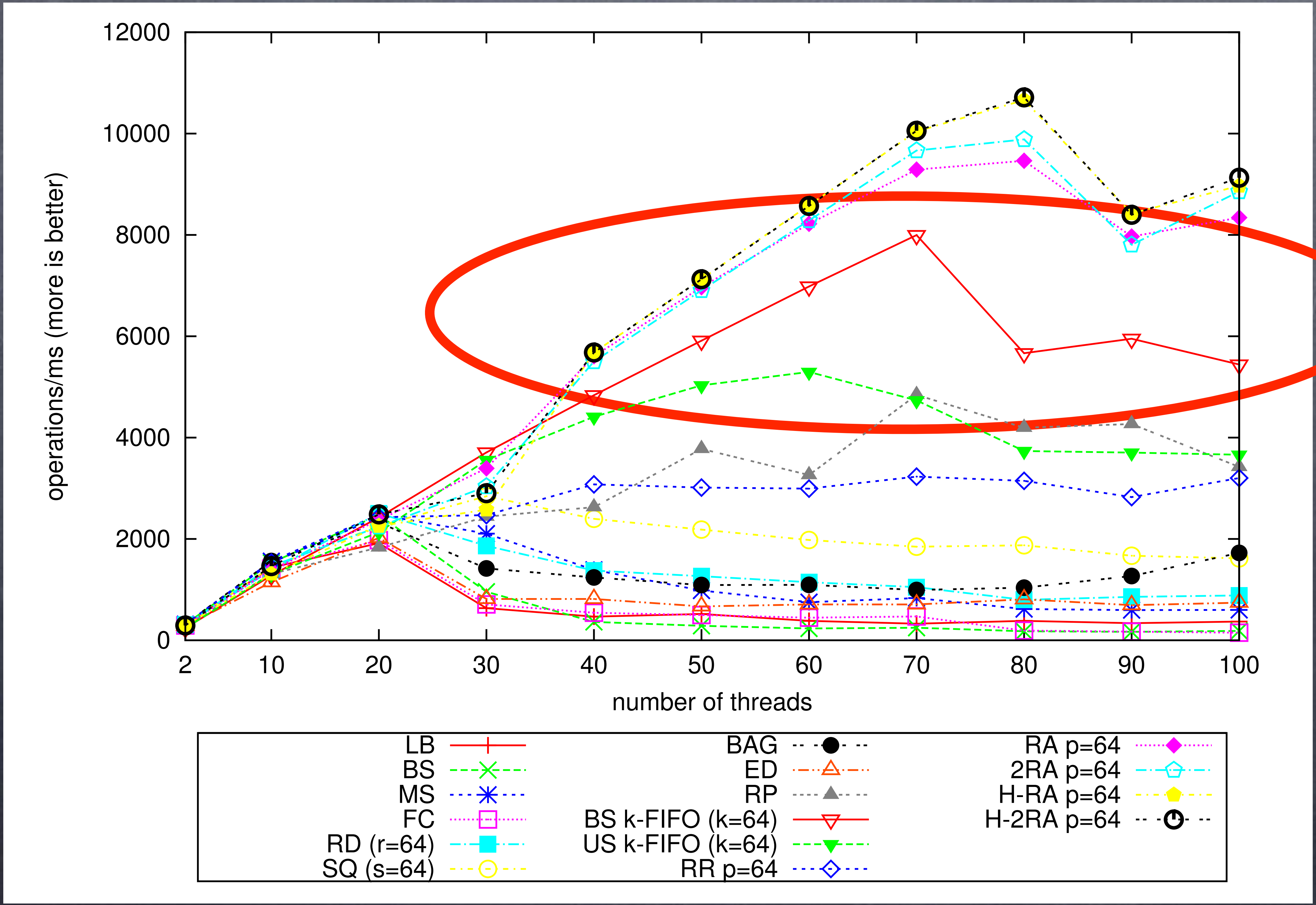


Load Balancing



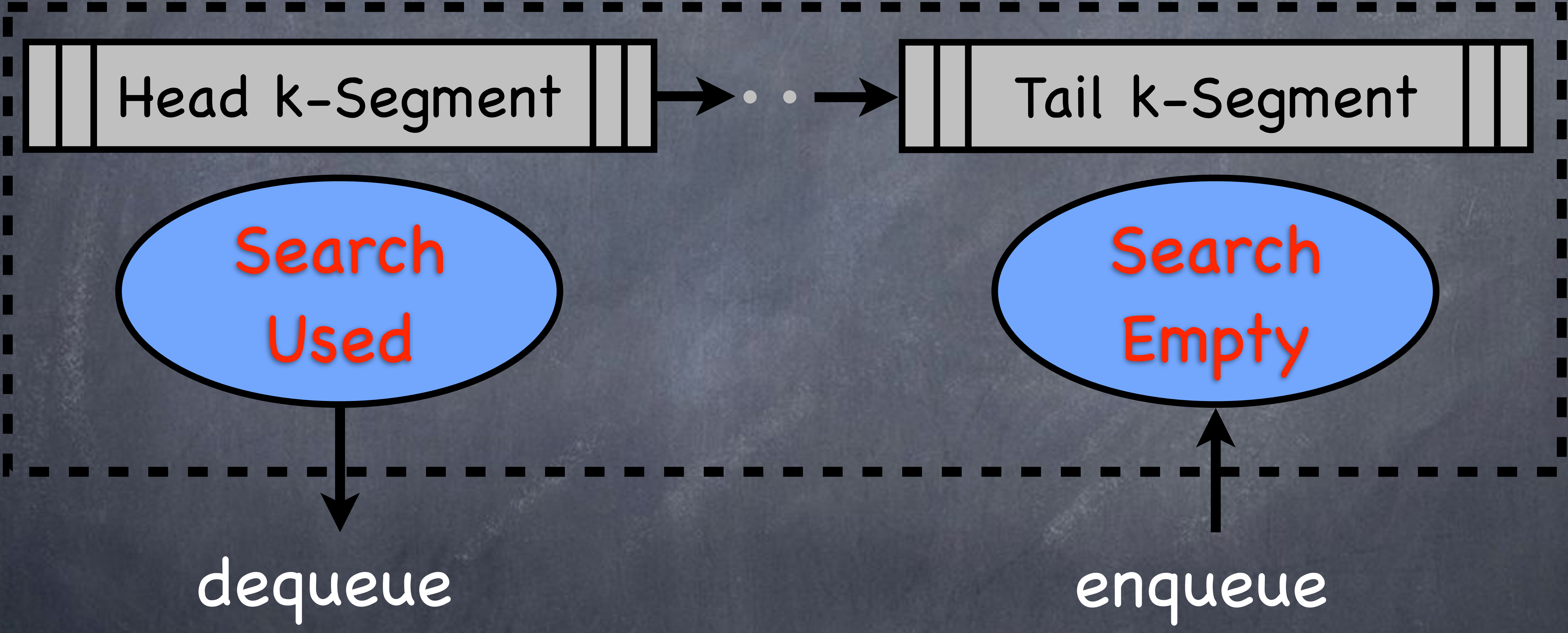
Emptiness Check?

Segmented Queues



Segmented Queues

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



Emptiness Check?

enqueue

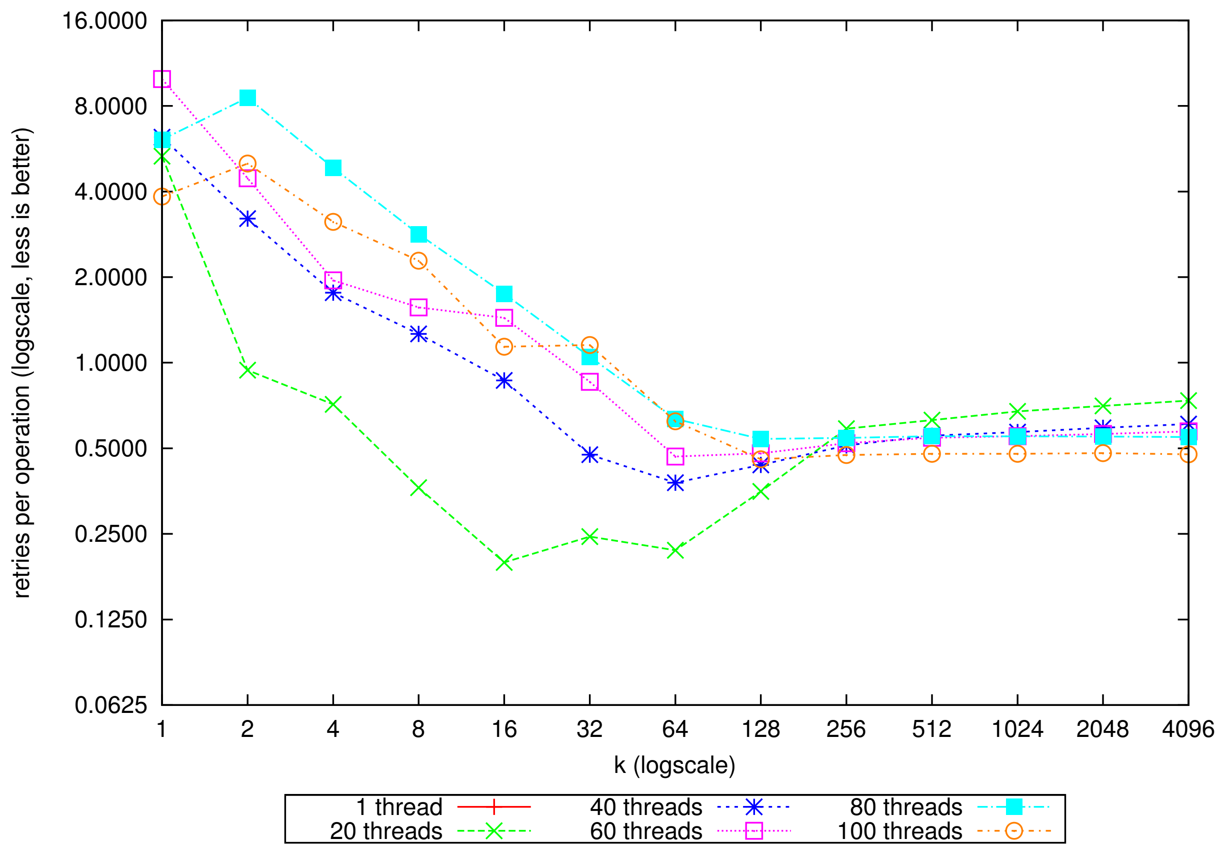
```
bool enqueue(item):
  while true:
    tail_old = get_tail();
    head_old = get_head();
    item_old, index = find_empty_slot(tail_old, k, TESTS);
    if tail_old == get_tail():
      if item_old.value == EMPTY:
        item_new = atomic_value(item, item_old.counter + 1);
        if CAS(&tail_old[index], item_old, item_new):
          if committed(tail_old, item_new, index):
            return true;
      else:
        if queue_full(head_old, tail_old):
          if segment_not_empty(head_old, k) && head == get_head():
            return false;
          advance_head(head_old, k);
        advance_tail(tail_old, k);

bool committed(tail_old, item_new, index):
  if tail_old[index] != item_new:
    return true;
  head_current = get_head();
  tail_current = get_tail();
  item_empty = atomic_value(EMPTY, item_new.counter + 1);
  if in_queue_after_head(tail_old, tail_current, head_current):
    return true;
  else if not_in_queue(tail_old, tail_current, head_current):
    if !CAS(&tail_old[index], item_new, item_empty):
      return true;
  else: //in queue at head
    head_new = atomic_value(head_current.value, head_current.counter + 1);
    if CAS(&head, head_current, head_new):
      return true;
    if !CAS(&tail_old[index], item_new, item_empty):
      return true;
  return false;
```

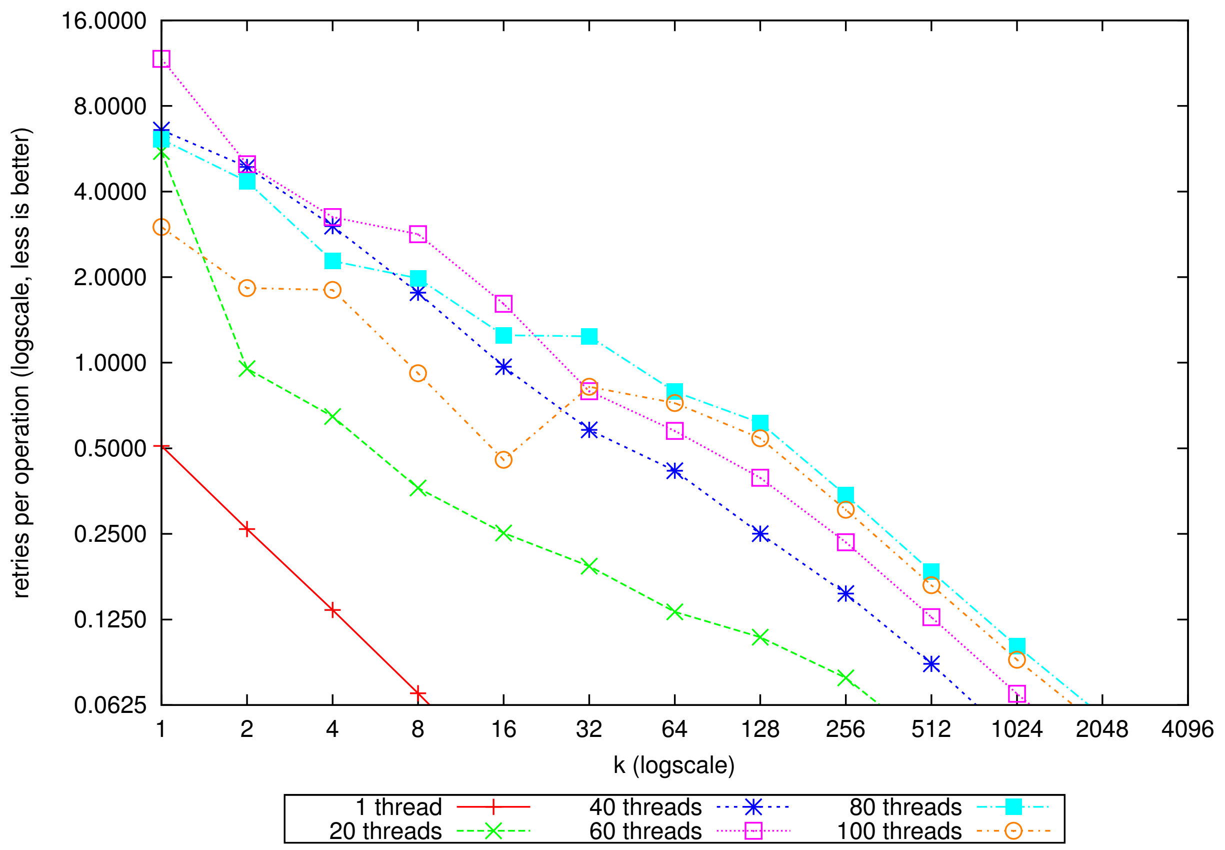
dequeue

```
item dequeue():
  while true:
    tail_old = get_tail();
    head_old = get_head();
    item_old, index = find_item(head_old, k);
    if head_old == head:
      if item_old.value != EMPTY:
        if head_old.value == tail_old.value:
          advance_tail(tail_old, k);
        item_empty = atomic_value(EMPTY, item_old.counter + 1);
        if CAS(&head_old[index], item_old, item_empty):
          return item_old.value;
      else:
        if head_old.value == tail_old.value && tail_old == get_tail():
          return null;
        advance_head(head_old, k);
```

CAS Retries/Operation in k

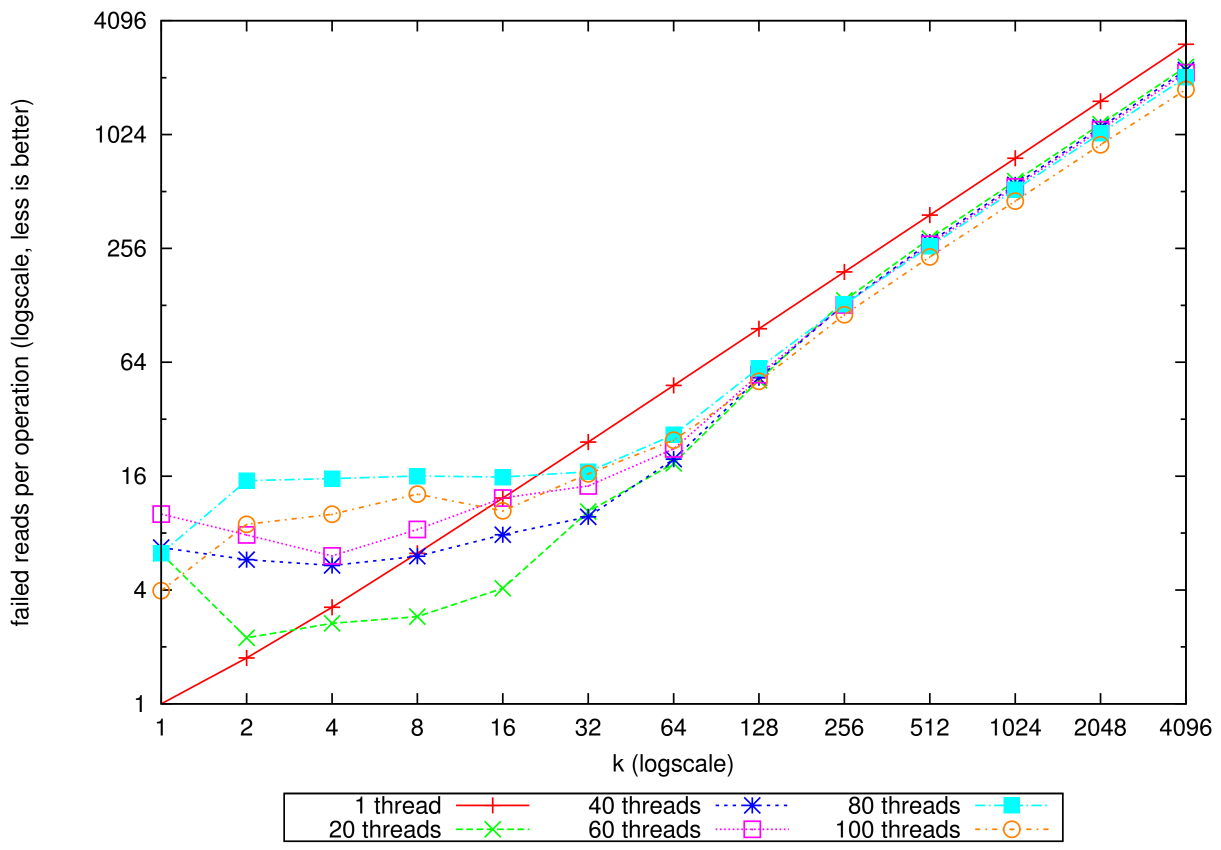


(a) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)

