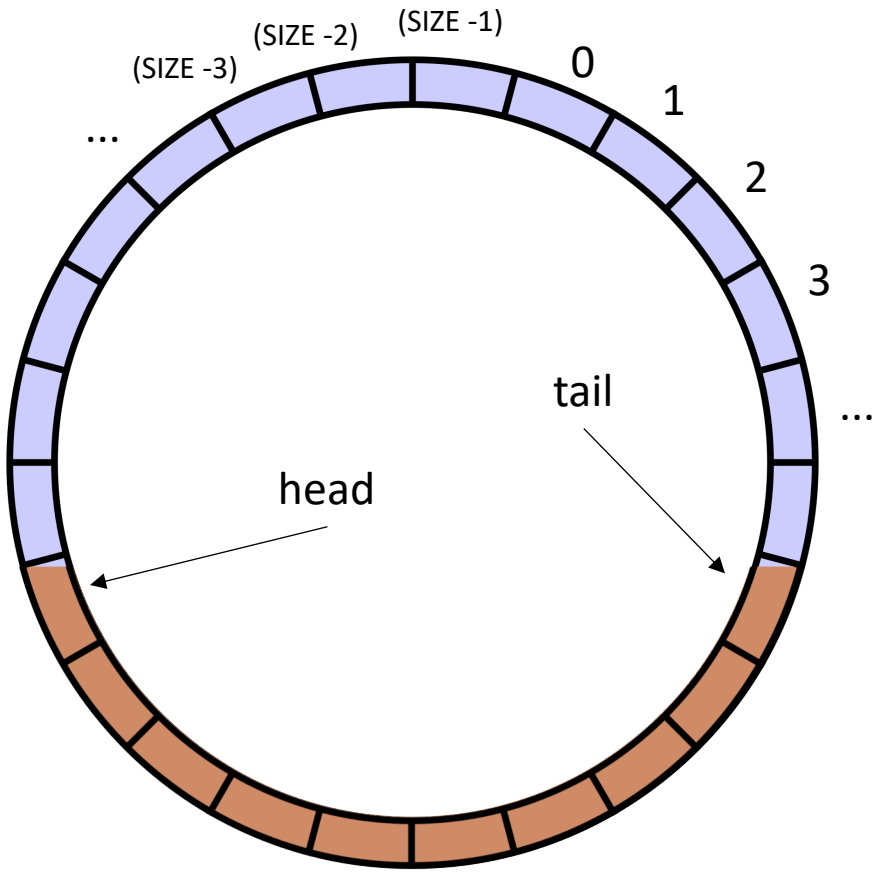
CSE113: Parallel Programming

May 6, 2021

- **Topic**: Concurrent Objects 4
  - Finishing up linked list
  - Reading/Writing Queues
  - Synchronous Producer Consumer
  - Async Producer Consumer



Announcements

- Busy Day (at least at midnight)!
  - HW2 due today!
  - Midterm due today!
  - HW3 released today!
  - HW1 Grades released today!
- Gan had office hours this morning
- Reese will have them after class

Announcements

- Erica is running a study on parallel programming:
  - Sign up if you are interested!

- We won't finish module 3 today:
  - Next week we'll discuss work stealing
  - I will still plan to release the HW today

- May 20 will be guest lecture:
  - Hugues Evrard will discuss message-passing concurrency
  - Alistair Donaldson will discuss testing GPU compilers