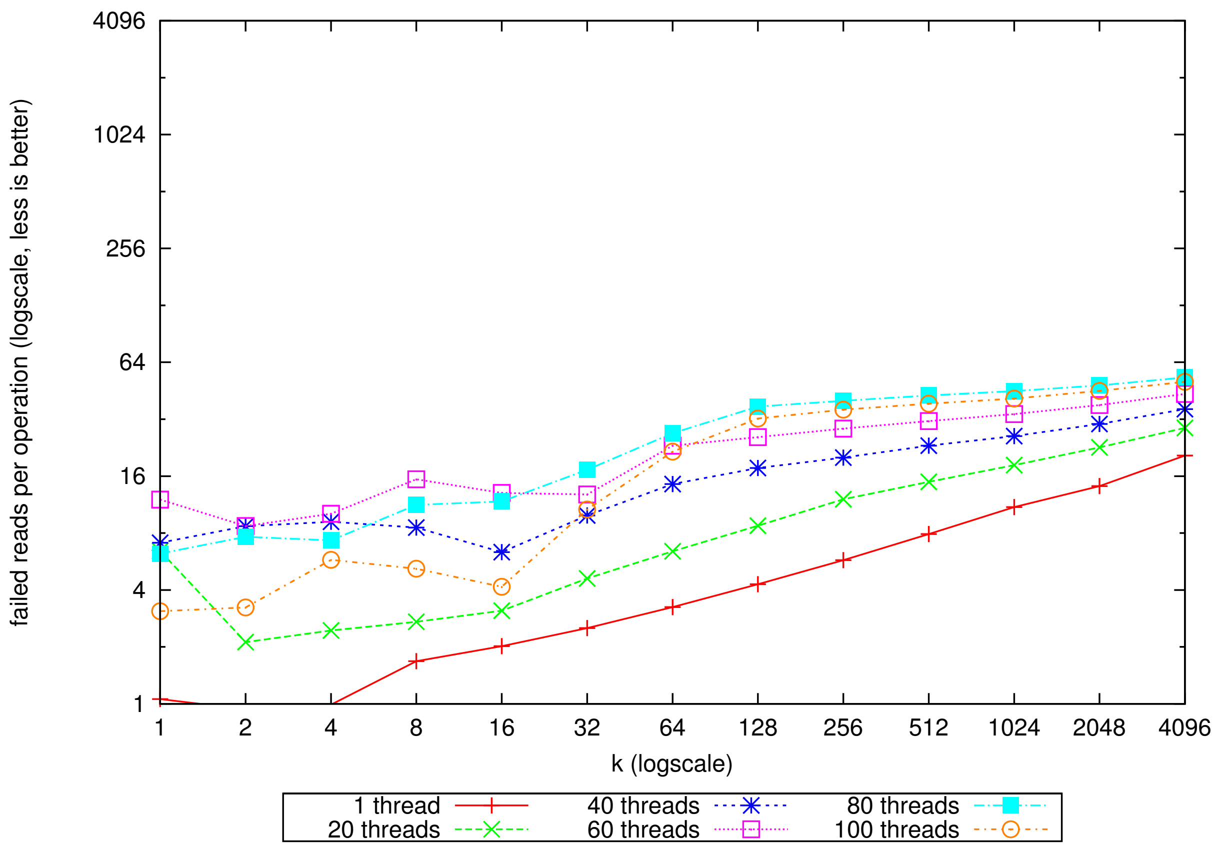


(b) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 5000$)

Failed Searches/Operation in k

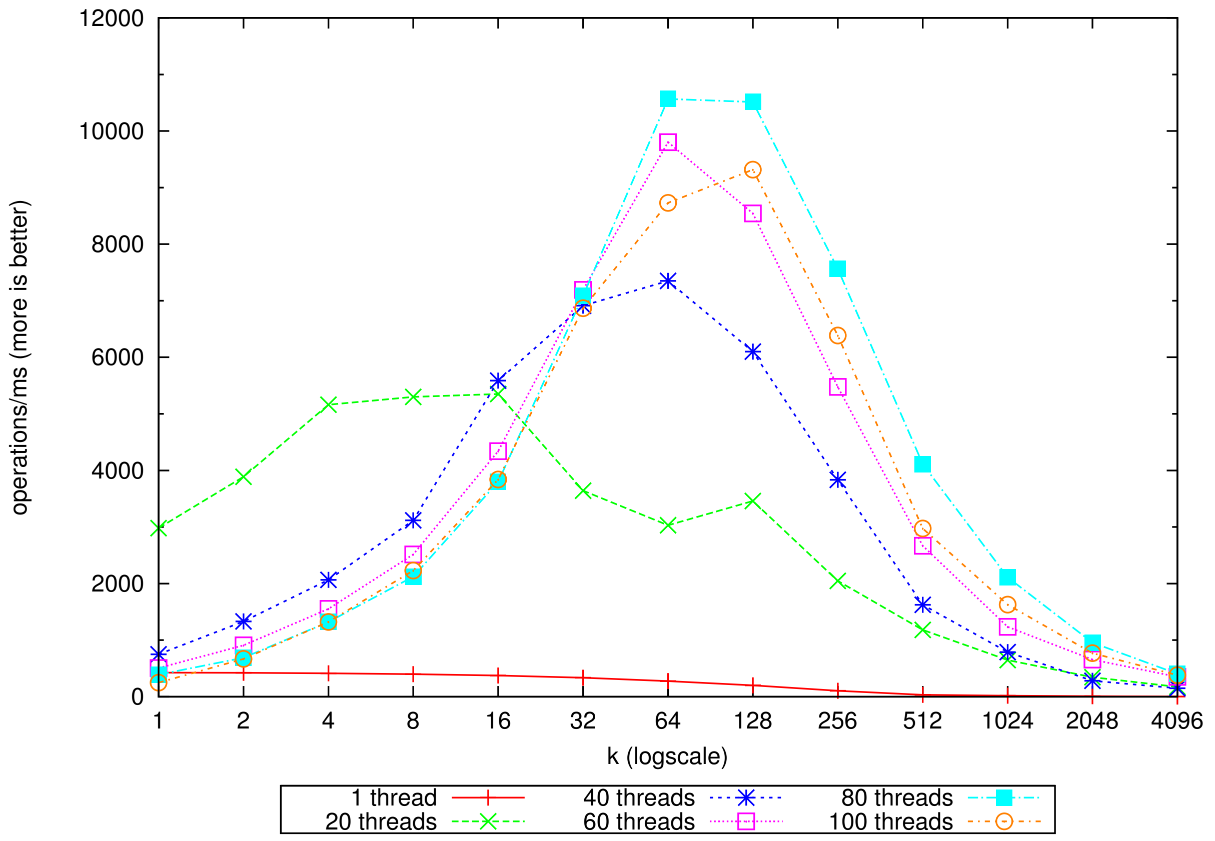


(c) BS number of failed reads per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)

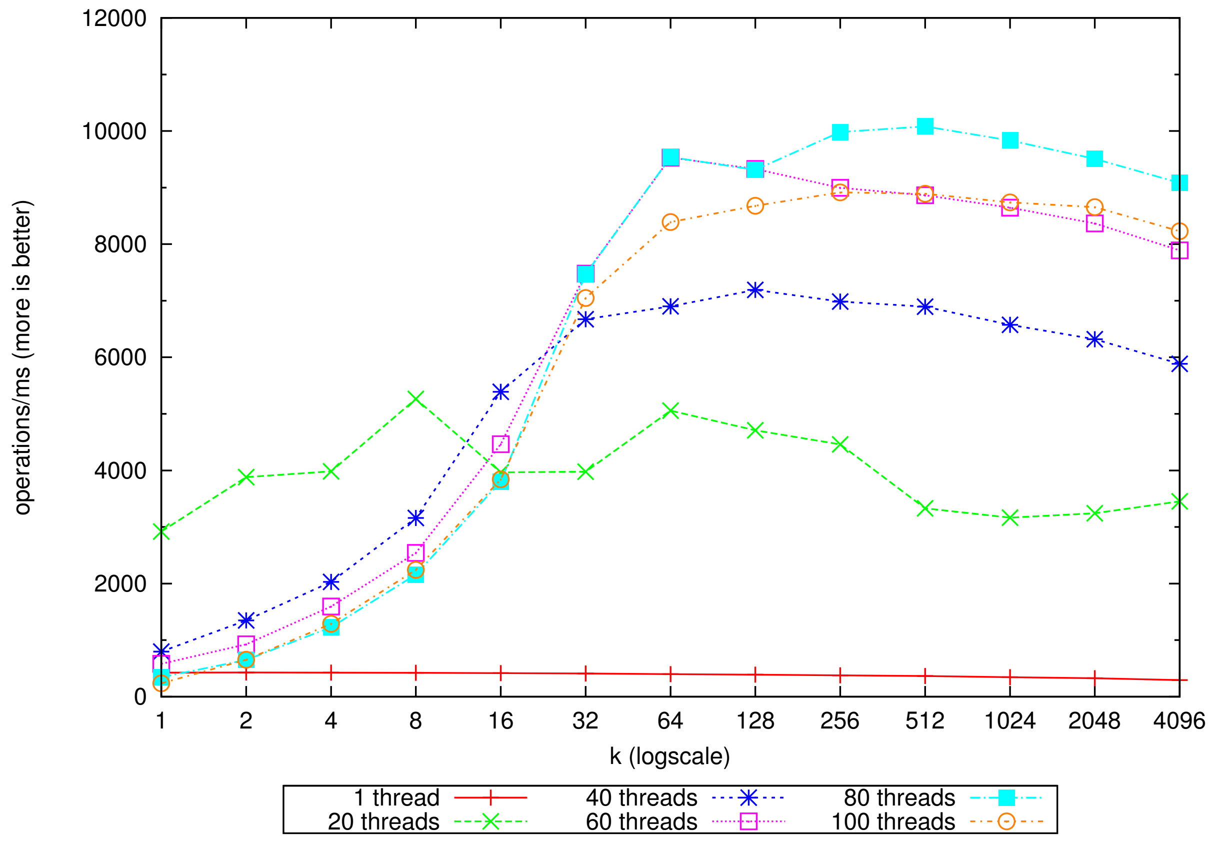


(d) BS number of failed reads per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 5000$)

Performance in k

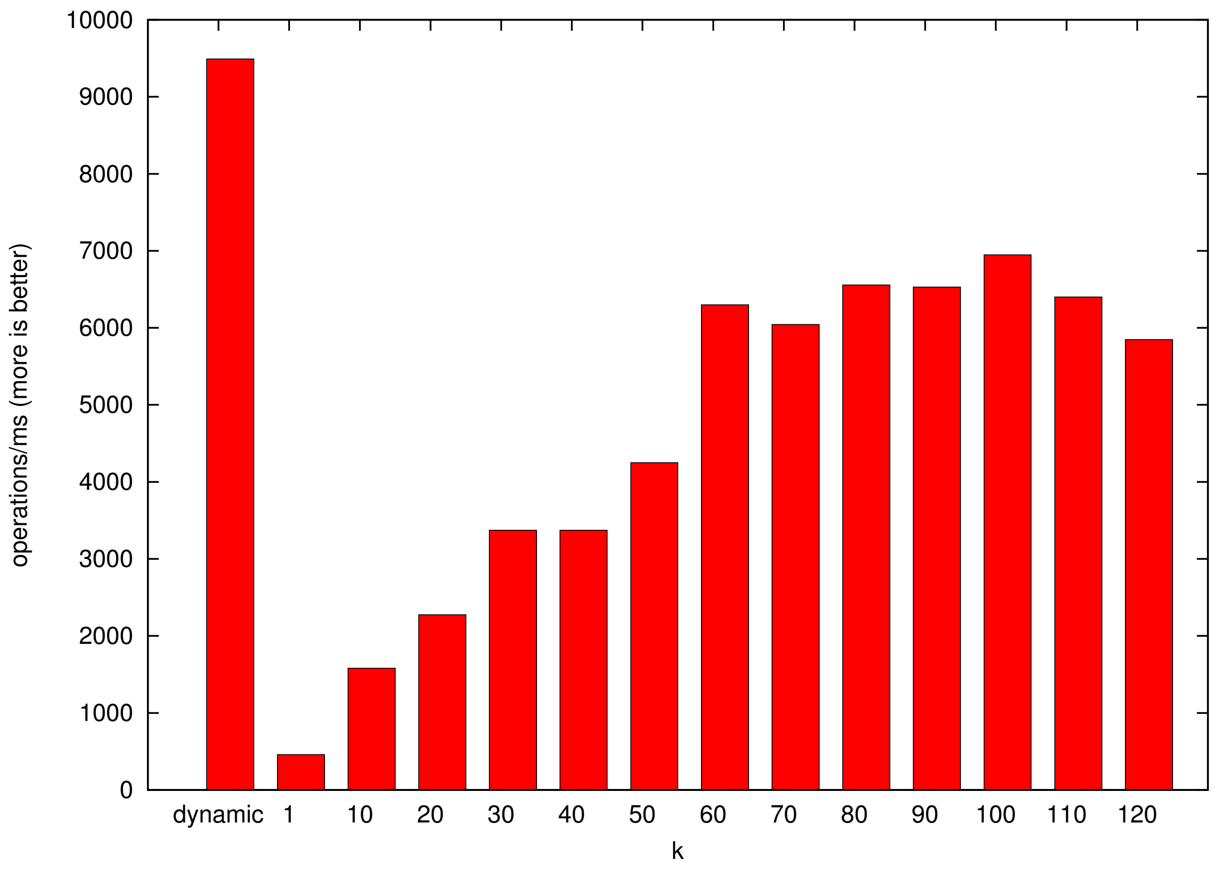


(e) BS performance of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)

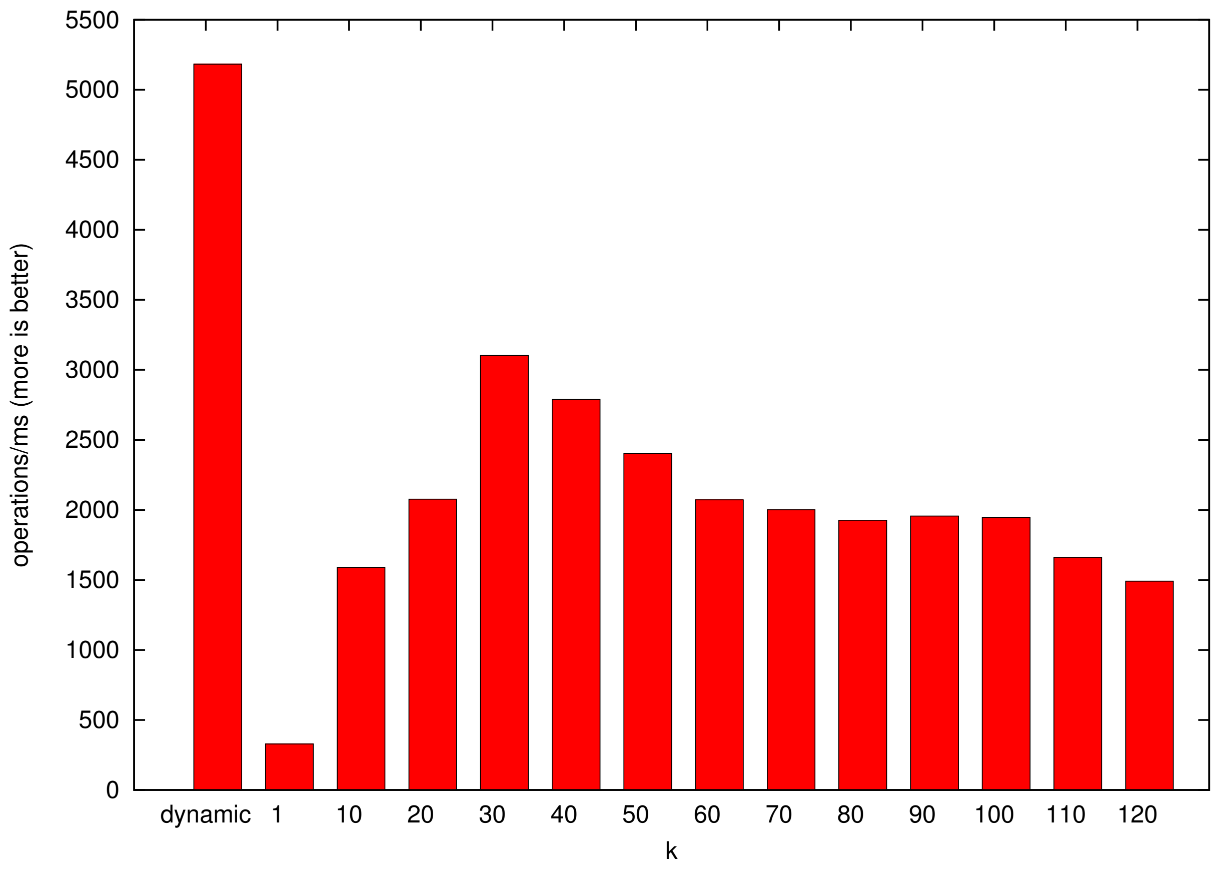


(f) BS performance of very high contention producer-consumer benchmark ($c = 1000$, $i = 5000$)

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+1 dequeue operations that may return elements not younger than k (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+1 dequeue operations that may return elements not younger than k (or return nothing)
- starvation-free for finite k

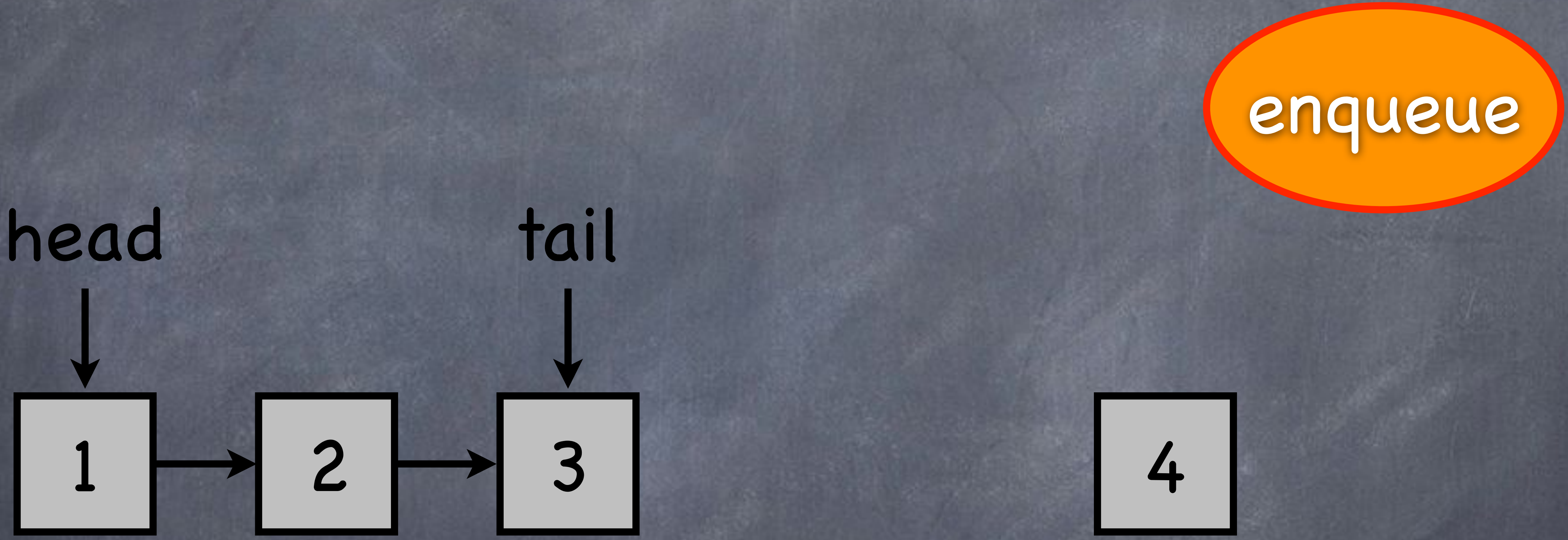
Concurrent k-FIFO Queue

- 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+1 dequeue operations that may return elements not younger than k (or return nothing)
- starvation-free for finite k
- 0-FIFO queue = regular FIFO queue