Quiz

Quiz

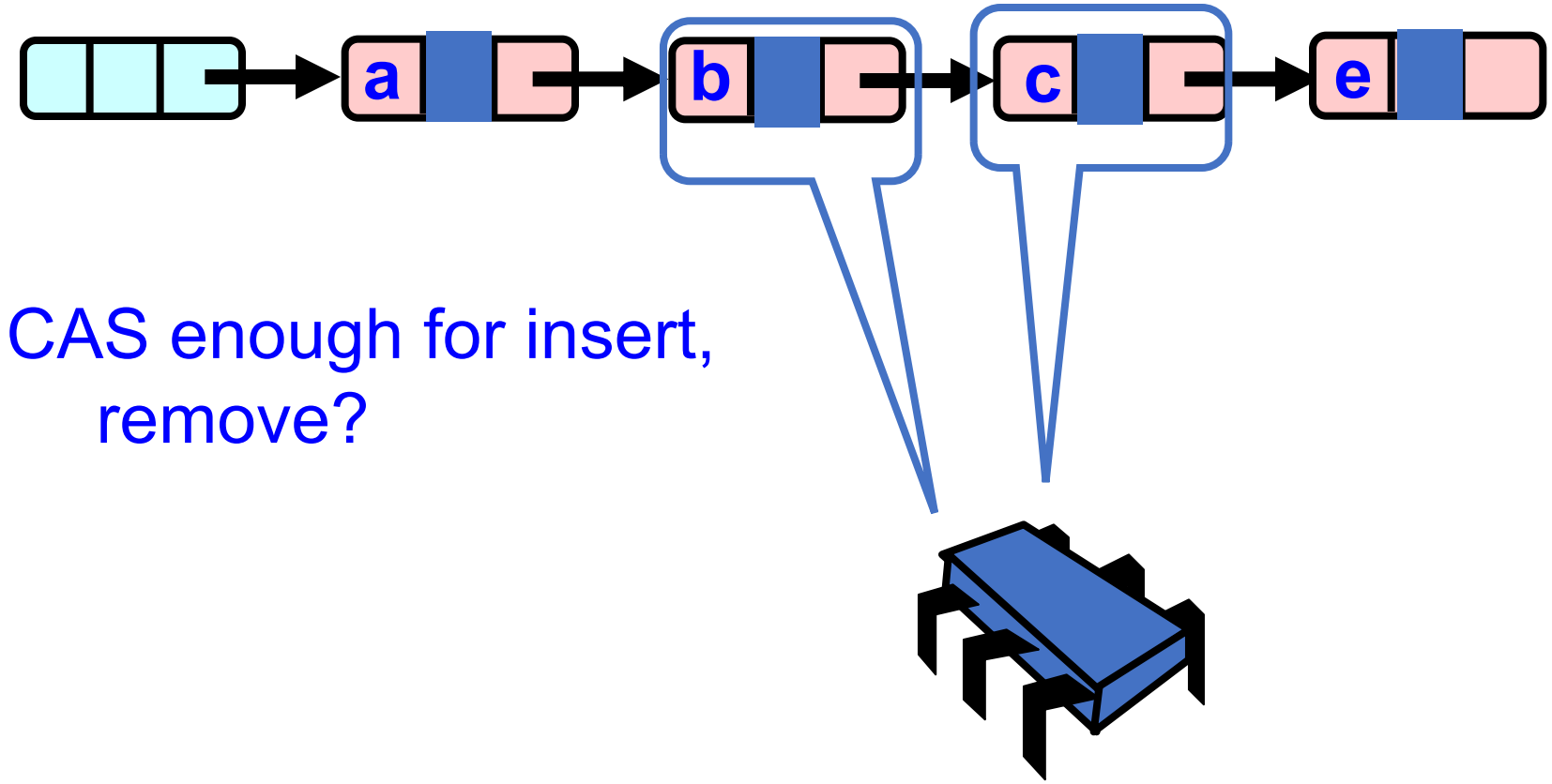
- Discuss Answers

Schedule

- **Finish up linked-list**

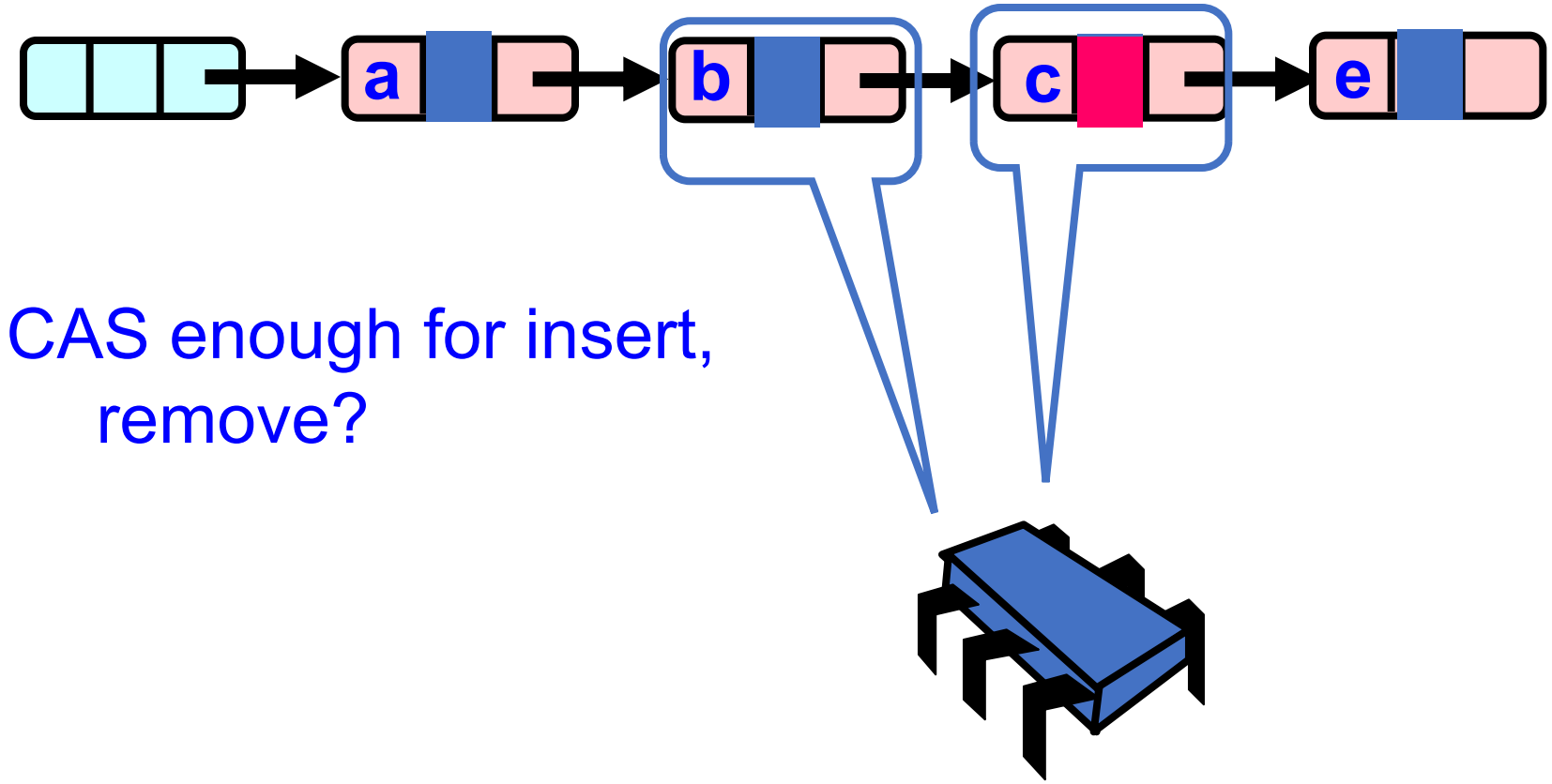
- Concurrent Queues
  - Input/Output Queues
  - Synchronous Producer/Consumer Queue
  - Async Producer/Consumer Queue

Lock-free Lists



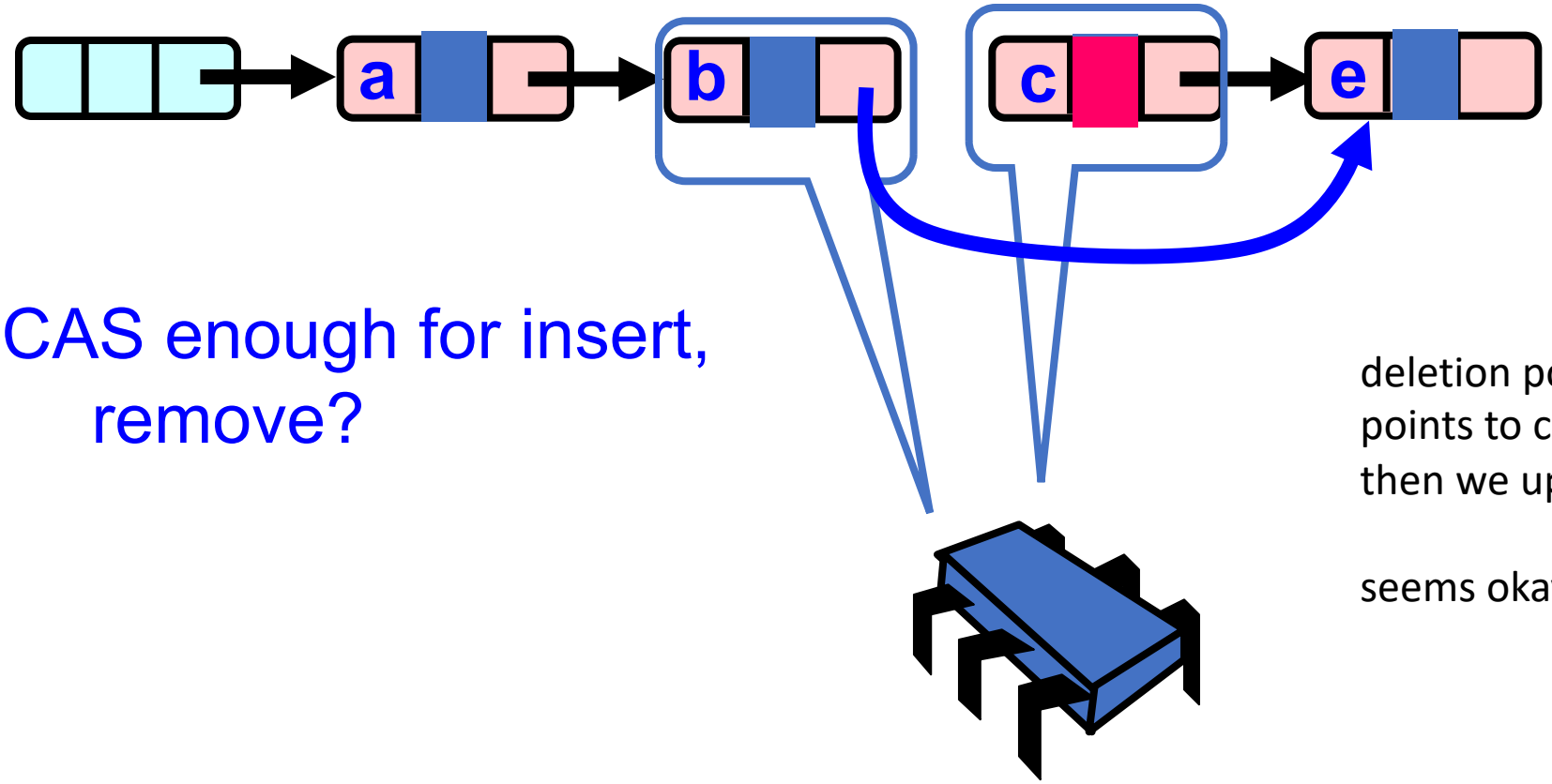
CAS enough for insert, remove?

Lock-free Lists



CAS enough for insert, remove?

Lock-free Lists



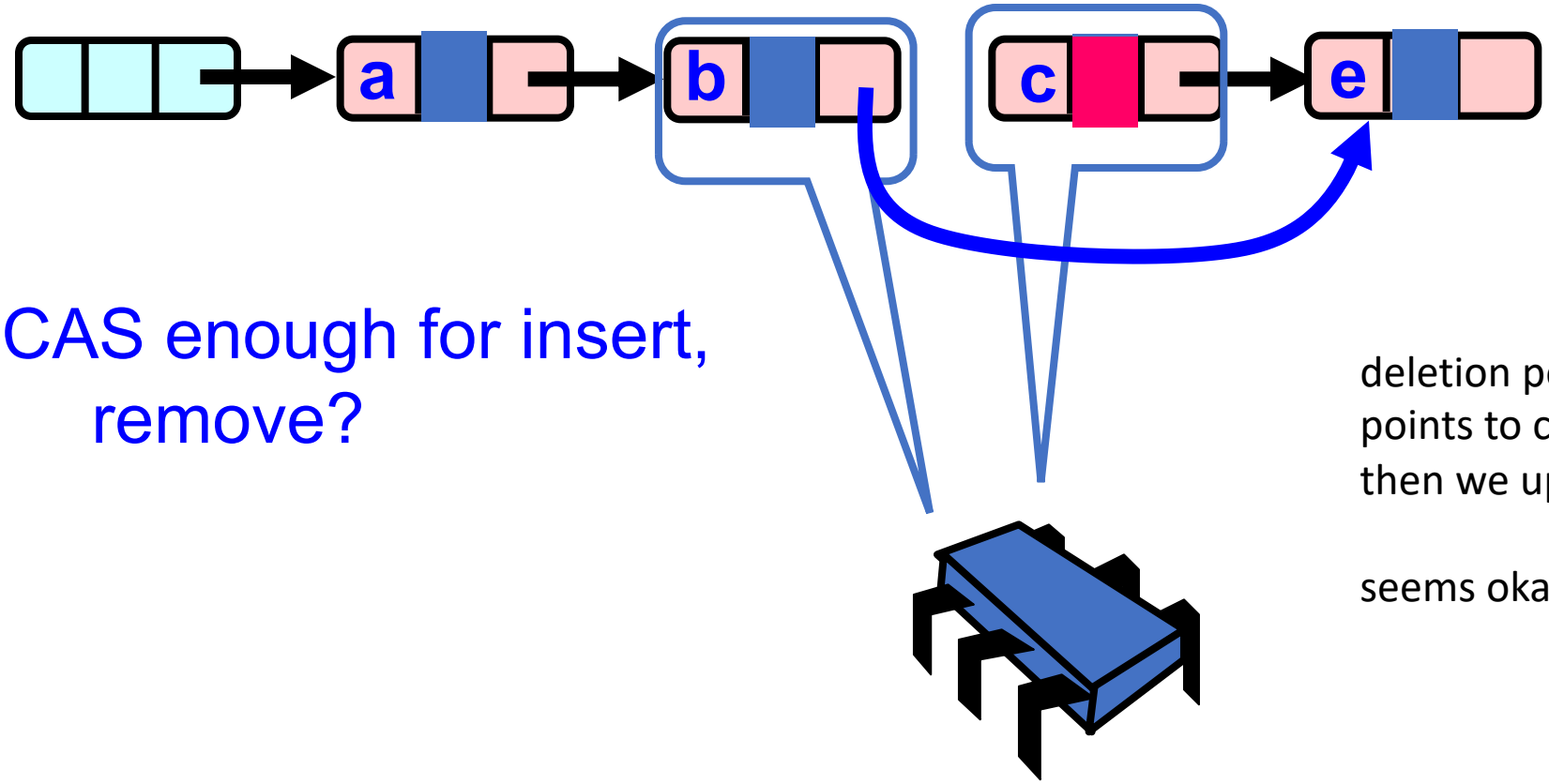
CAS enough for insert, remove?