Concurrent k-FIFO Queue

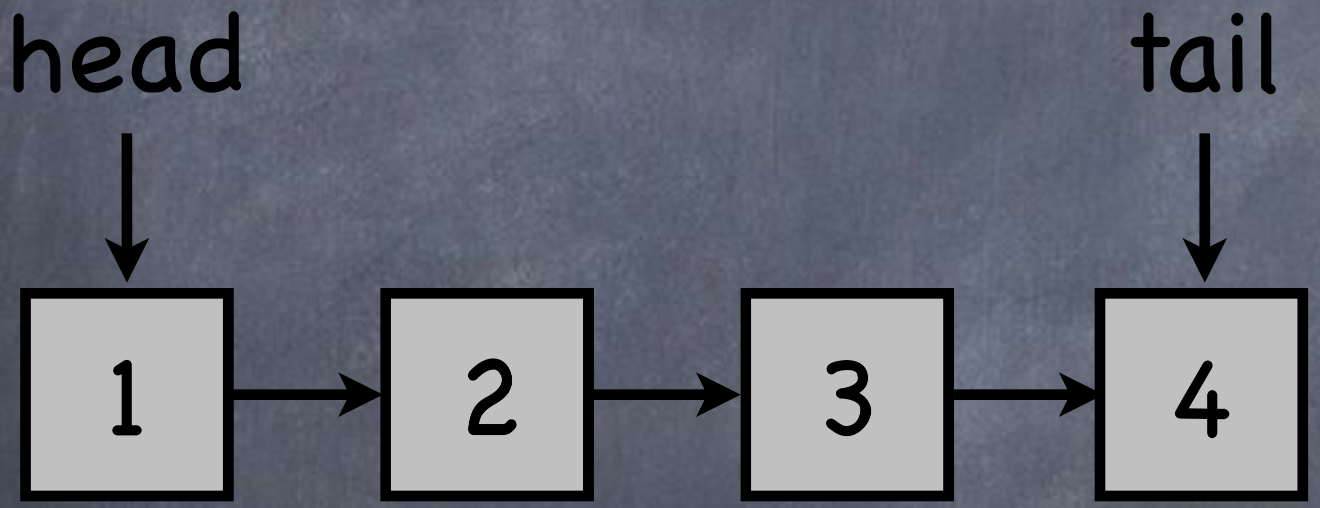
- with a k-FIFO queue elements may be returned out-of-FIFO order up to k
- the oldest element is returned after at most k+1 dequeue operations that may return elements not younger than k (or return nothing)
- starvation-free for finite k
- 0-FIFO queue = regular FIFO queue
- bigger k -> better performance,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

Concurrent 2-FIFO Queue (k=2)

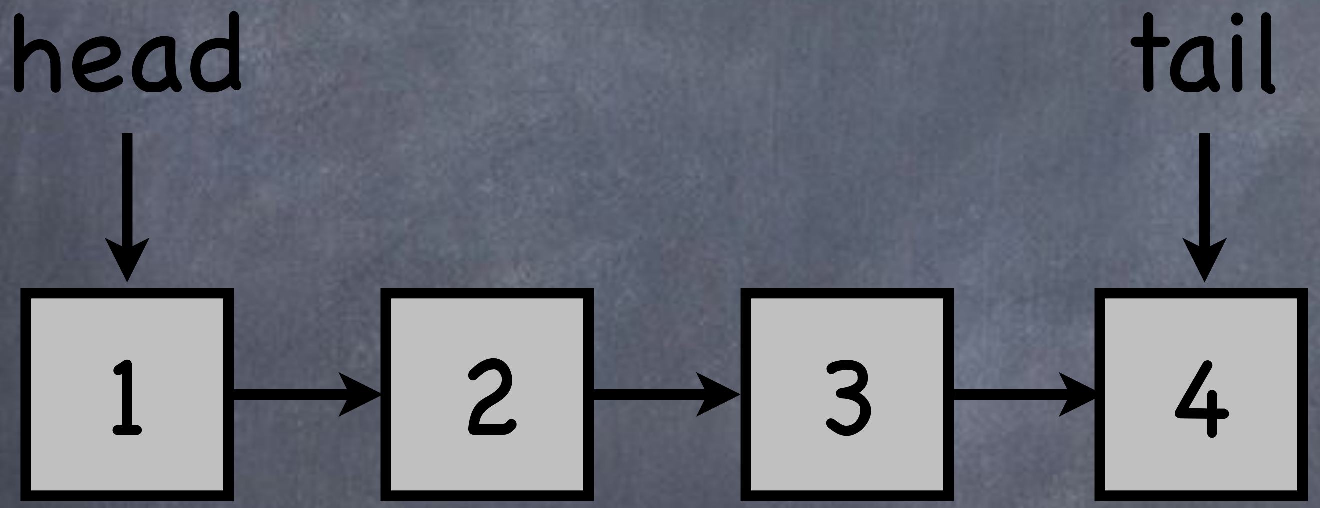


Concurrent 2-FIFO Queue (k=2)

enqueue

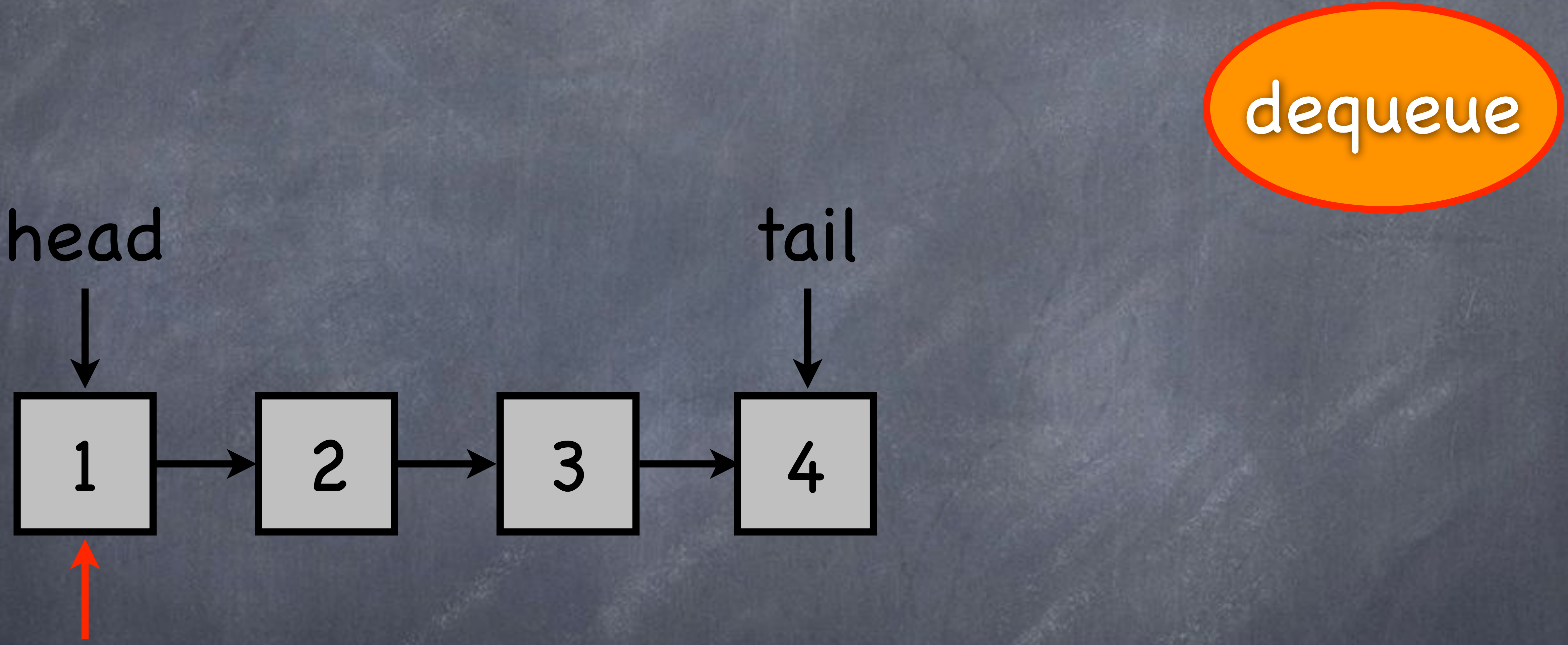


Concurrent 2-FIFO Queue (k=2)

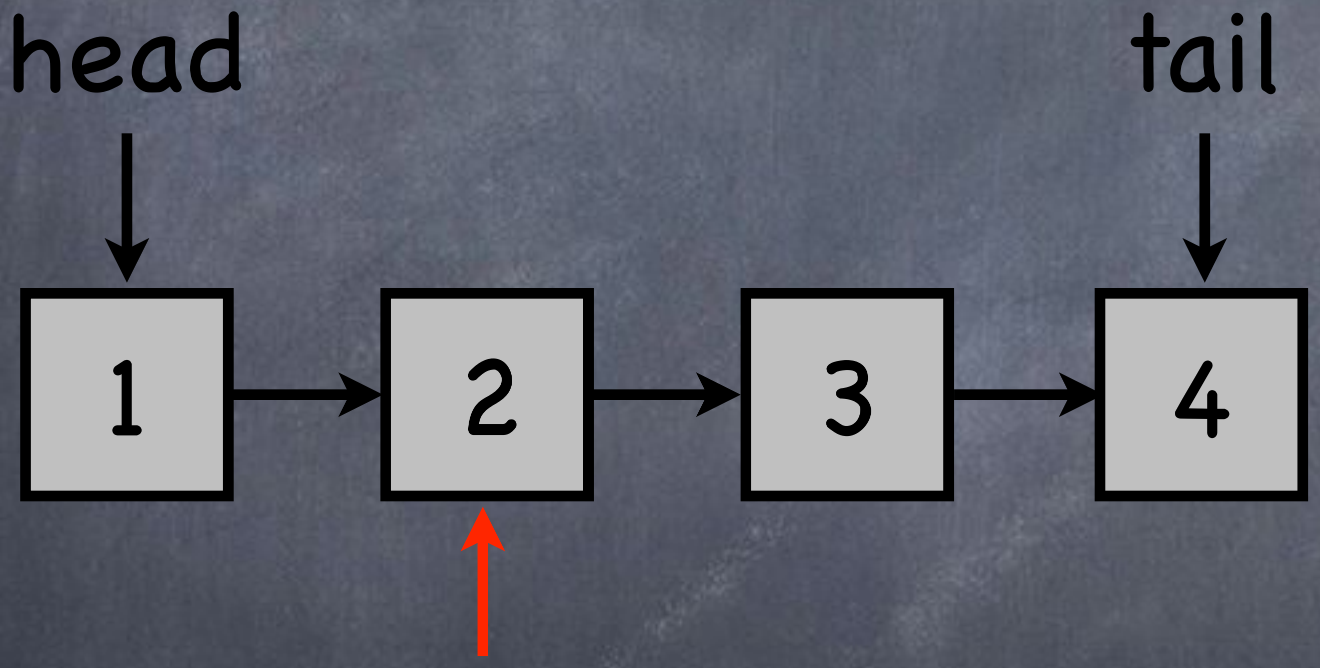


dequeue

Concurrent 2-FIFO Queue (k=2)

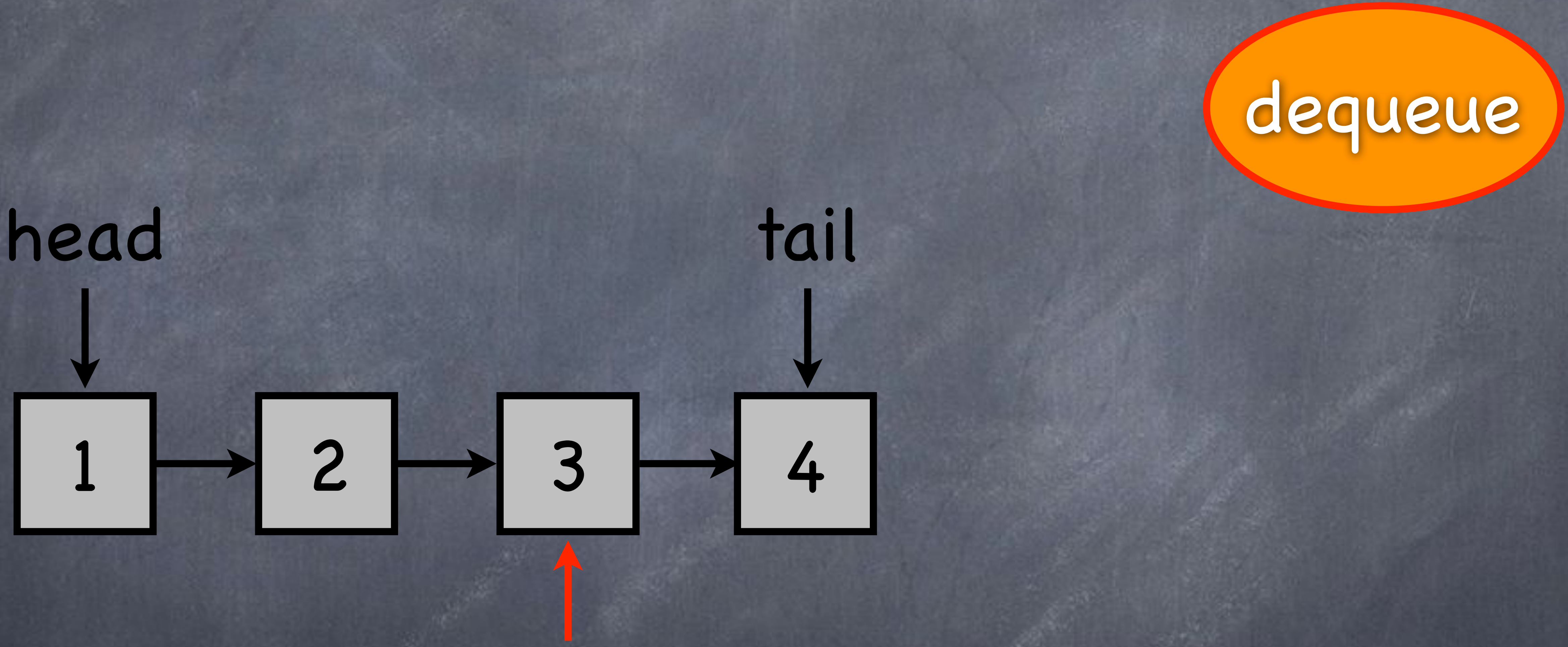


Concurrent 2-FIFO Queue (k=2)

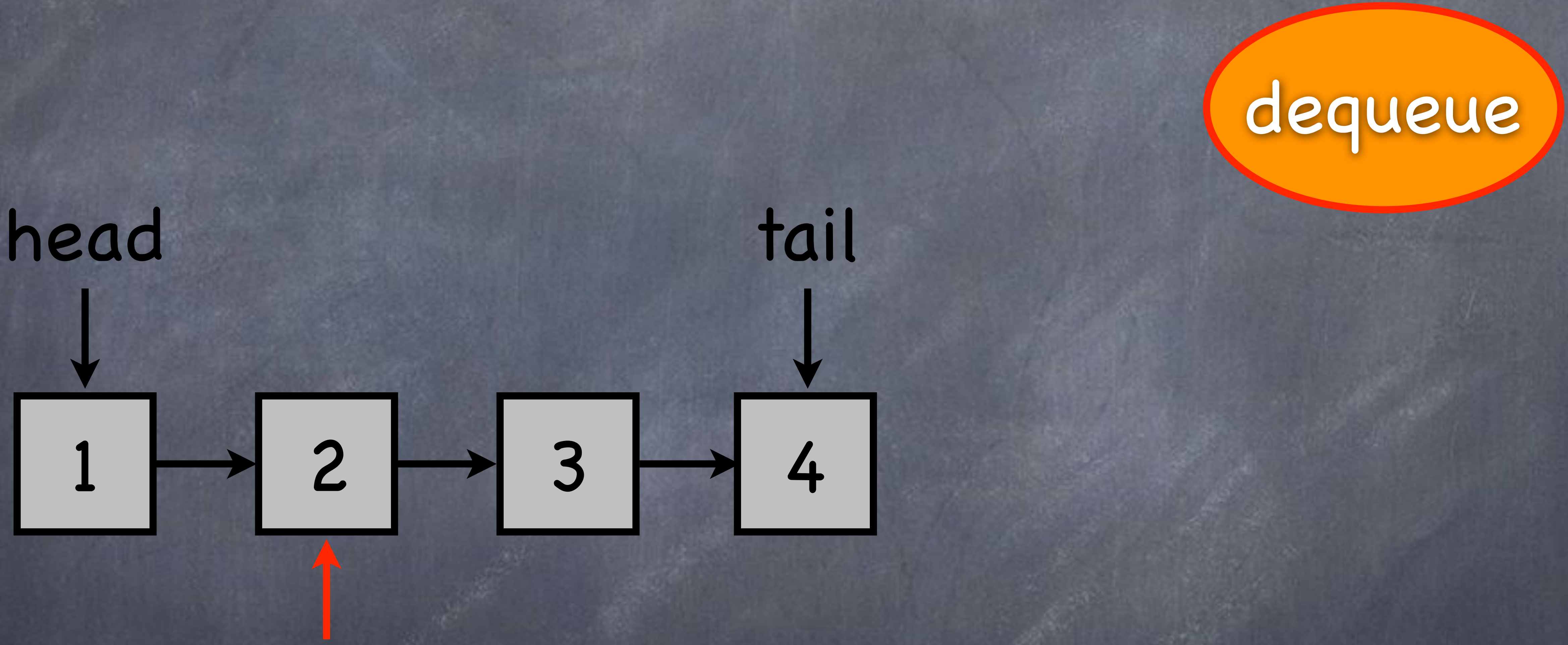


dequeue

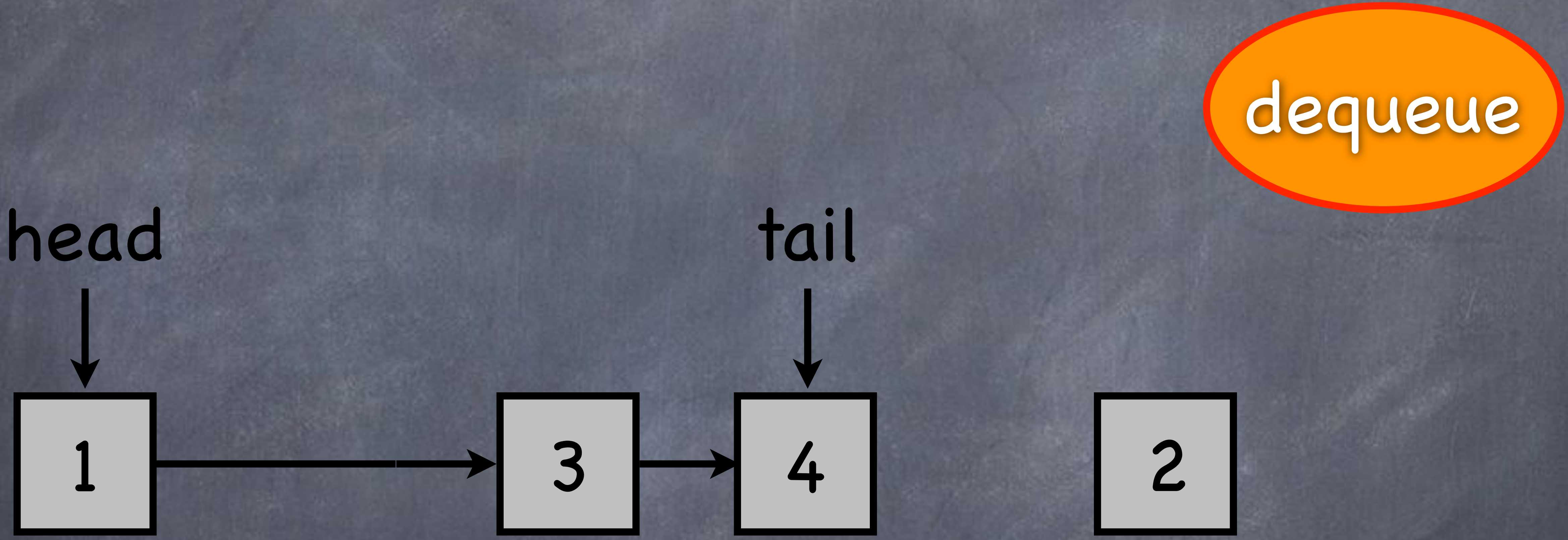
Concurrent 2-FIFO Queue (k=2)



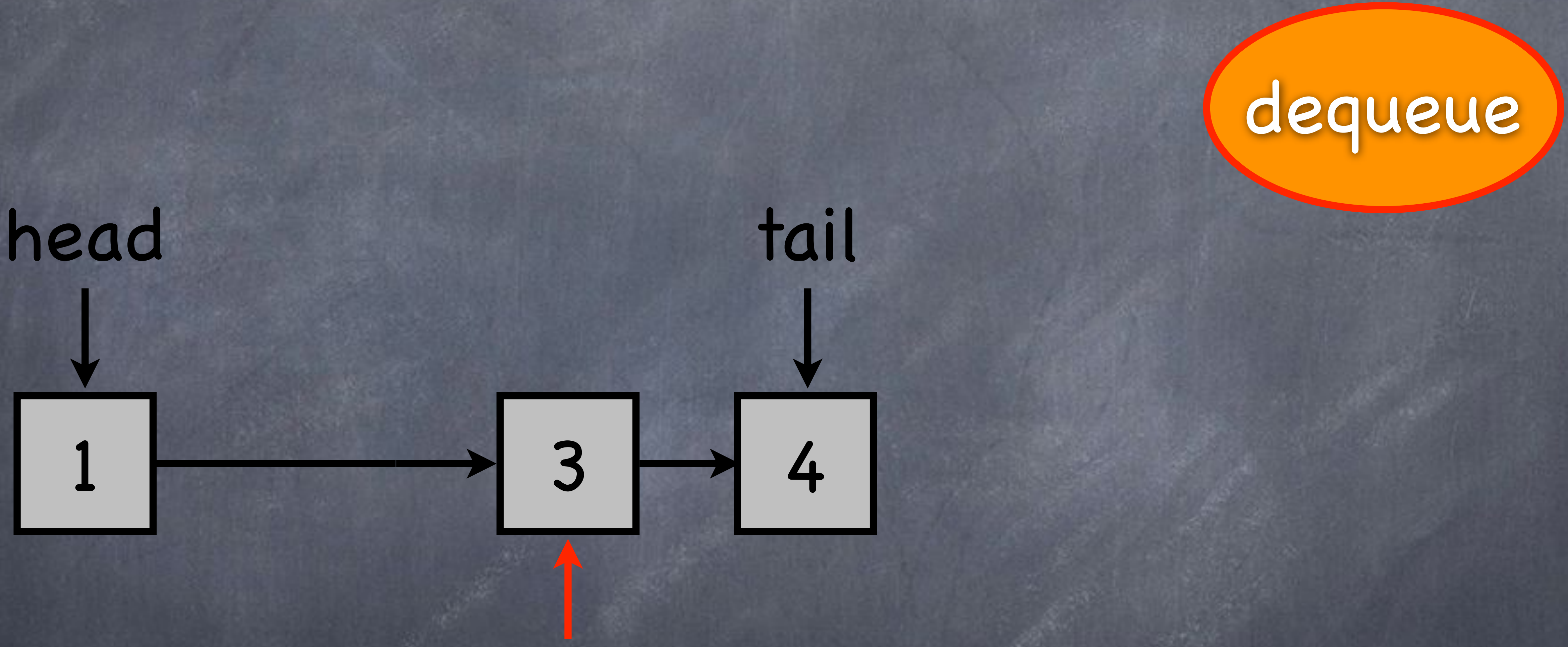
Concurrent 2-FIFO Queue (k=2)



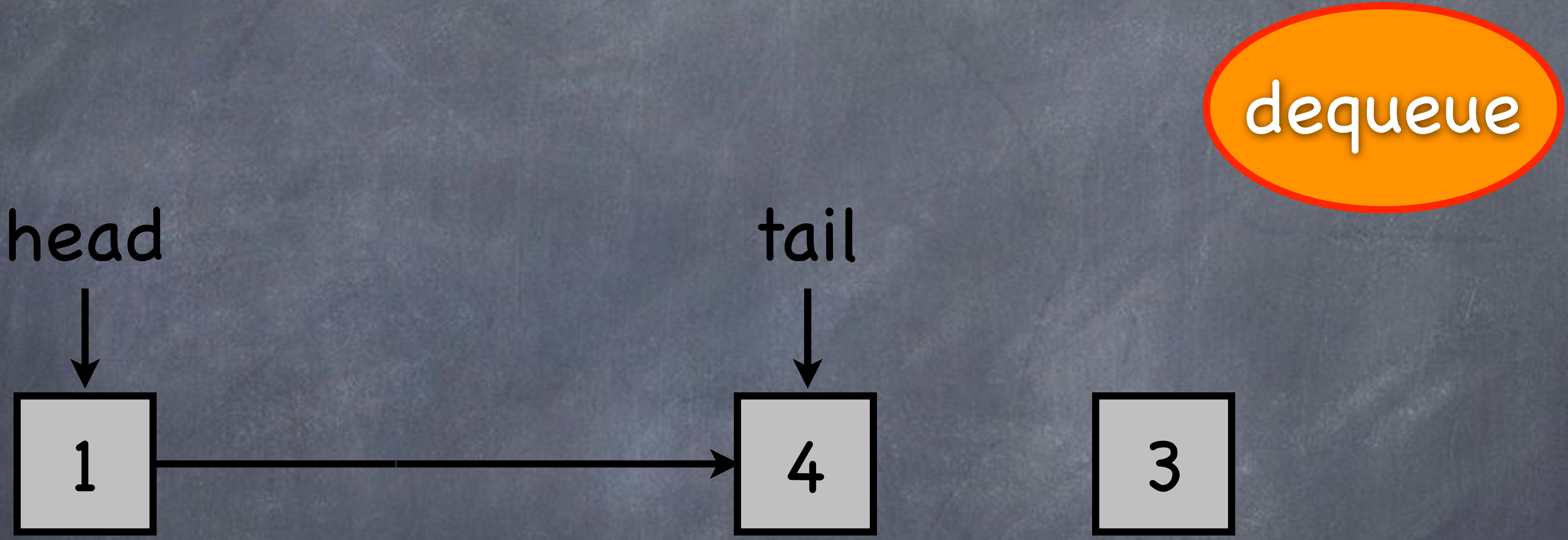
Concurrent 2-FIFO Queue (k=2)



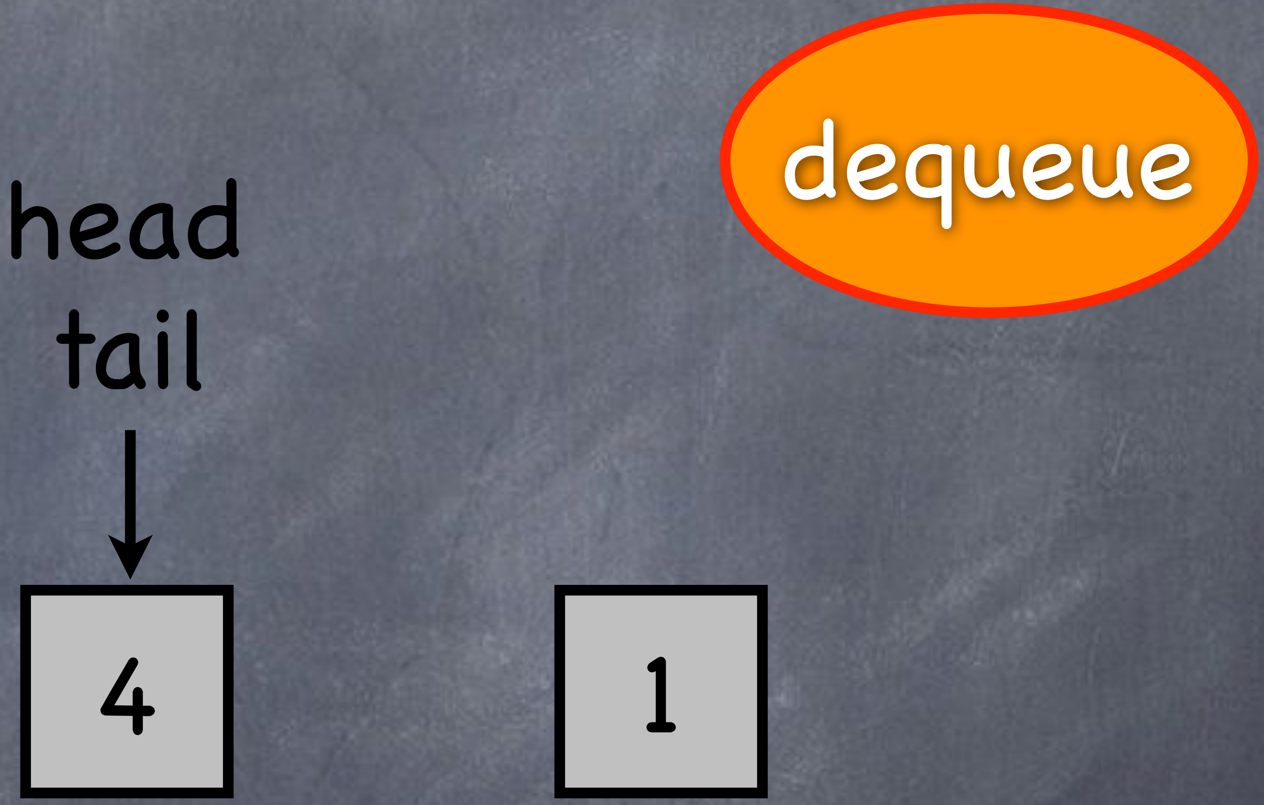
Concurrent 2-FIFO Queue (k=2)



Concurrent 2-FIFO Queue (k=2)



Concurrent 2-FIFO Queue (k=2)

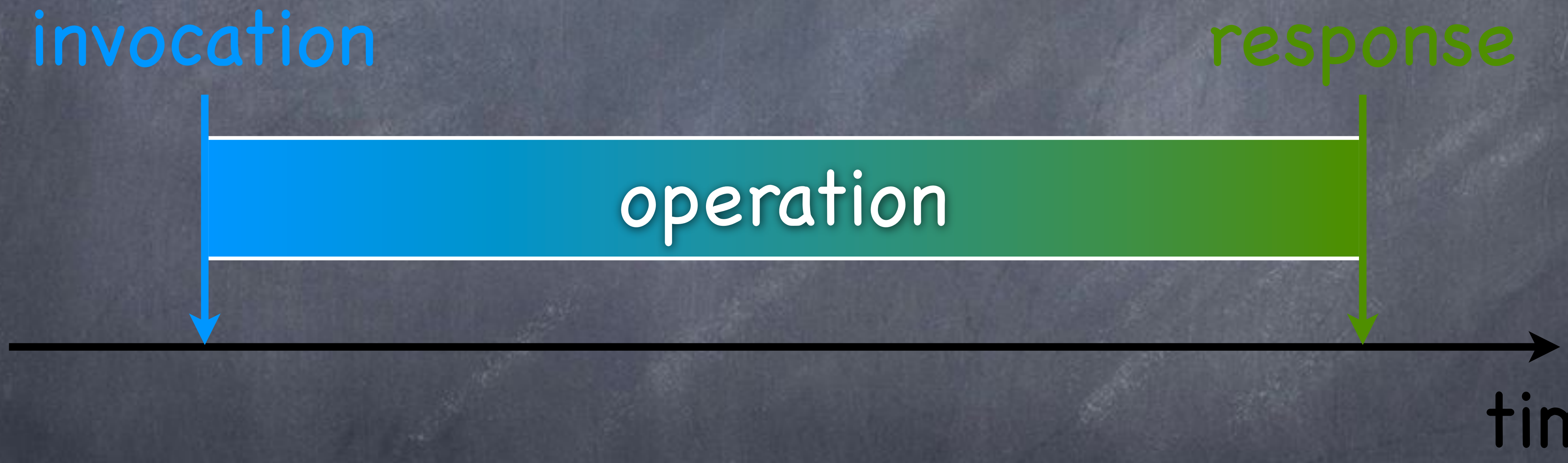


We call k
the <u>worst-case semantical deviation</u> (WCSD) of
a k-FIFO queue from
a regular FIFO queue