deletion point requires b points to c. If that is valid then we update to e.

seems okay...

Lock-free Lists

ensures that nobody has inserted a node between b and c



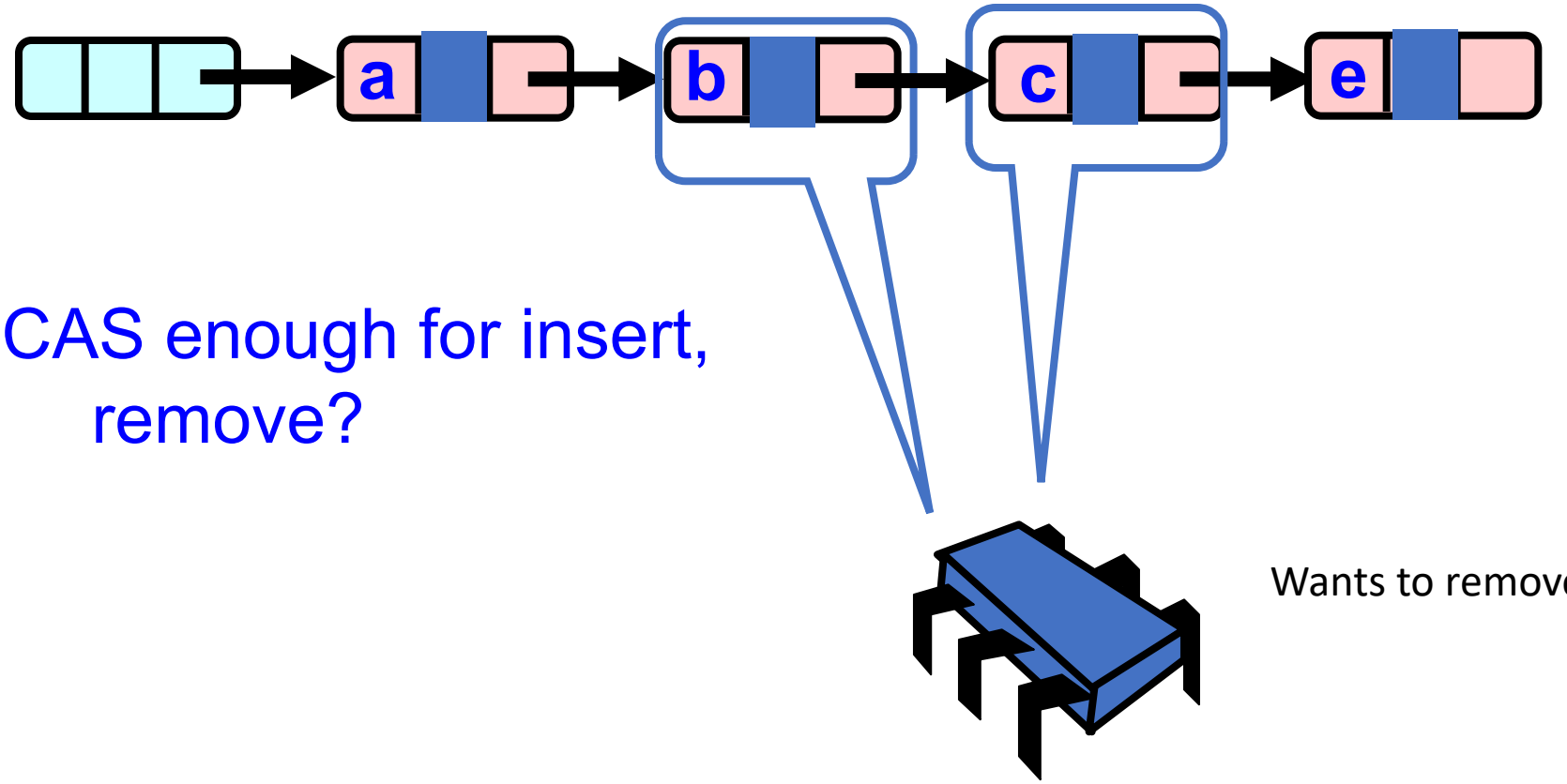
CAS enough for insert, remove?

deletion point requires b points to c. If that is valid then we update to e.

seems okay...