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

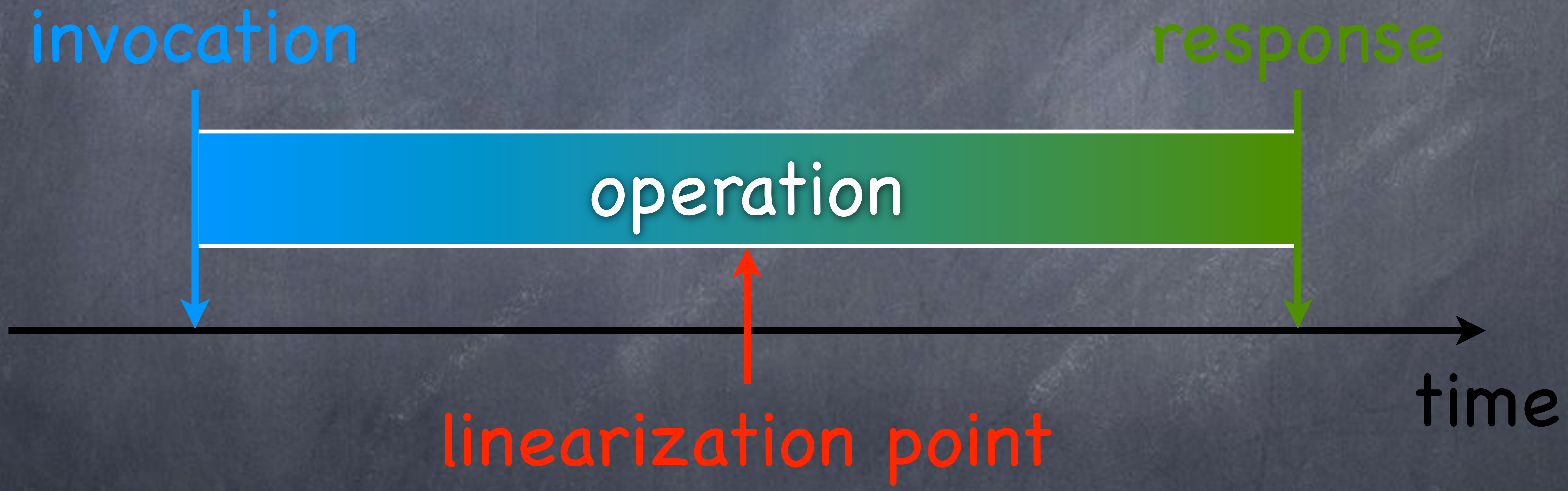
Execution History

Sequence of Time-Stamped Invocation and Response Events

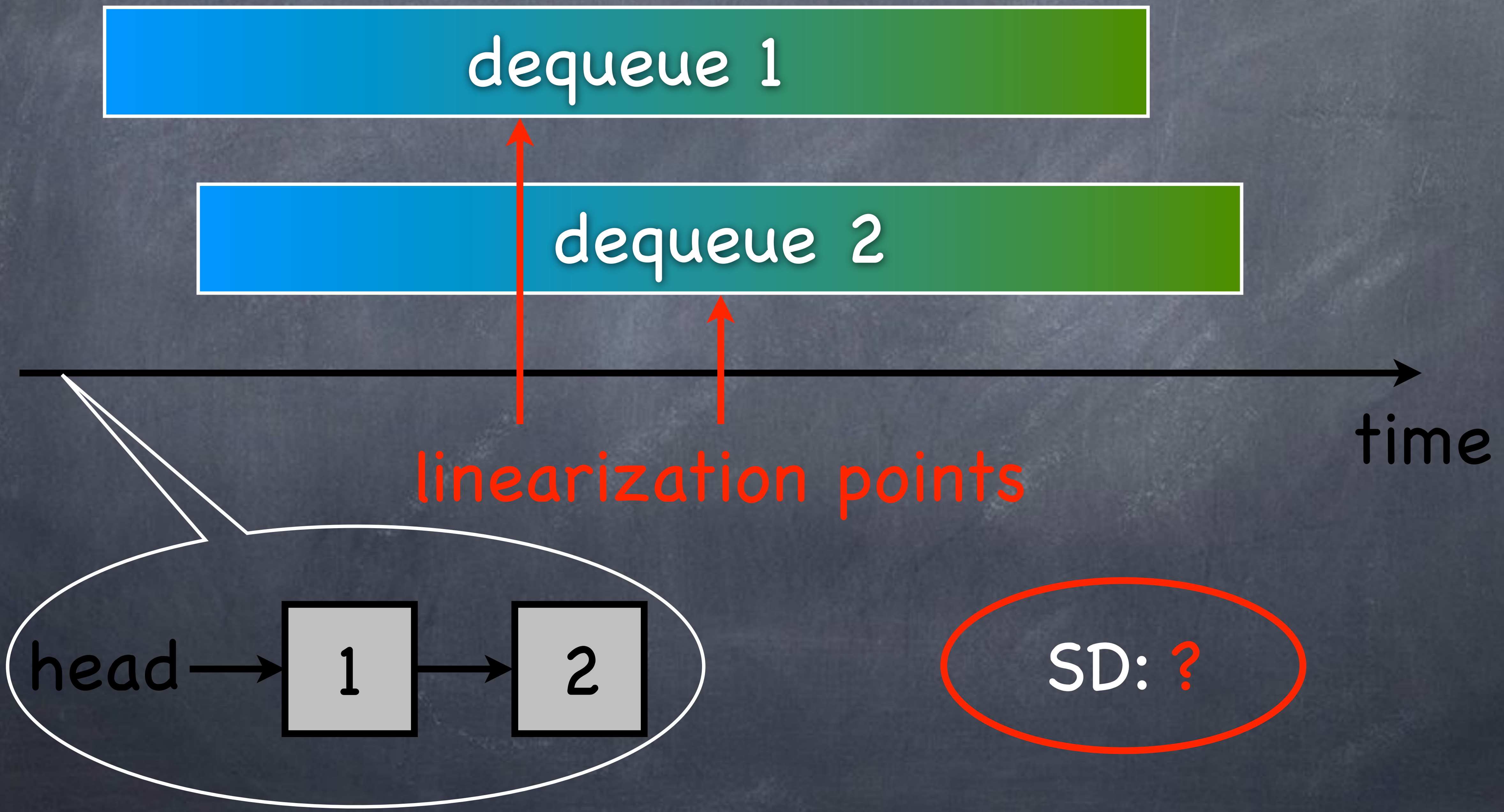


Execution History

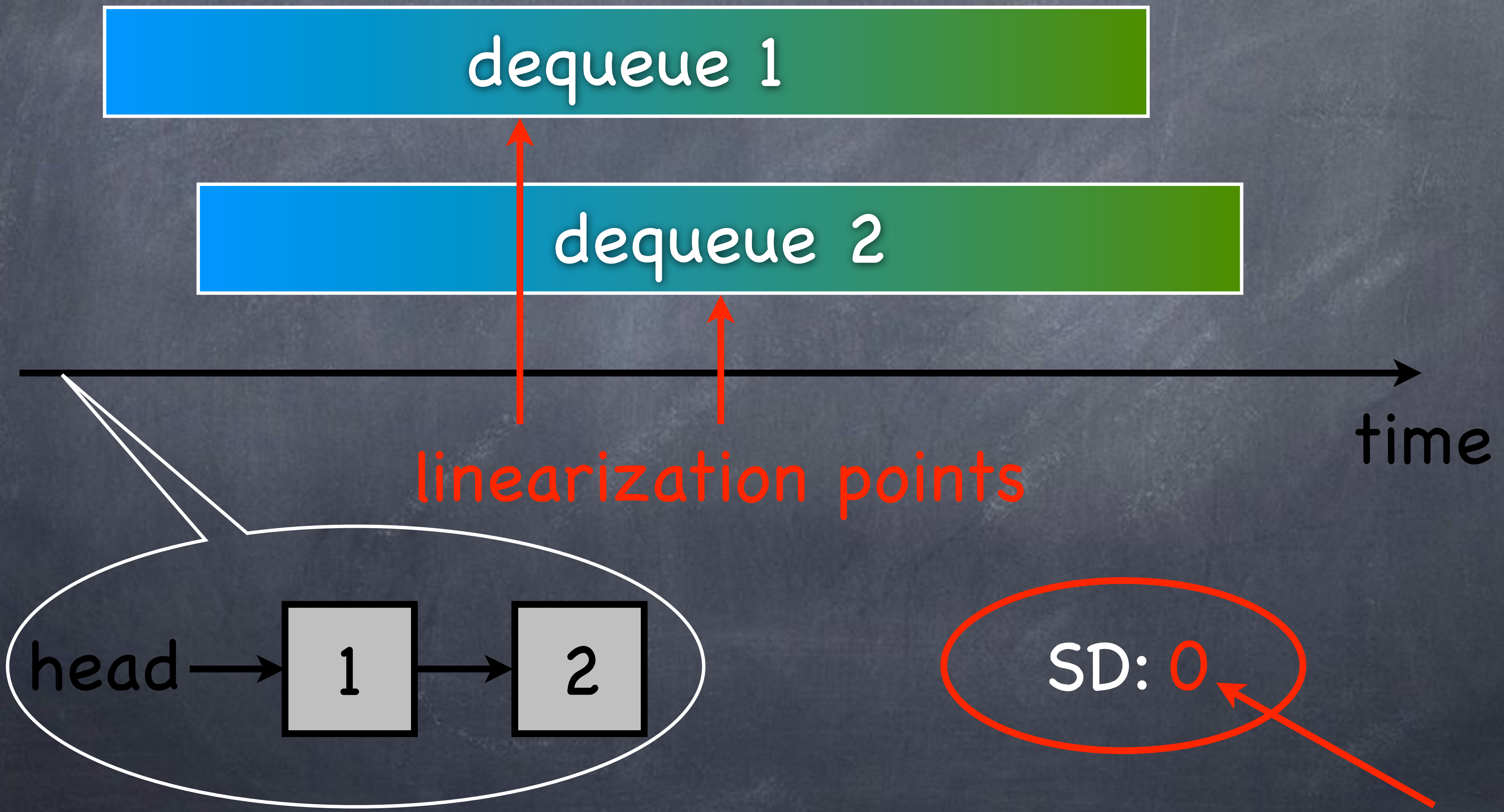
Sequence of Time-Stamped Invocation and Response Events



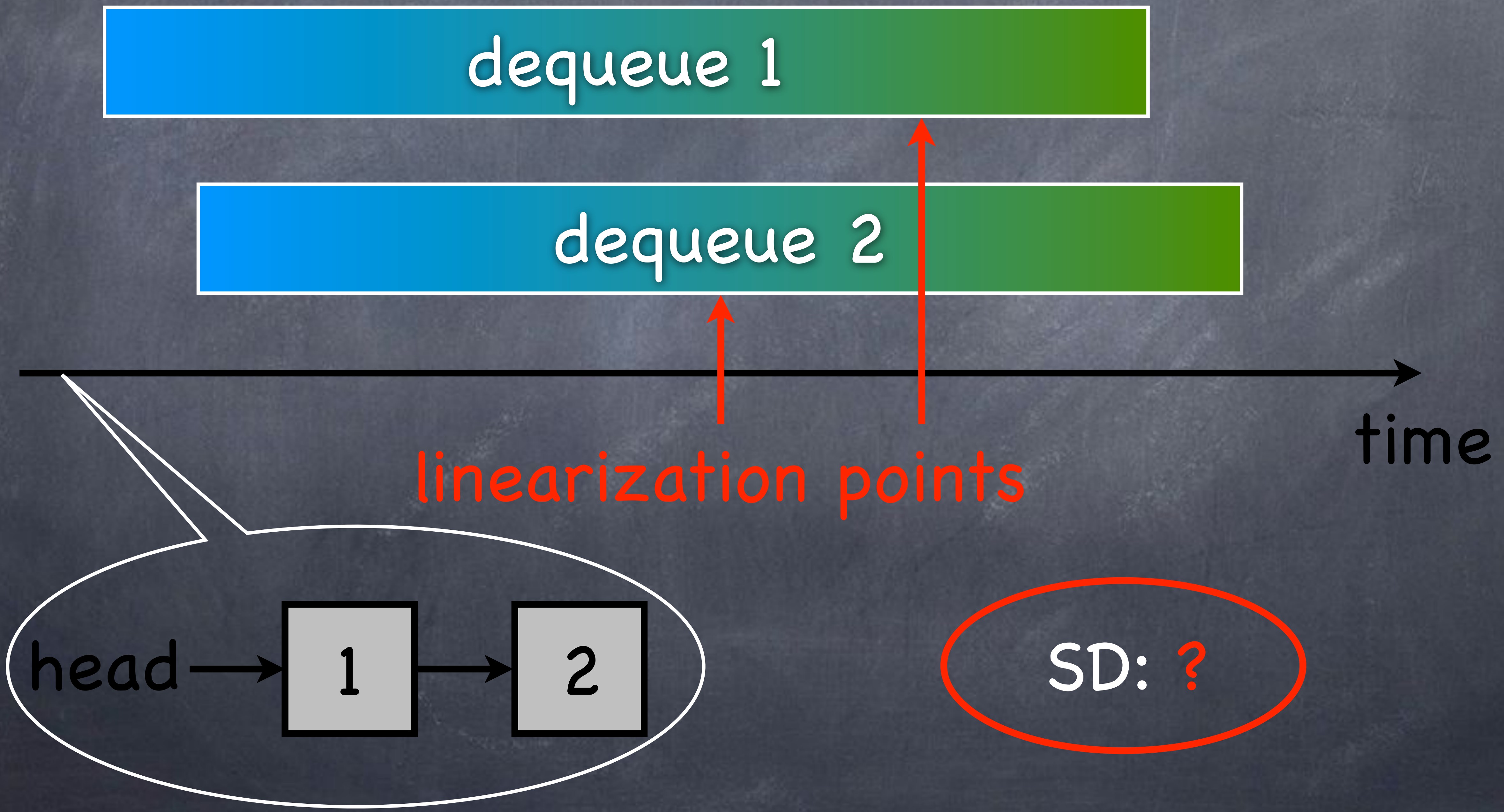
Measuring Semantical Deviation (SD)



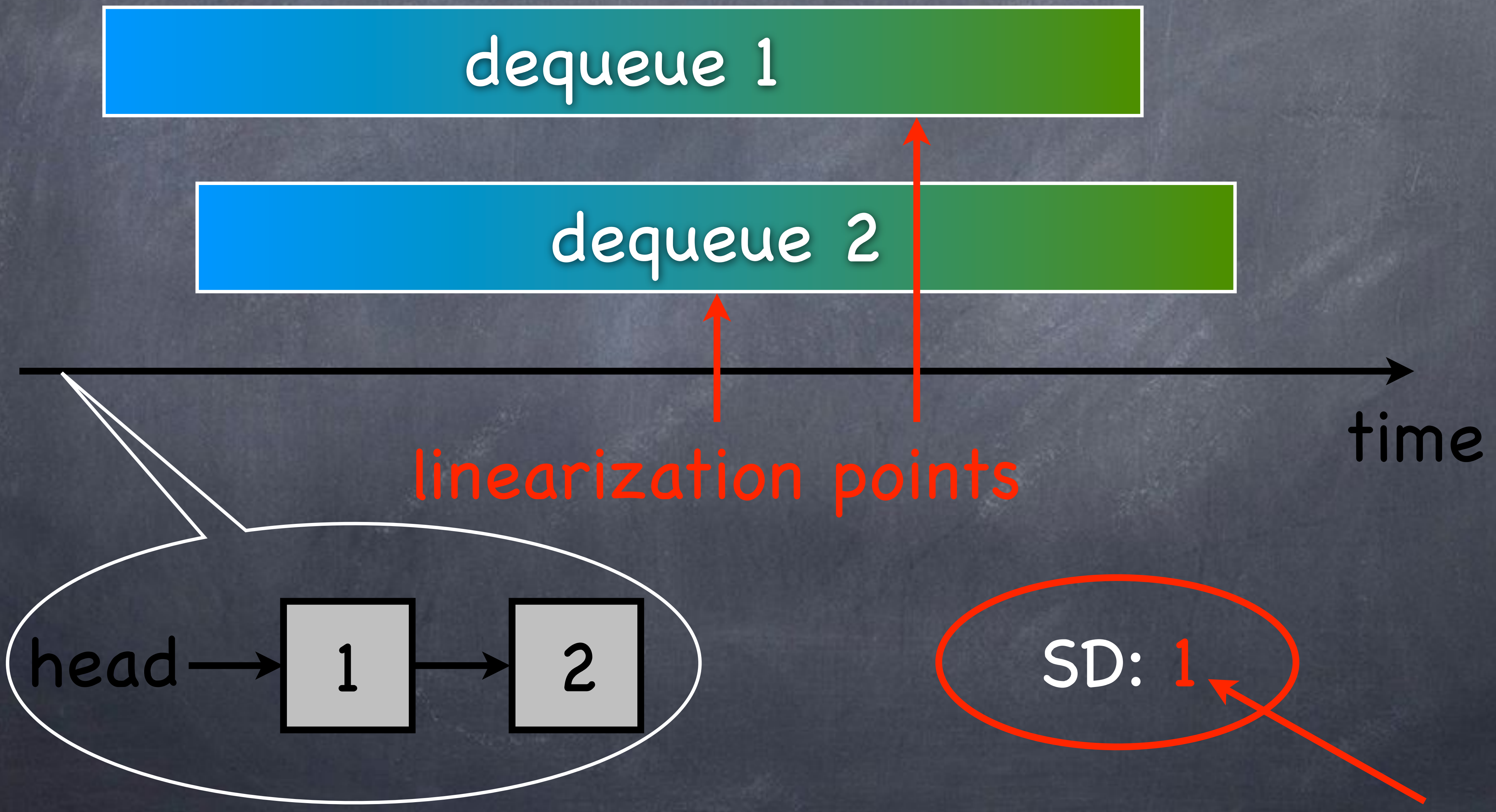
Measuring Semantical Deviation (SD)



Measuring Semantical Deviation (SD)

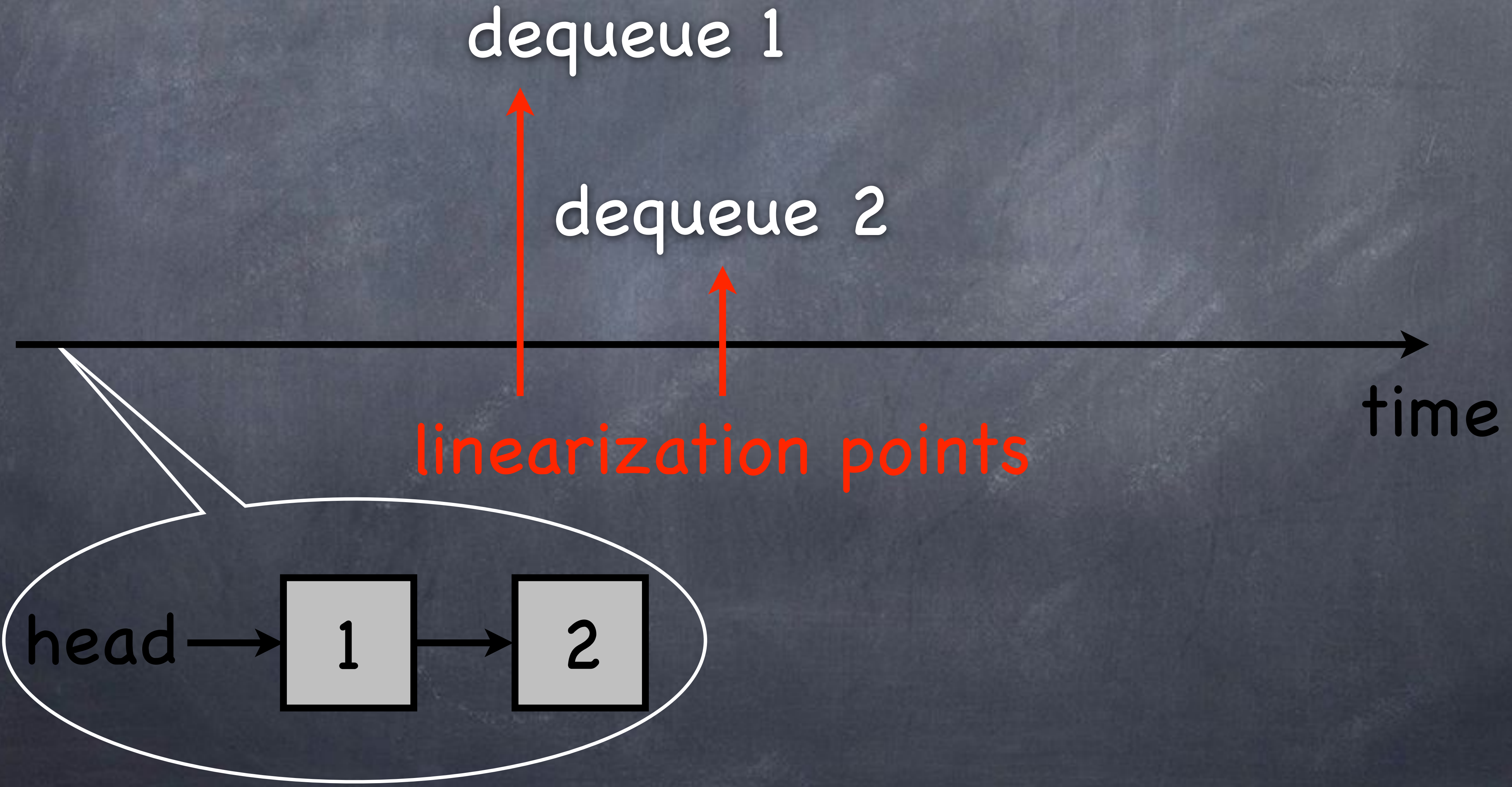


Measuring Semantical Deviation (SD)



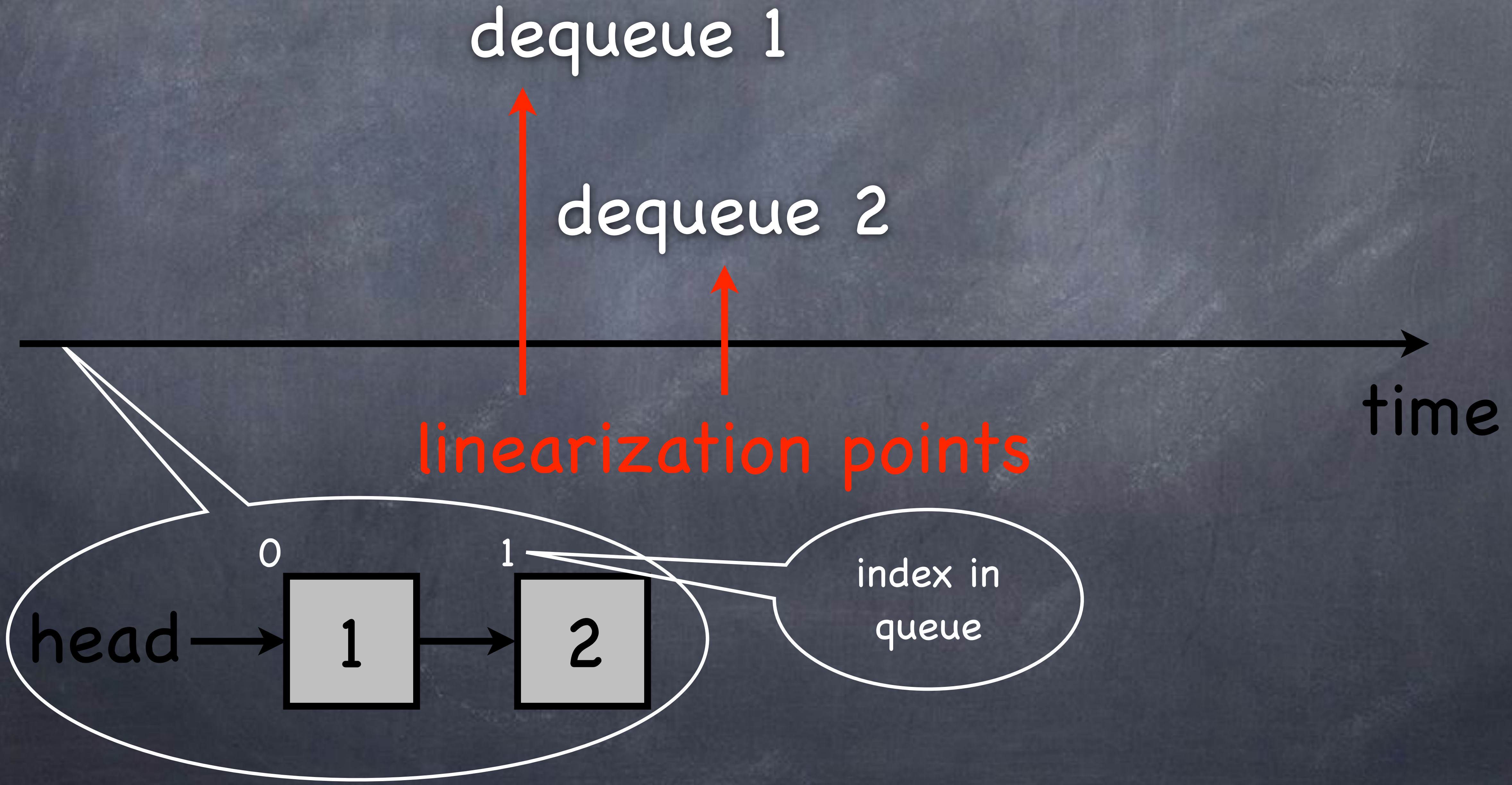
Sequential History

Sequence of Operations (Linearization Points)