Lock-free Lists

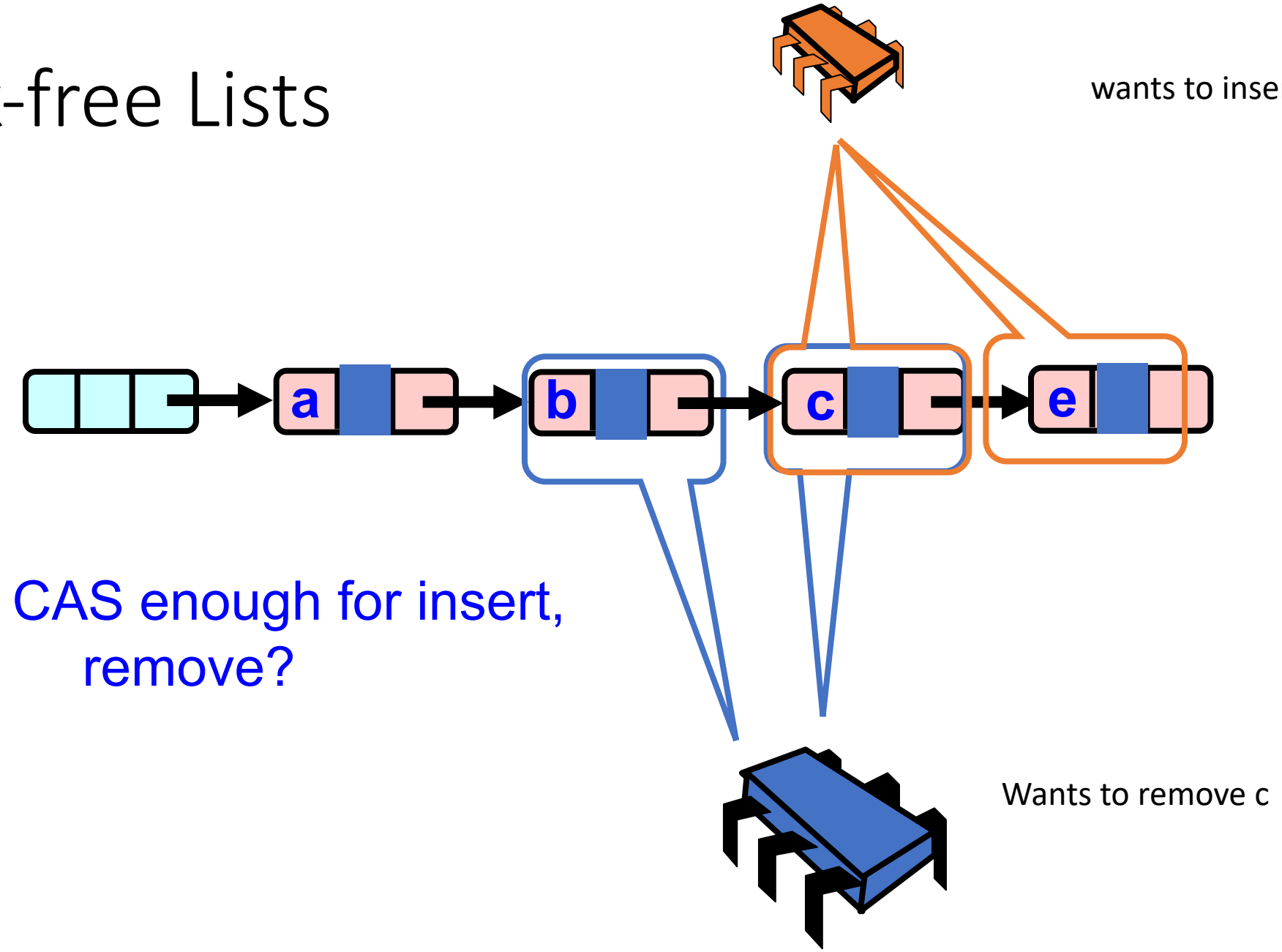
Rewind



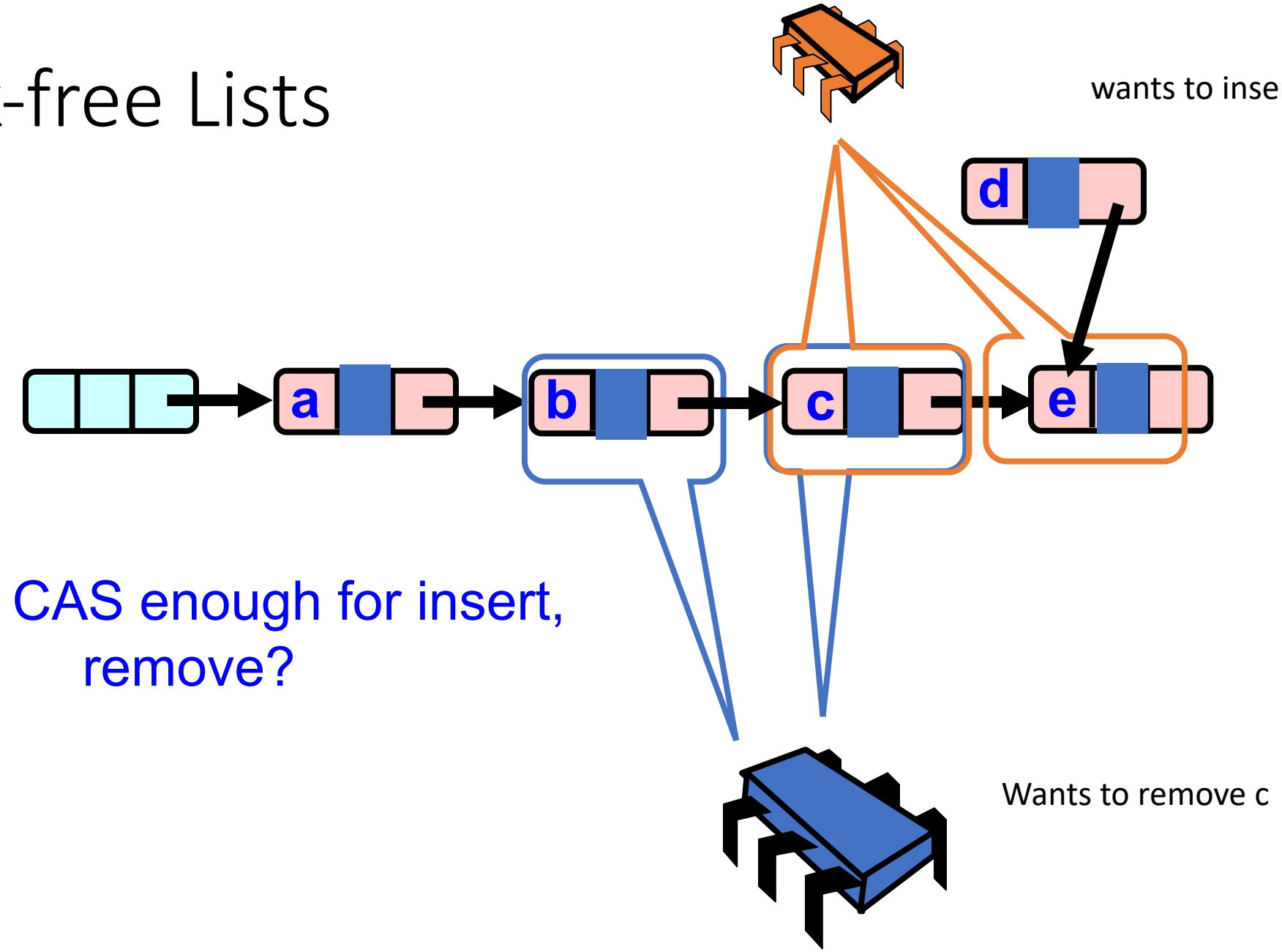
CAS enough for insert, remove?