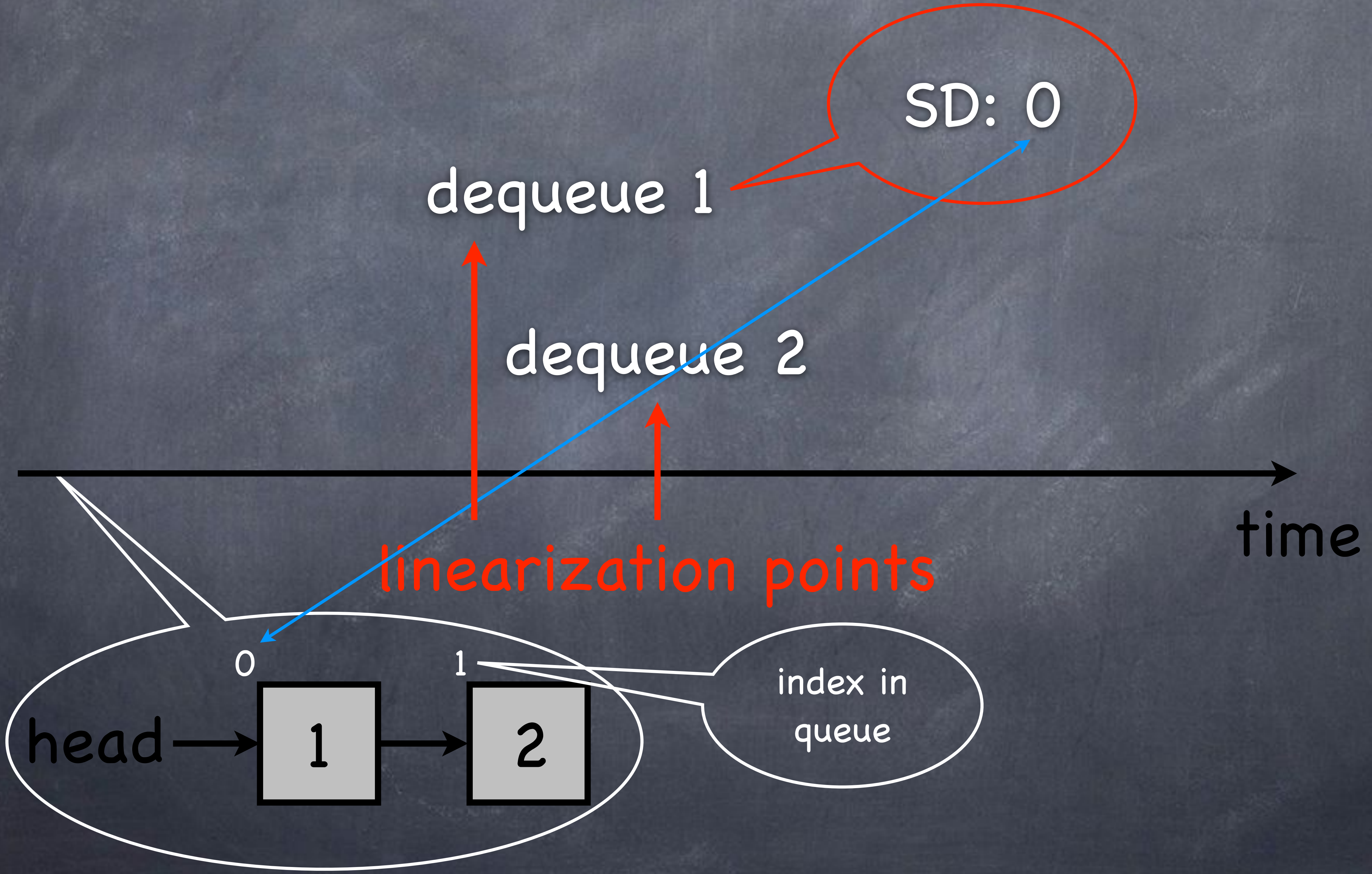
Sequential History

Sequence of Operations (Linearization Points)



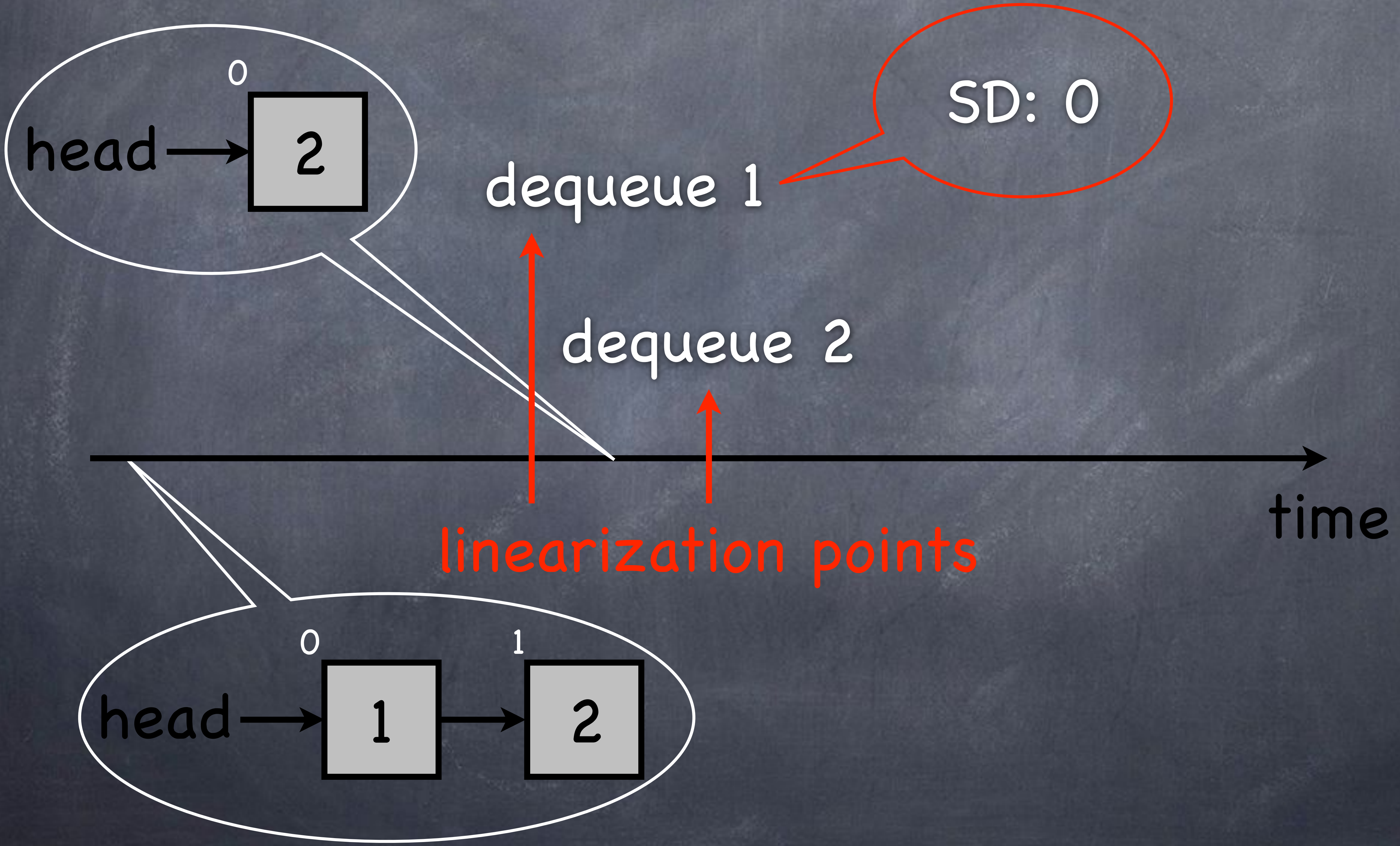
Sequential History

Sequence of Operations (Linearization Points)



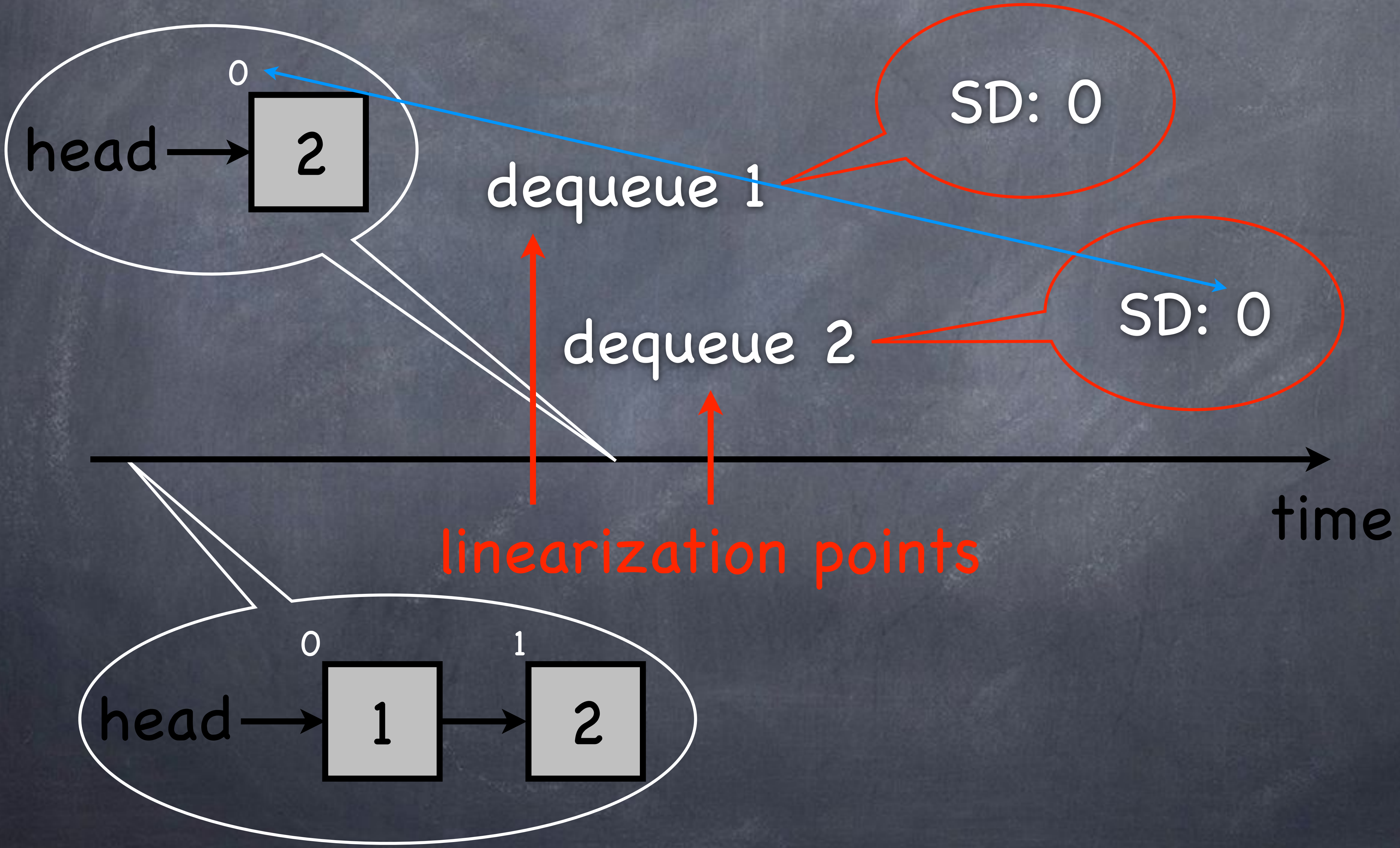
Sequential History

Sequence of Operations (Linearization Points)



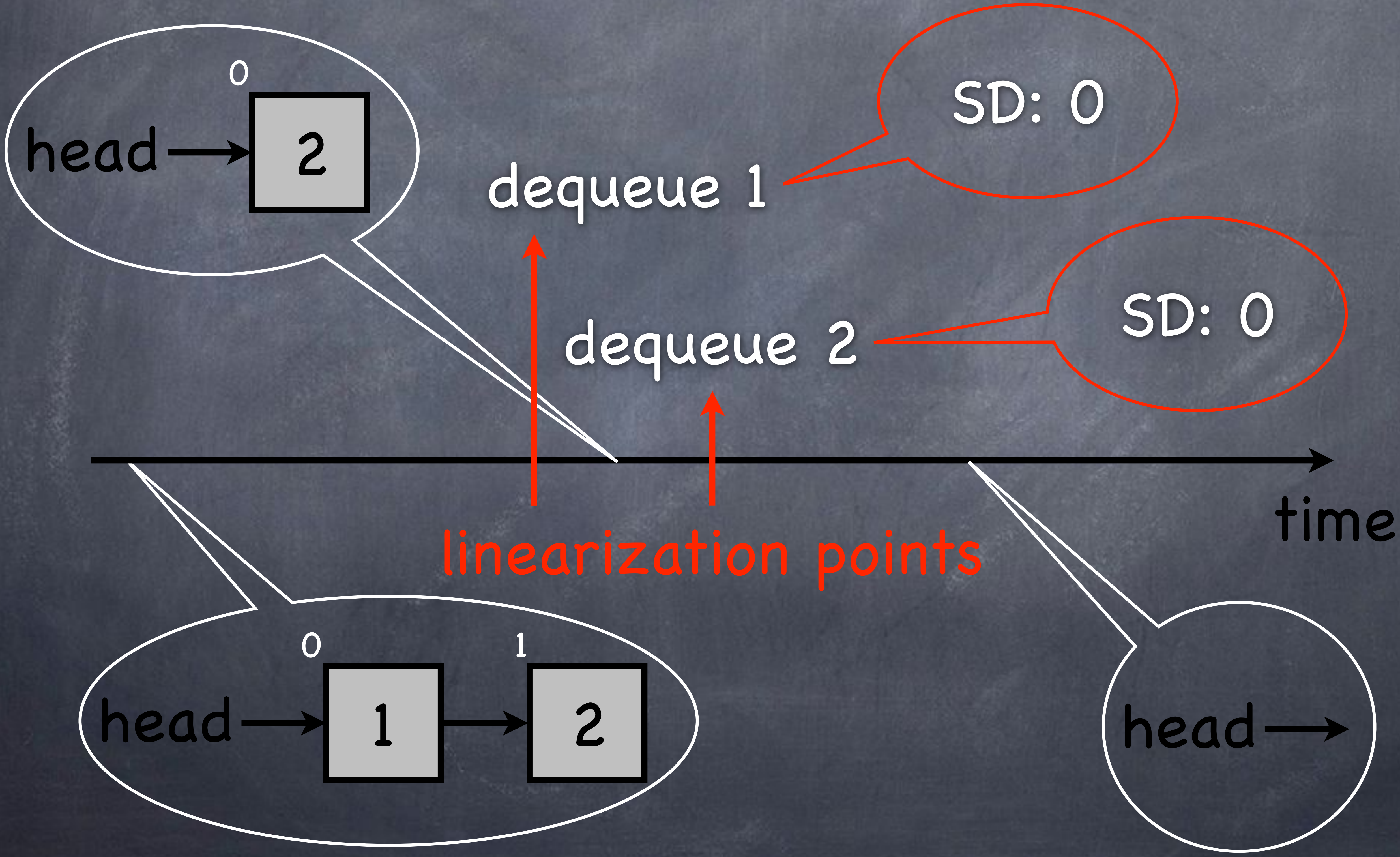
Sequential History

Sequence of Operations (Linearization Points)



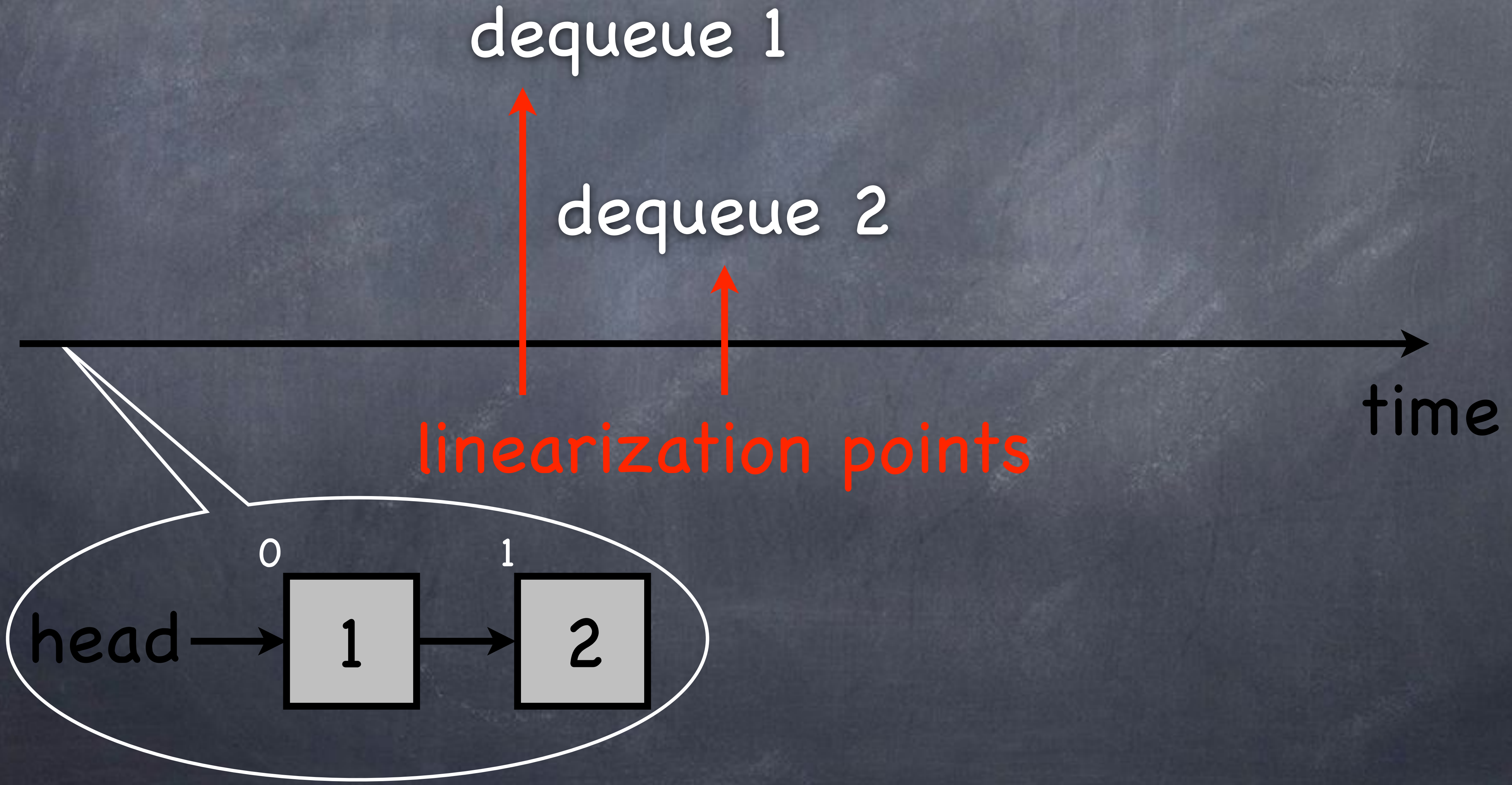
Sequential History

Sequence of Operations (Linearization Points)