Wants to remove c

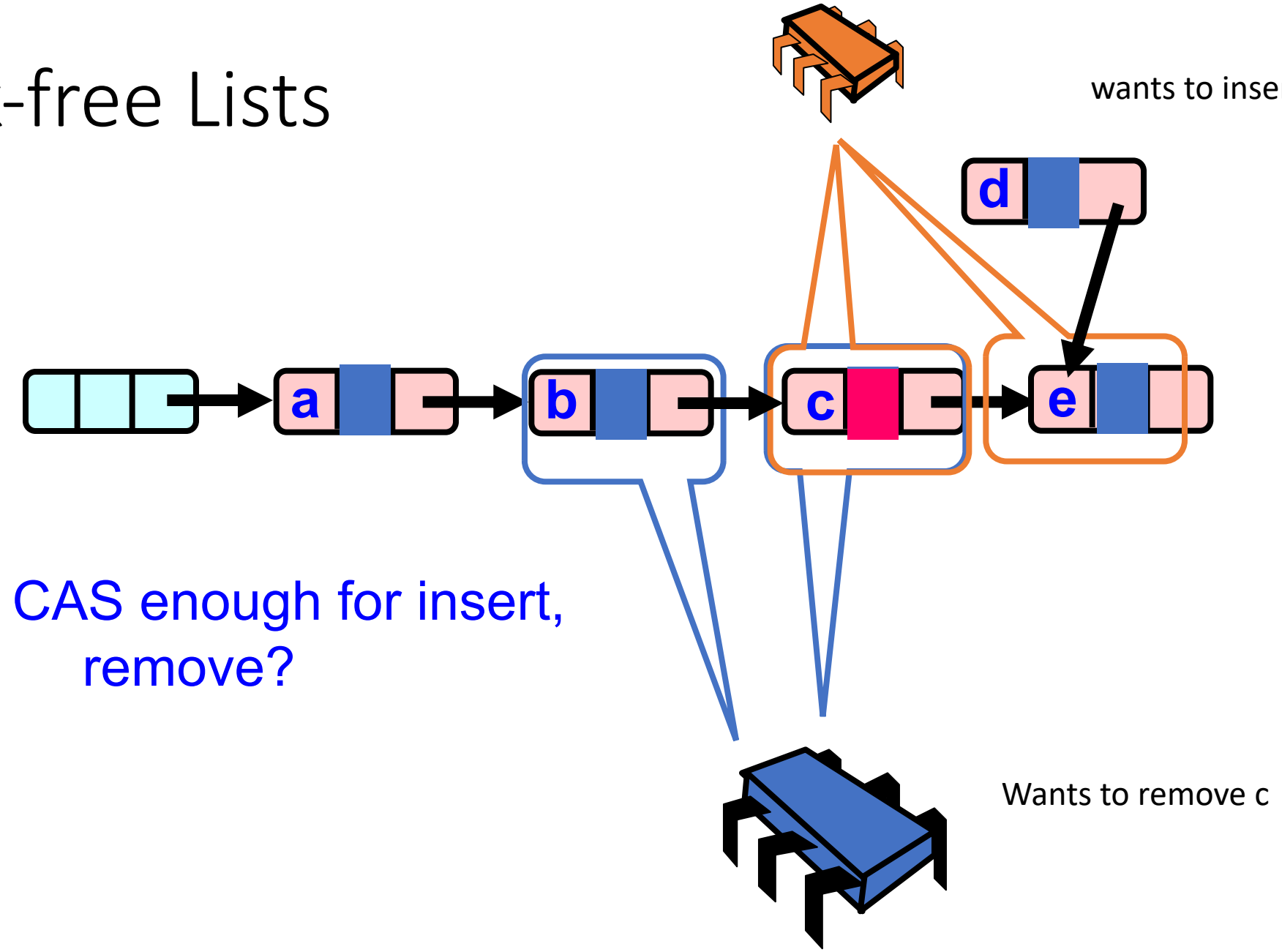
Lock-free Lists



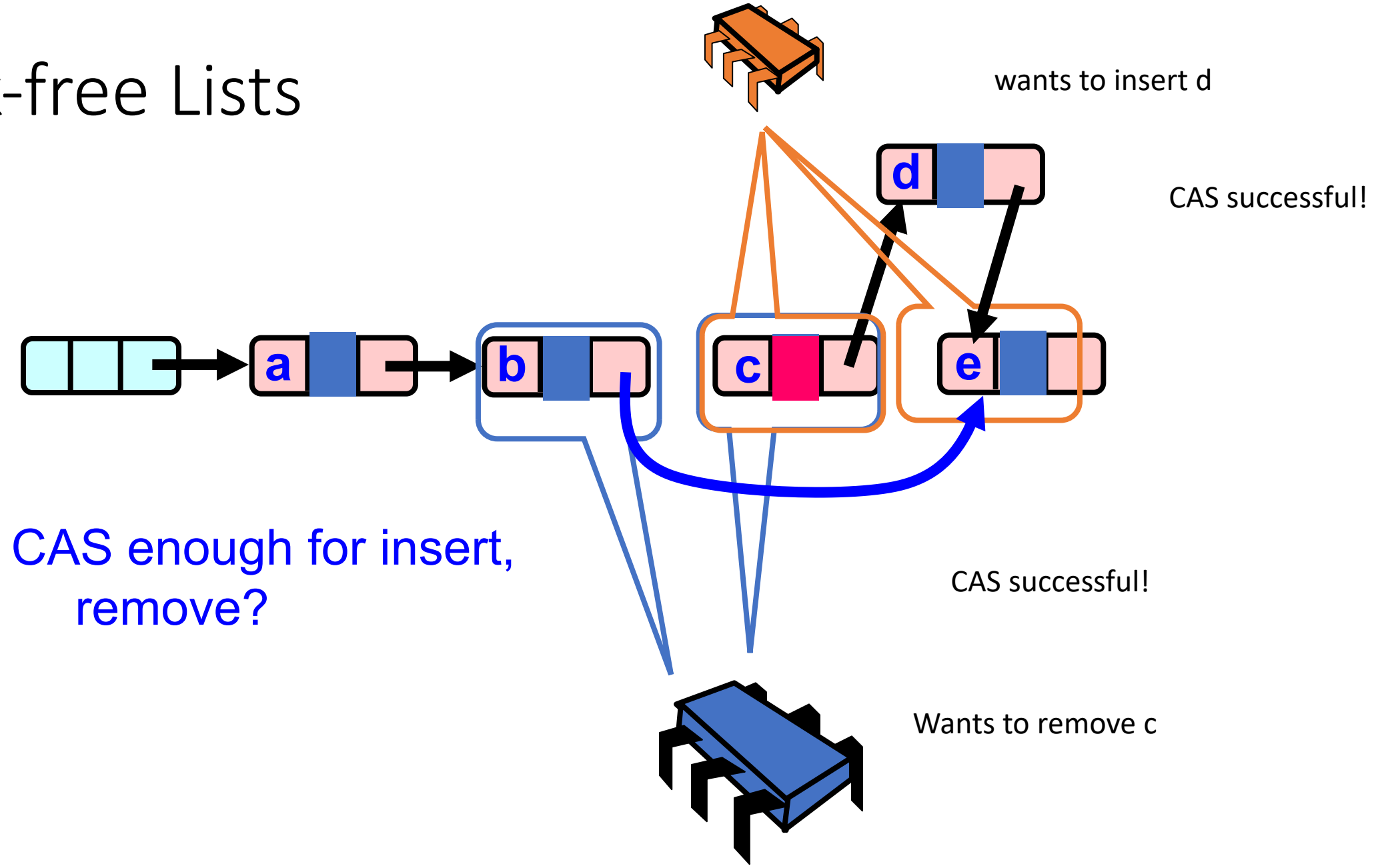
Lock-free Lists



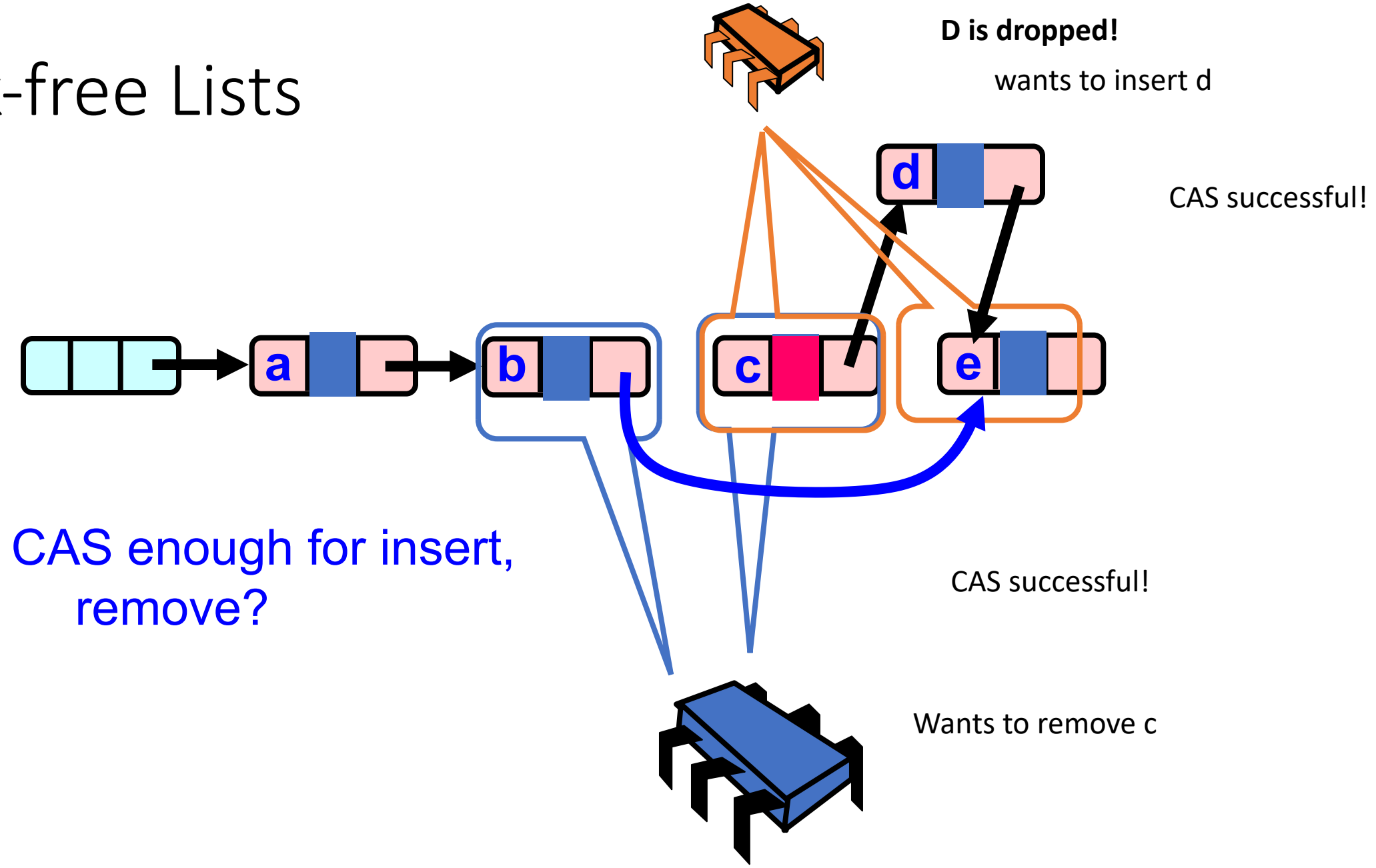
Lock-free Lists



Lock-free Lists



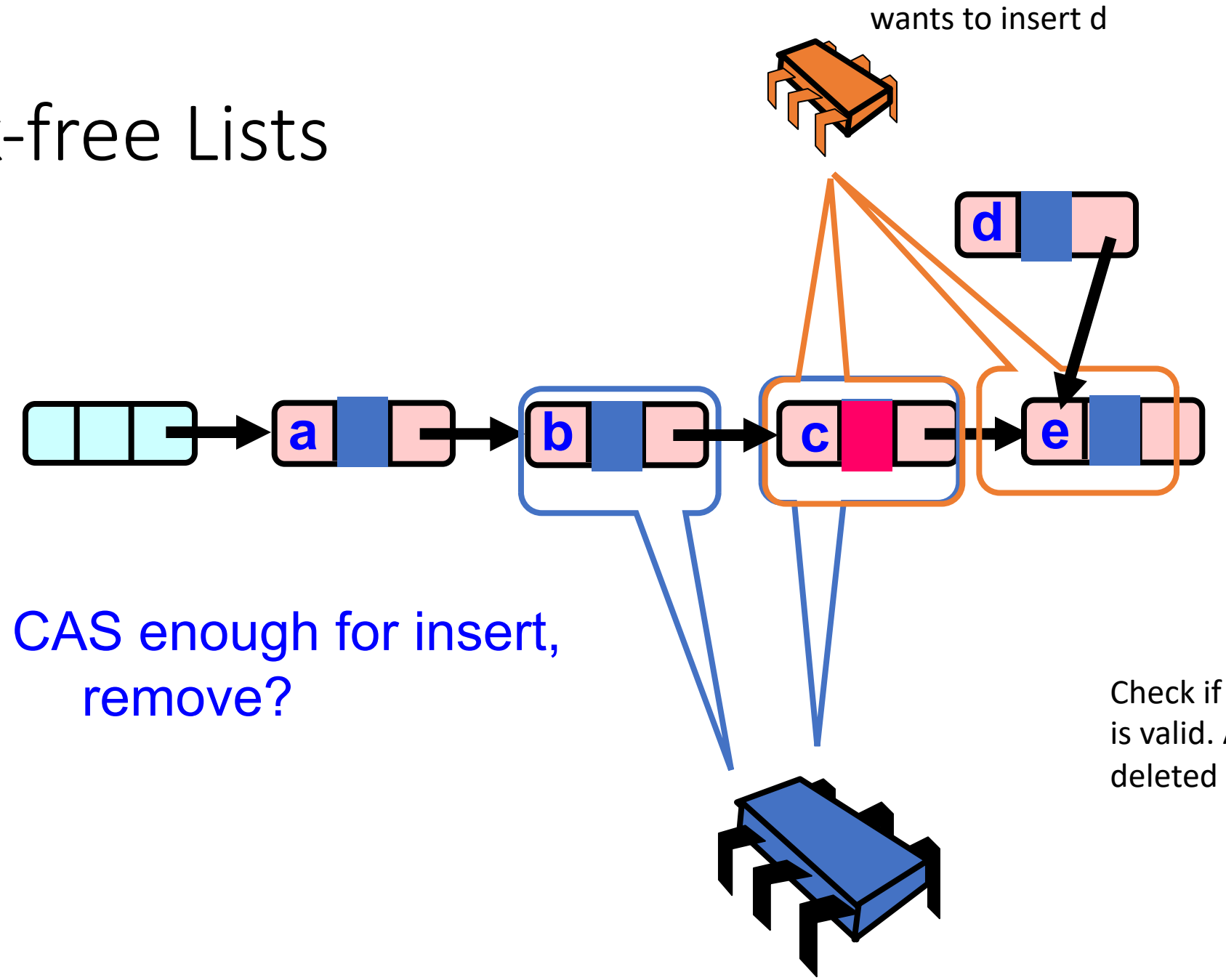
Lock-free Lists



Solution

- Use AtomicMarkableReference
- Atomic CAS that checks not only the address, but also a bit
- We can say: update pointer if the insertion point is valid AND if the node has not been logically removed.

Lock-free Lists



Check if insertion point is valid AND if C is not logically deleted

CAS enough for insert, remove?

Check if insertion point is valid. And B is not logically deleted

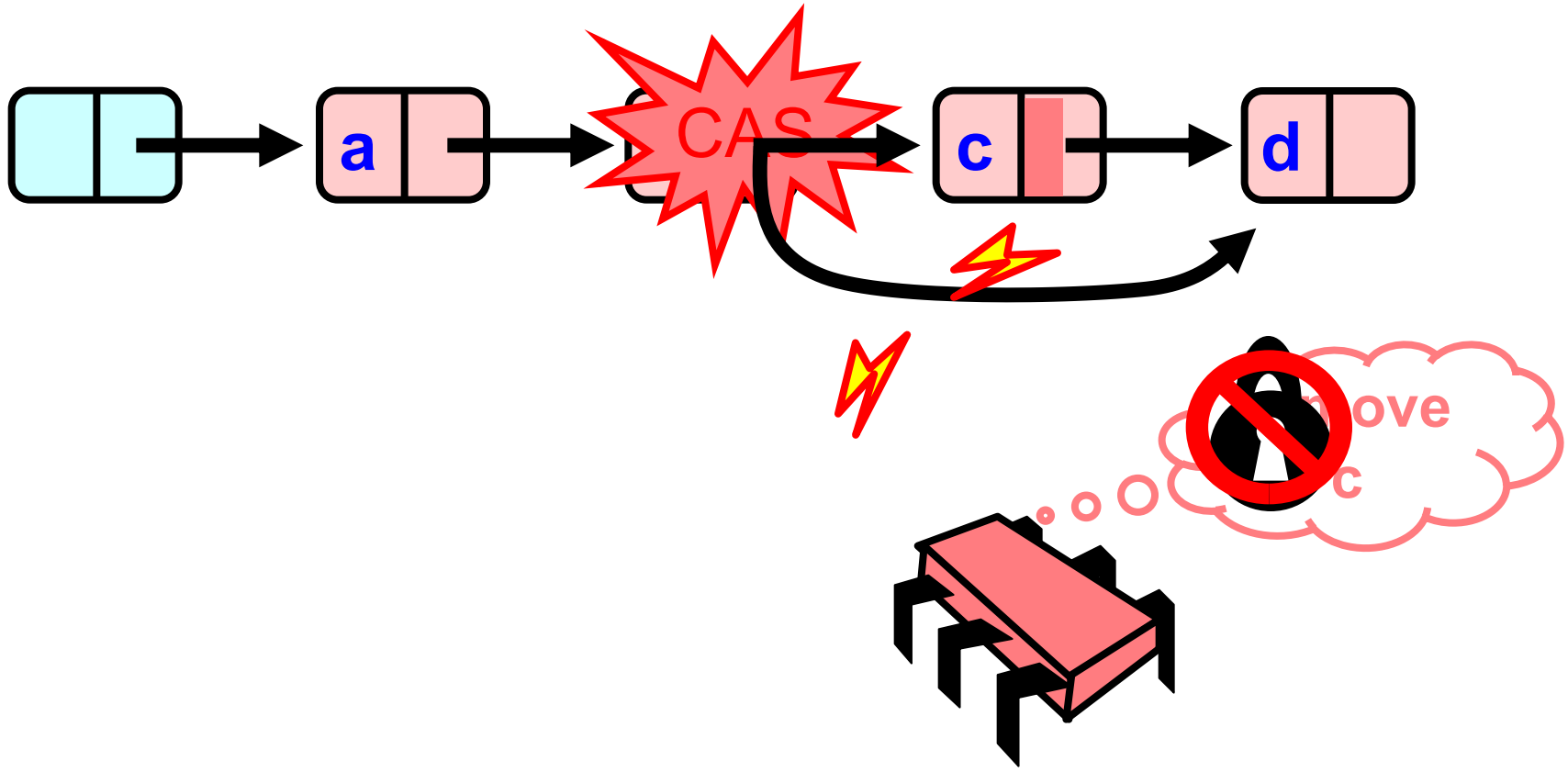
Marking a Node

- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package
  - But we're using a better™ language (C++)