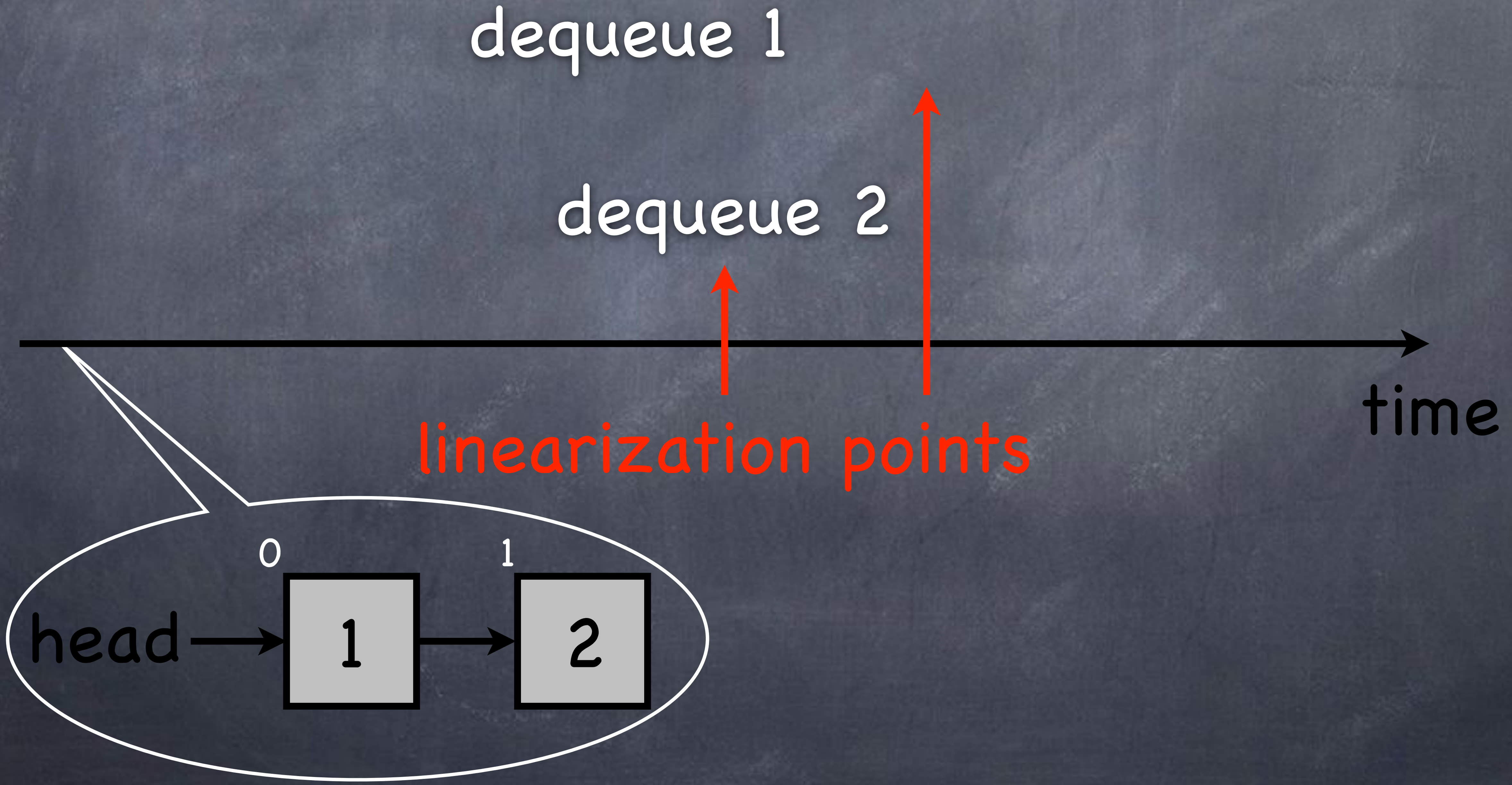
Sequential History II

Sequence of Operations (Linearization Points)



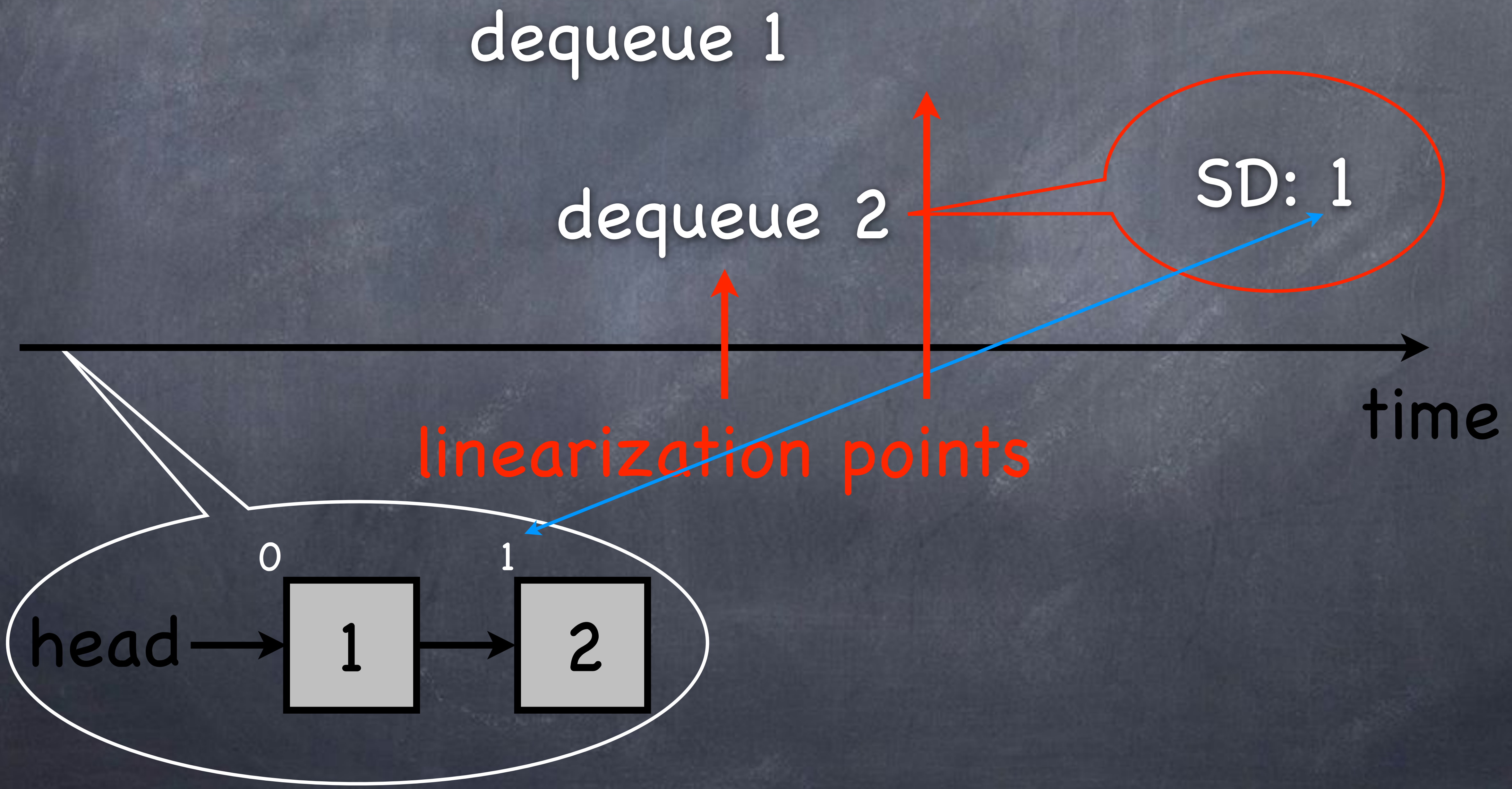
Sequential History II

Sequence of Operations (Linearization Points)



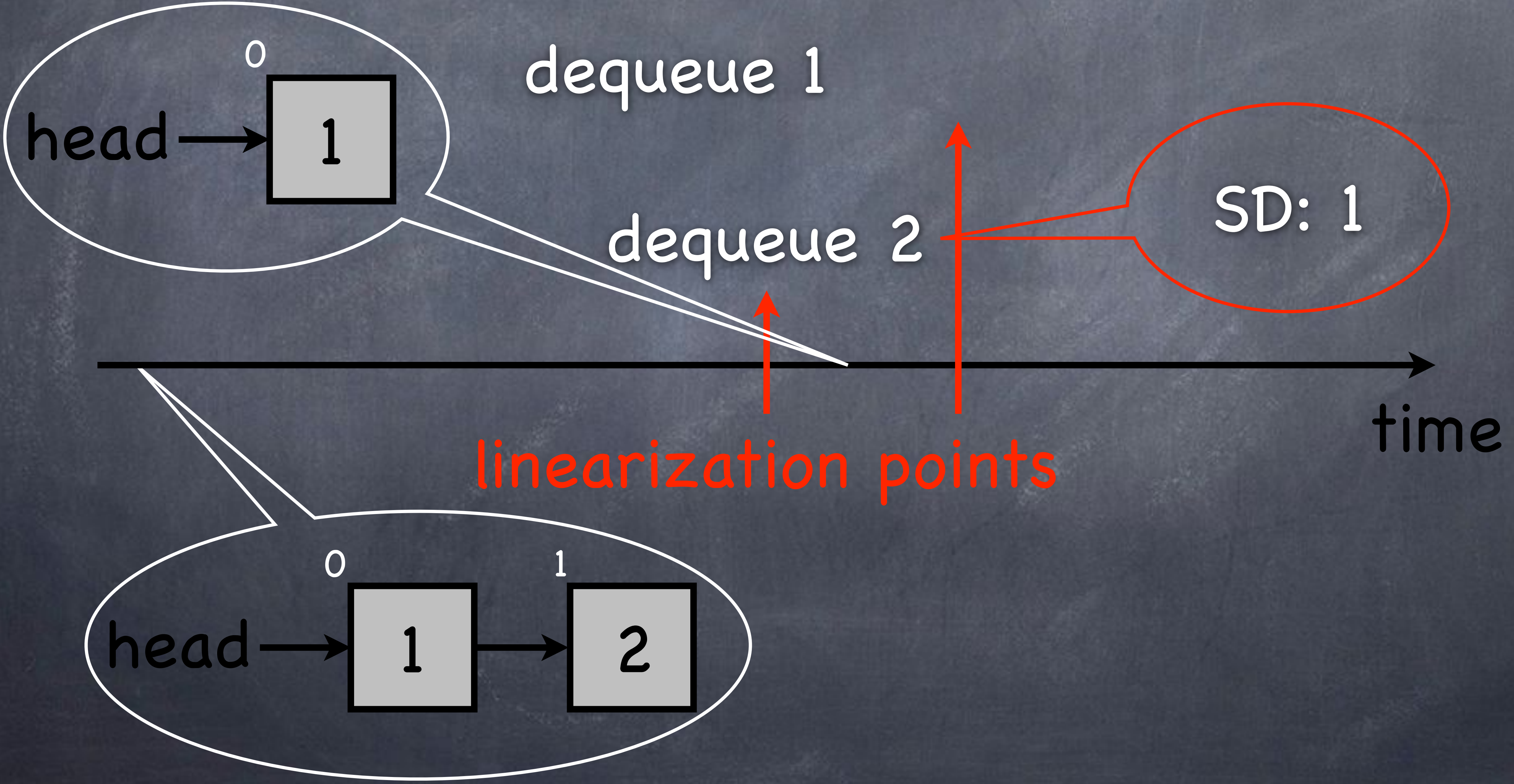
Sequential History II

Sequence of Operations (Linearization Points)



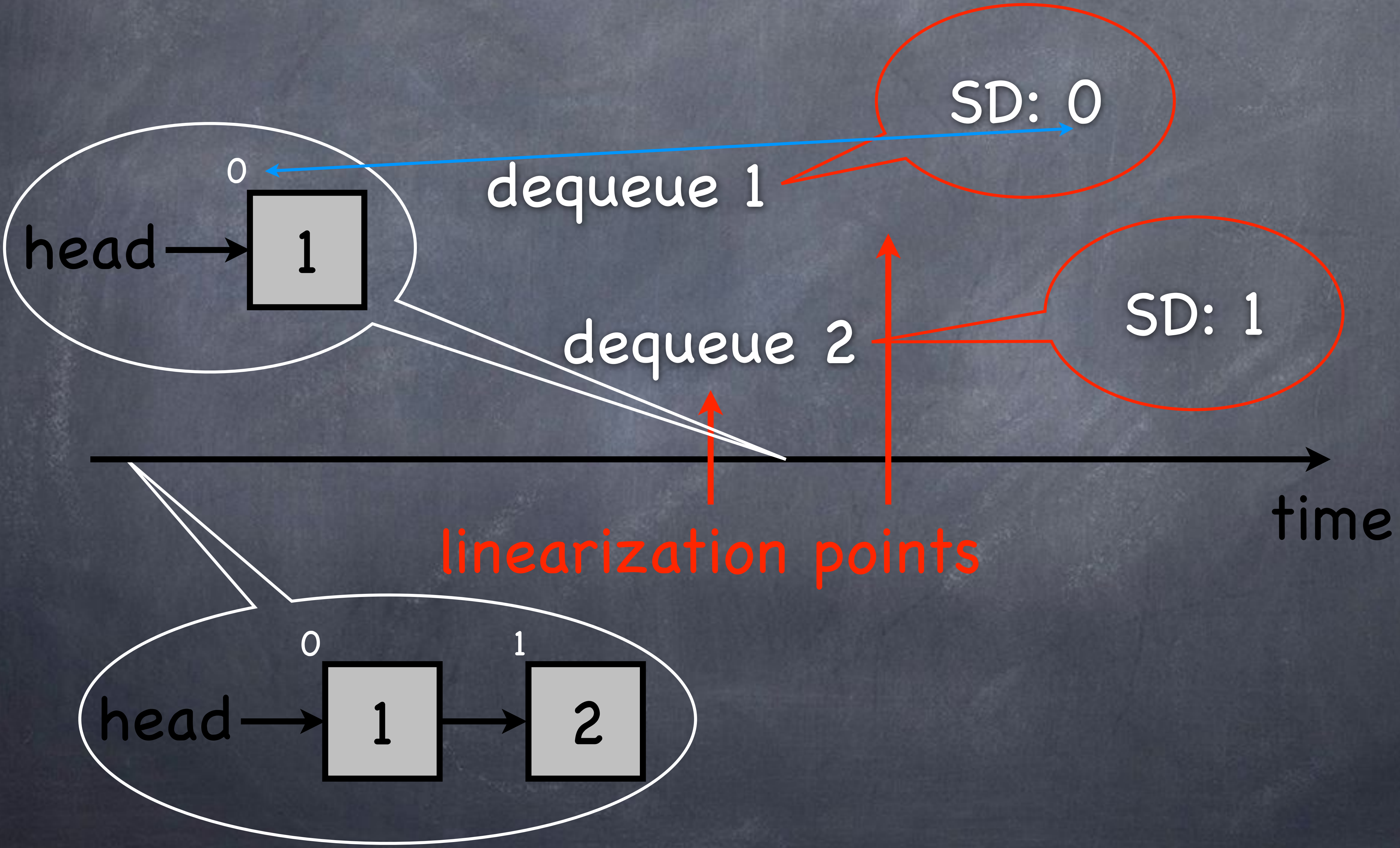
Sequential History II

Sequence of Operations (Linearization Points)



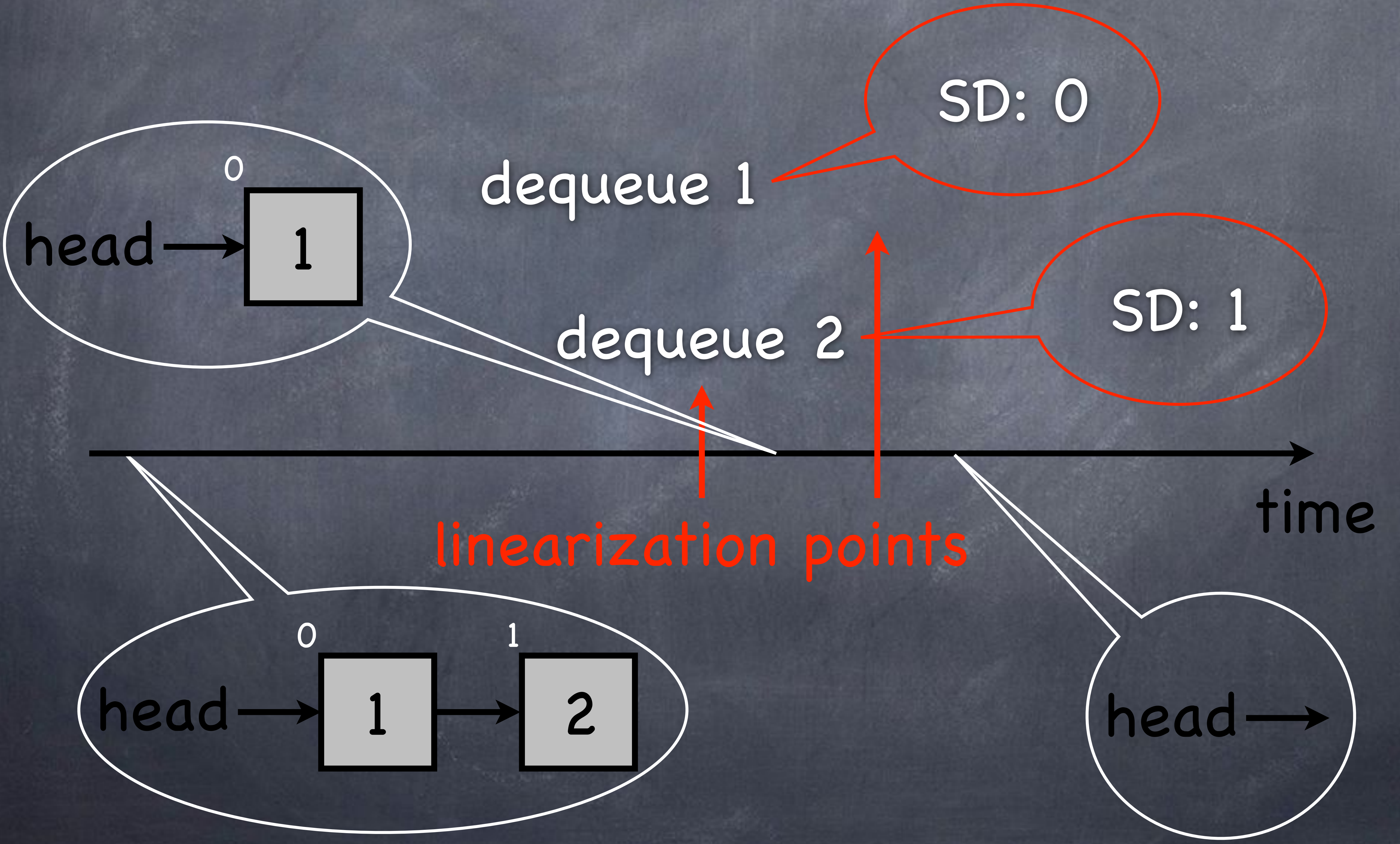
Sequential History II

Sequence of Operations (Linearization Points)