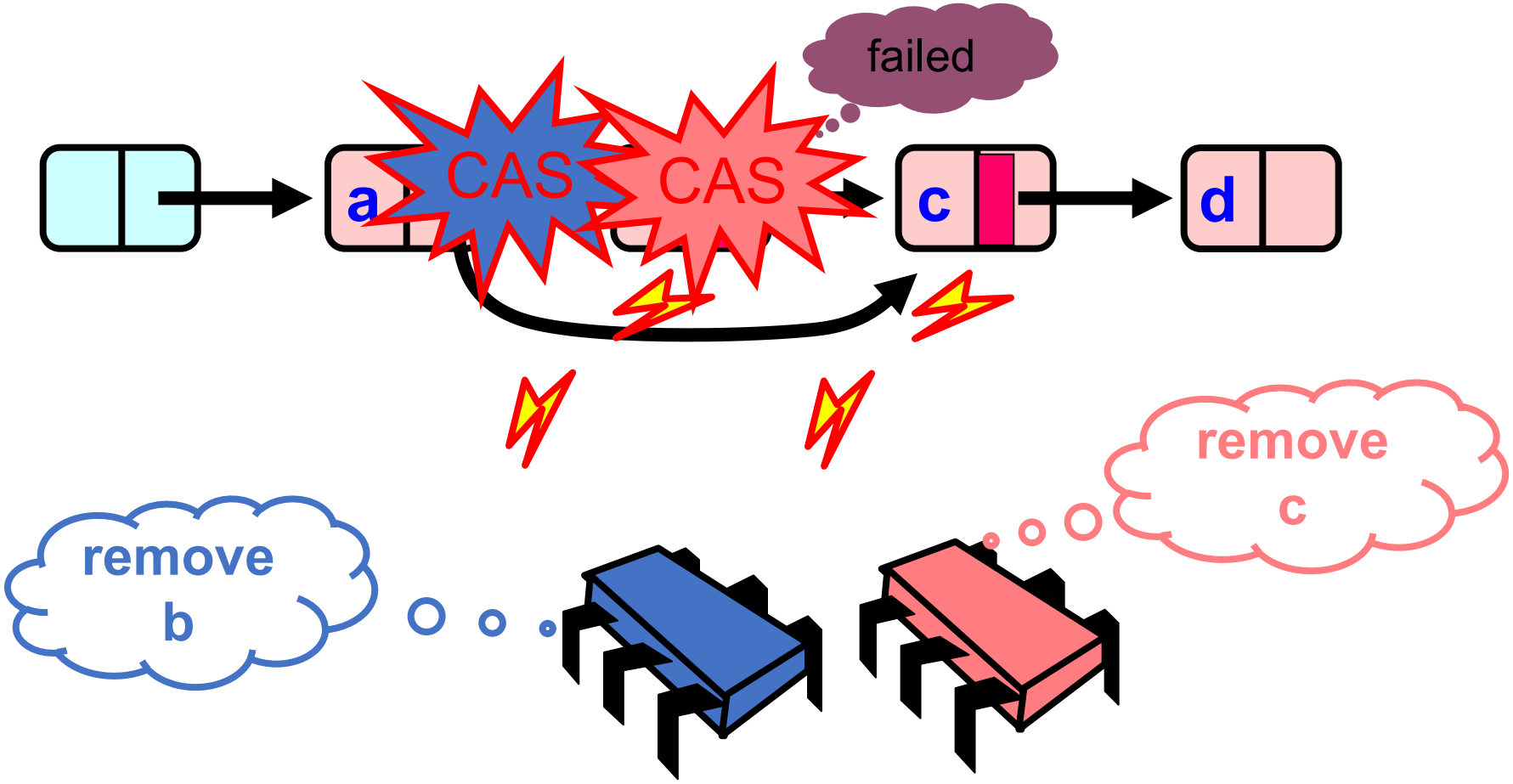
```
class AtomicMarkedNodePtr {
  private:
    atomic<node *> ptr;
  public:
     AtomicMarkedNodePtr(node *p) {
        node * marked = p | 1;
        ptr.store(marked);
     }

     void logically_delete() {
        // how to store the marked bit atomically?
     }

     node * get_ptr() {
        return ptr.load() & (~1);
     }

     bool CAS (node *e, node *n) {
        node * expected = e | 1;
        node * new_node = n | 1;
        return atomic_compare_exchange(&ptr, &expected, new_node);

     }
}
```

Lazy node removal

Removing a Node

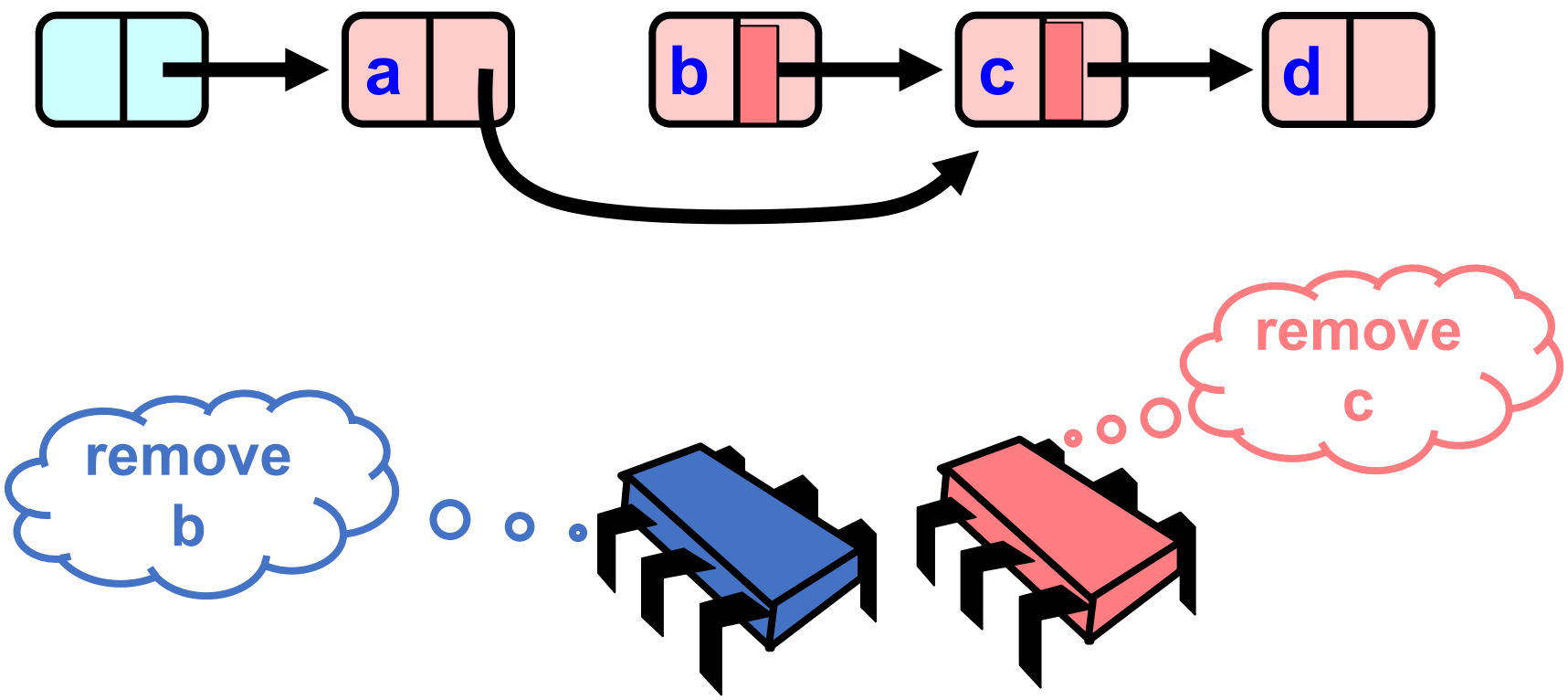


Removing a Node



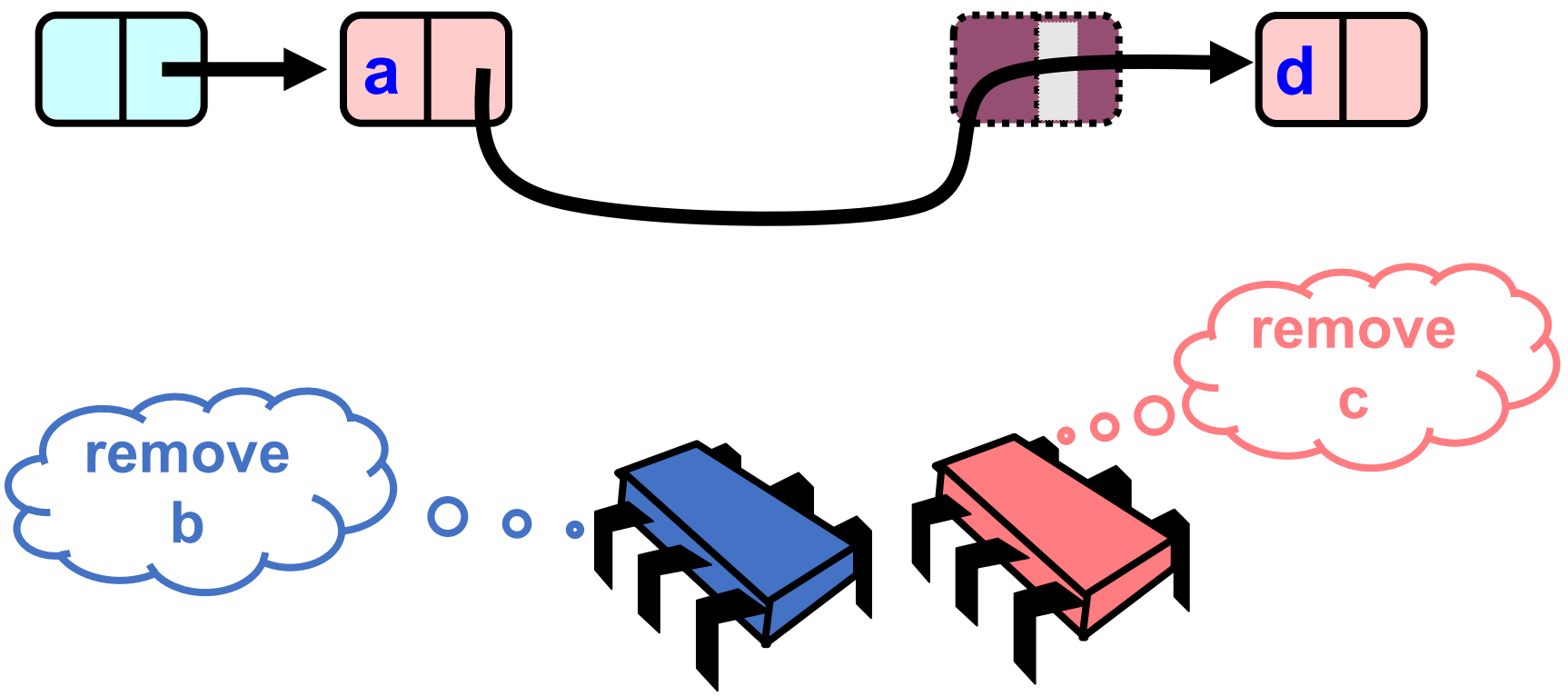
Removing a Node

Two options:
Try removing C again
or...