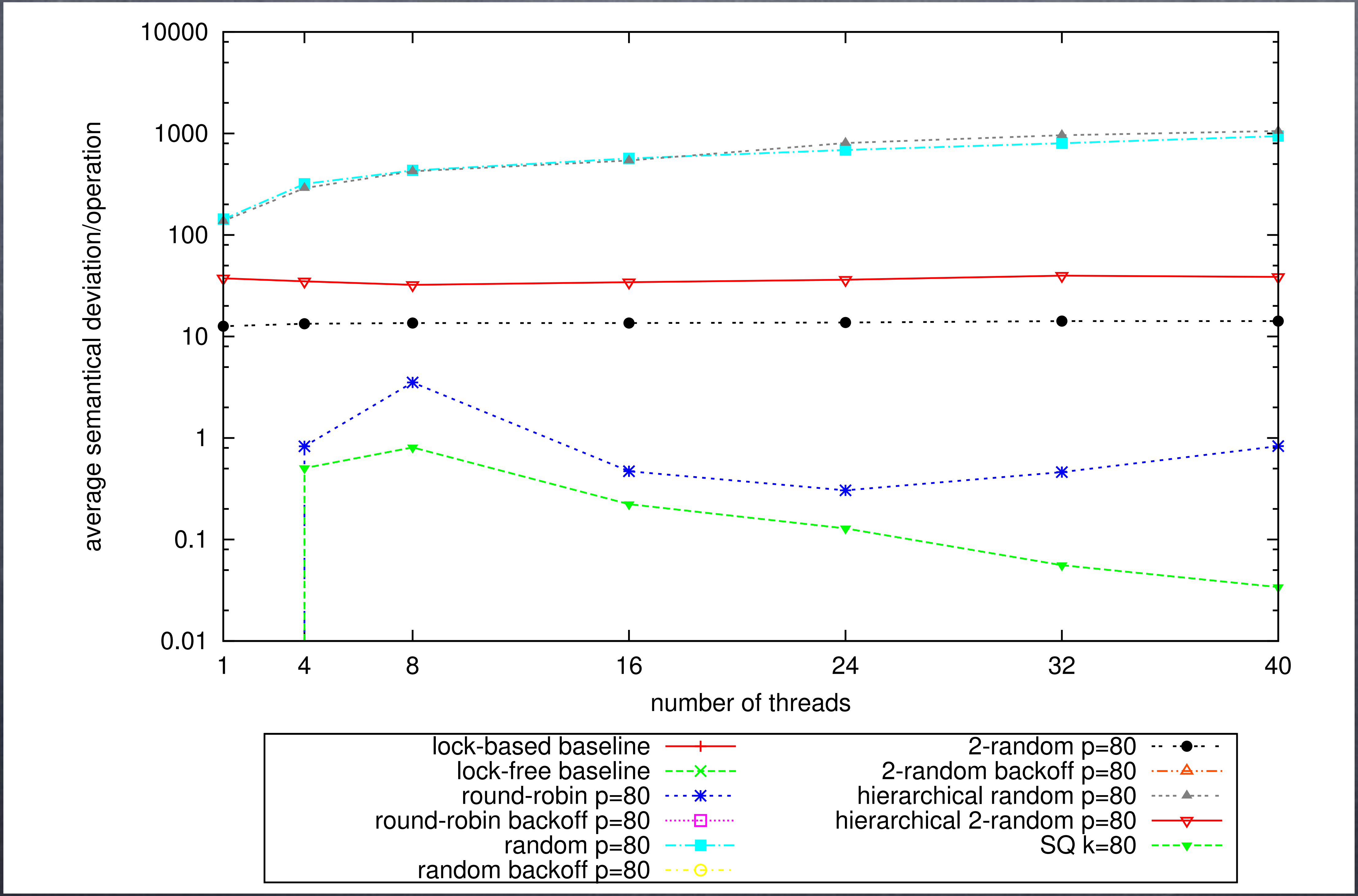


Sequential History II

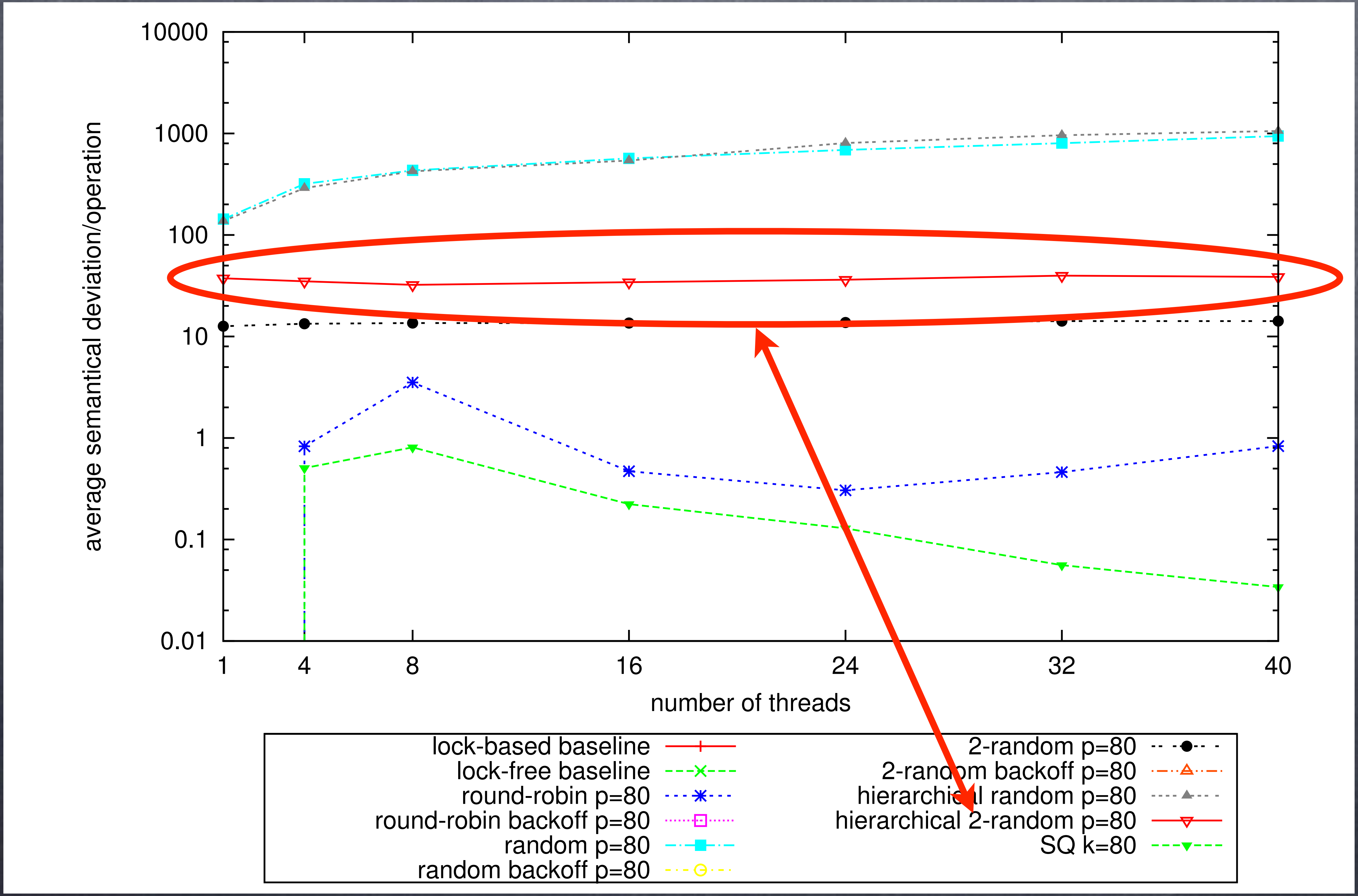
Sequence of Operations (Linearization Points)



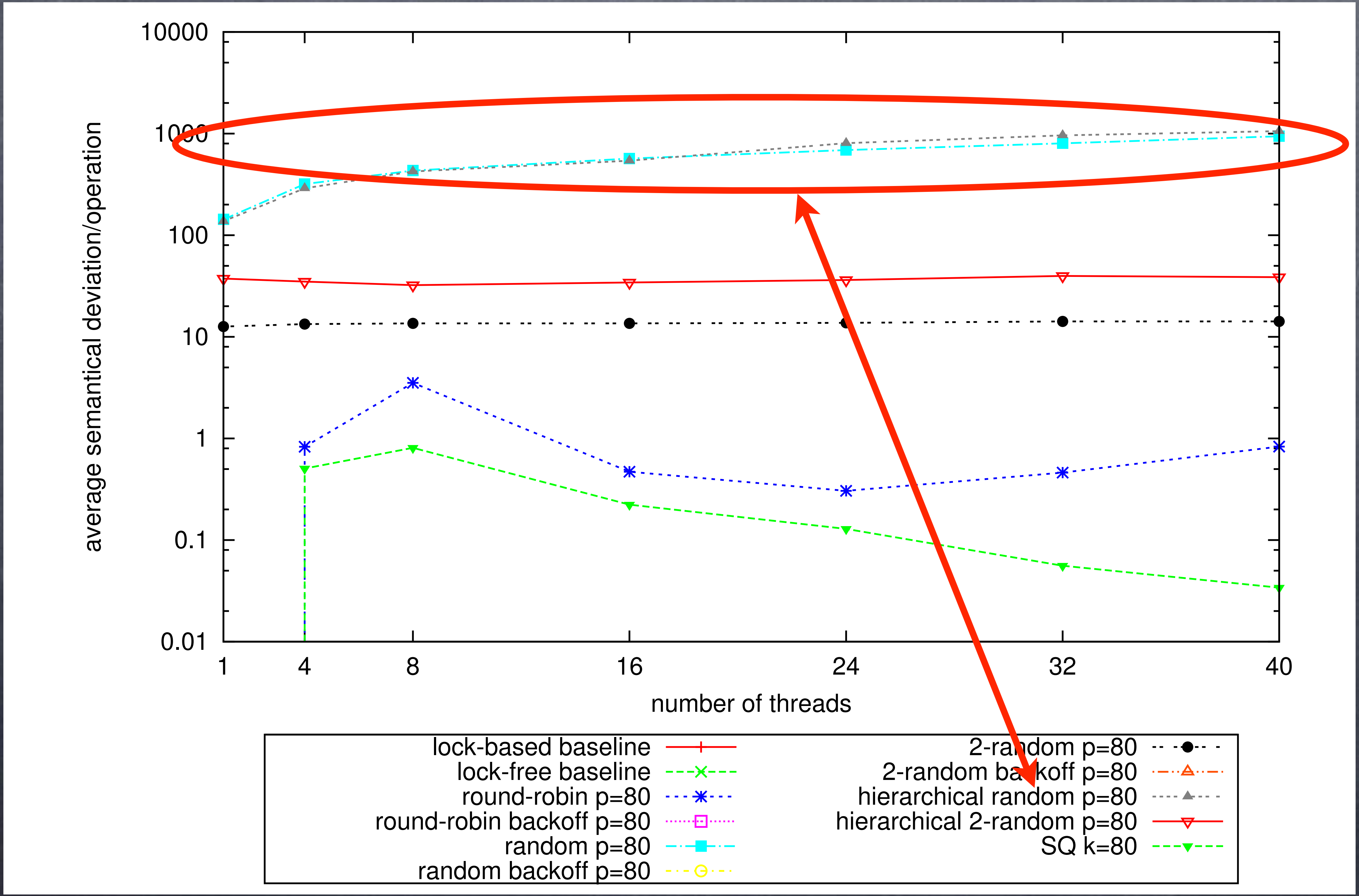
Average Semantical Deviation



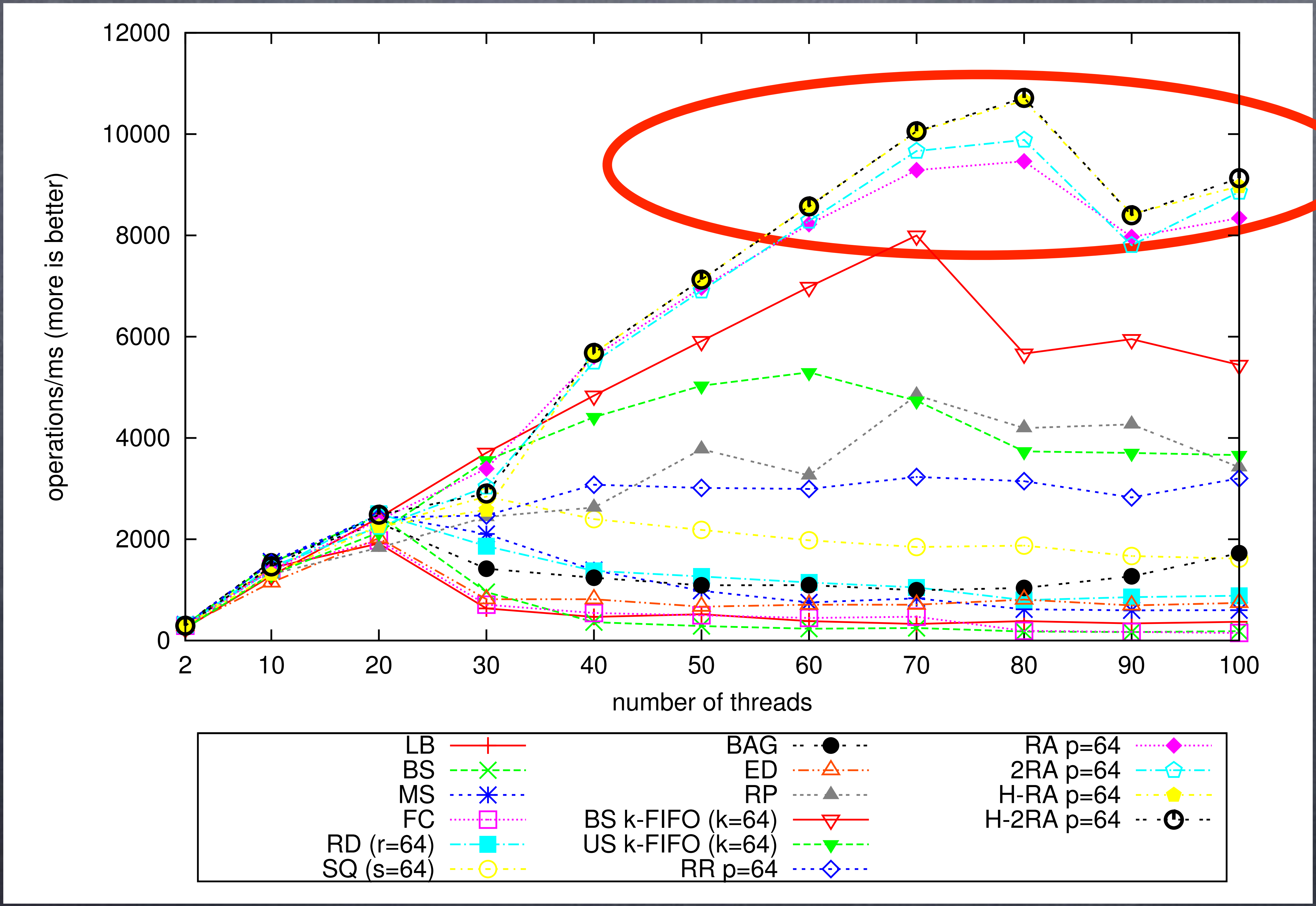
Here k may be around 40 on average: best tradeoff



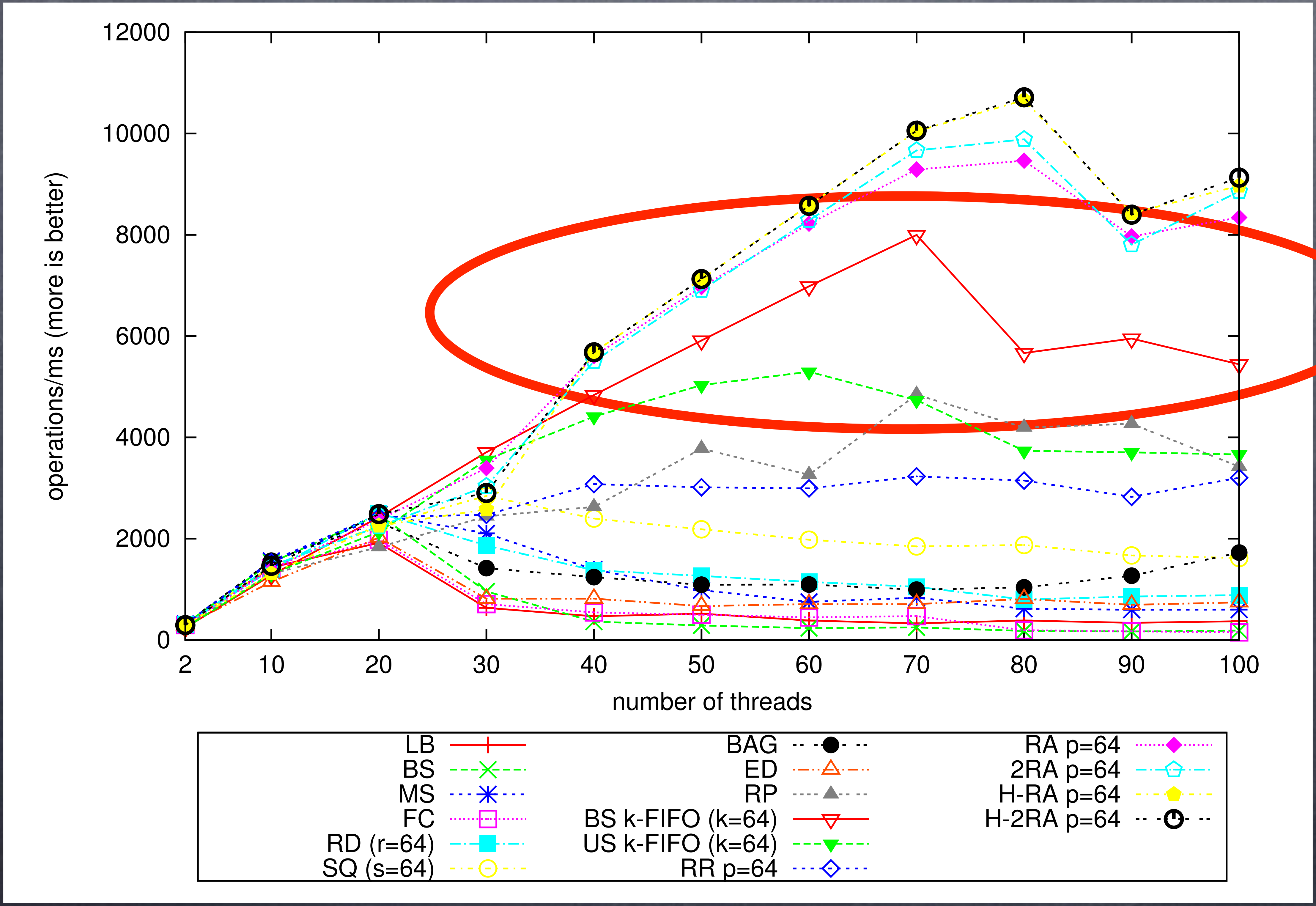
Whereas here k is one order of magnitude bigger w/o gain



Random vs. d-Random



Segmented Queues



Back to Correctness?

Questions?