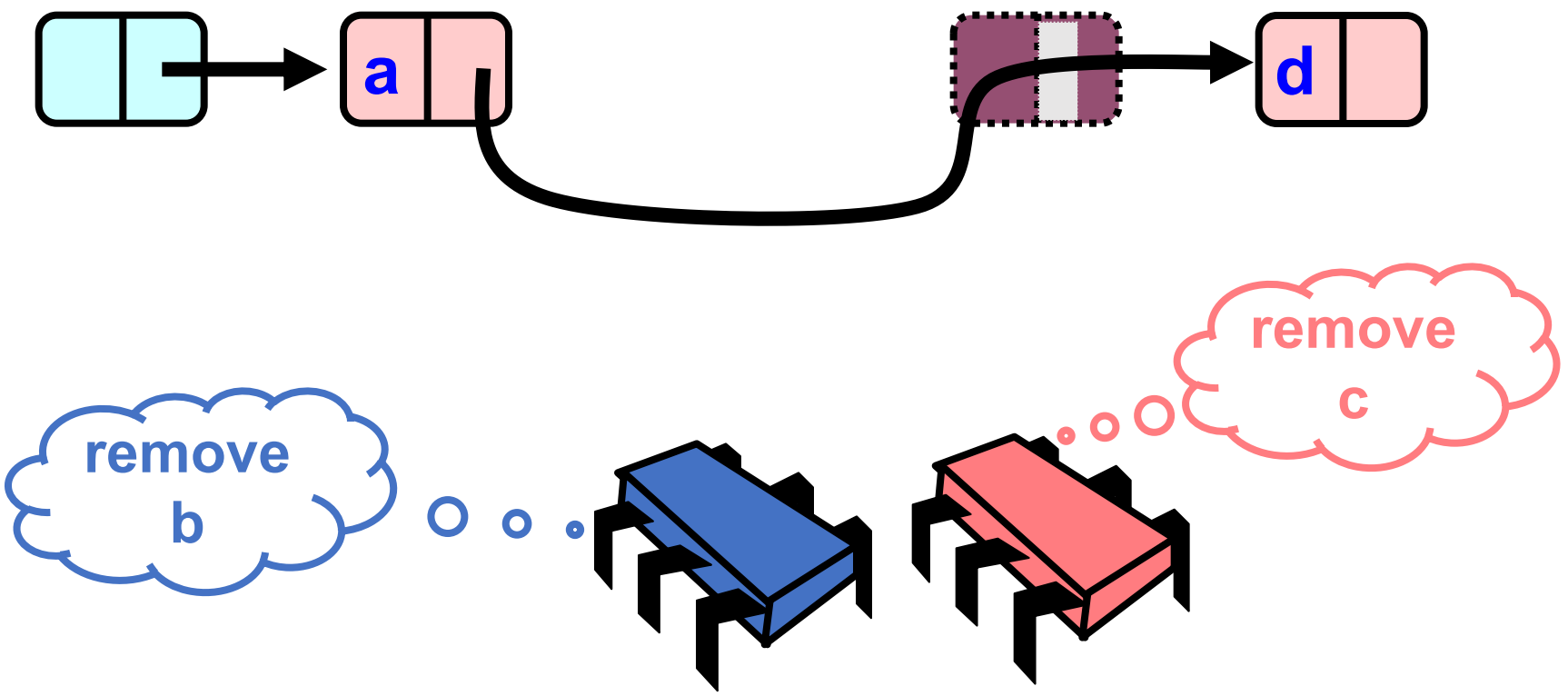
Removing a Node

c stays in the list as logically deleted



Removing a Node

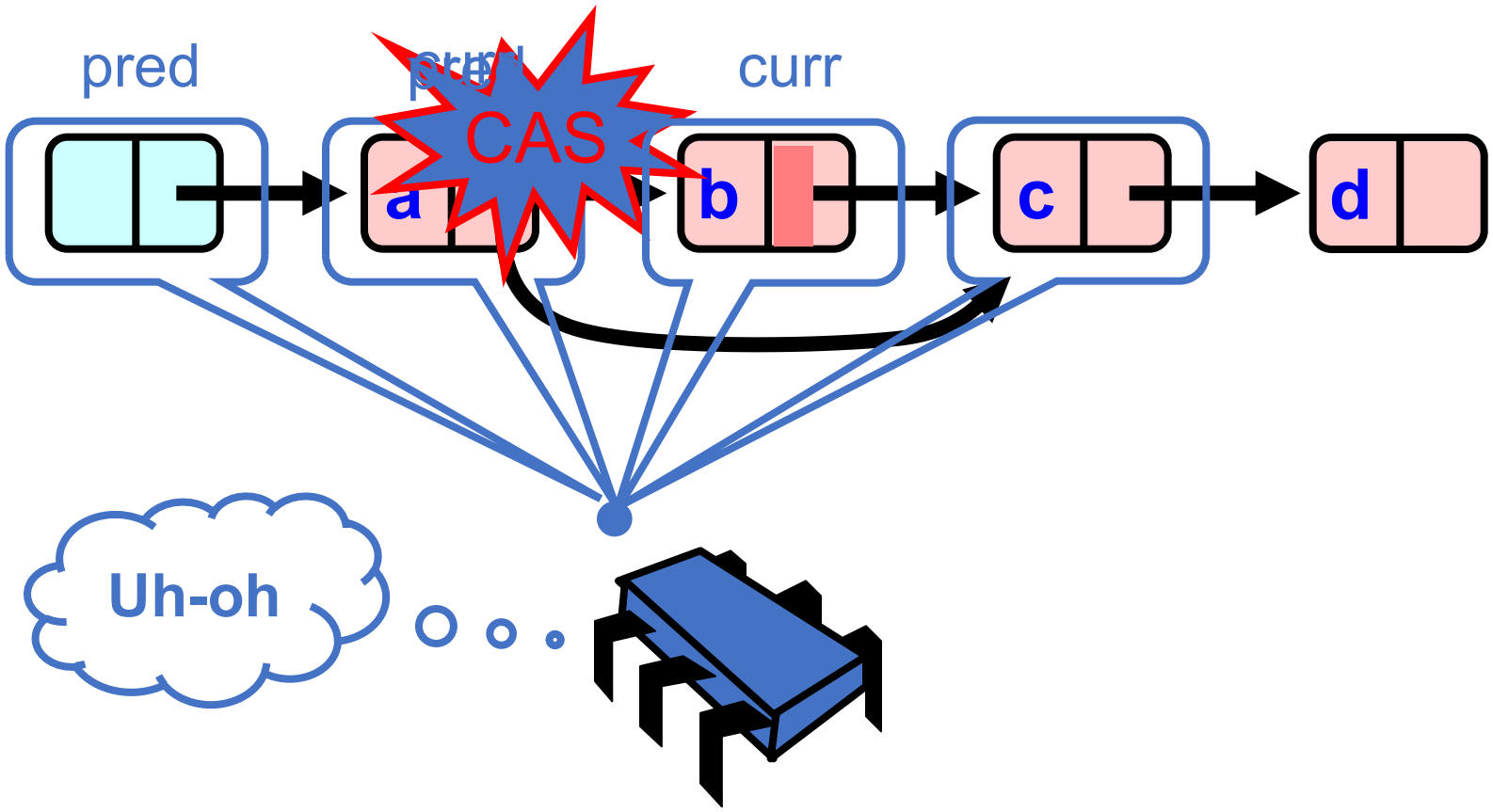
c stays in the list as logically deleted



Traversing the List

- Q: what do you do when you find a “logically” deleted node in your path?
- A: finish the job.
  - CAS the predeqessor’s next field
  - Proceed (repeat as needed)

Lock-Free Traversal



Further Reading

- Chapter 9 goes over implementations in detail.
  - This is tricky stuff! Please read to get a different perspective!

- Skip Lists
  - Binary search over linked list (log(n) lookup time)
  - Chapter 14 of the book

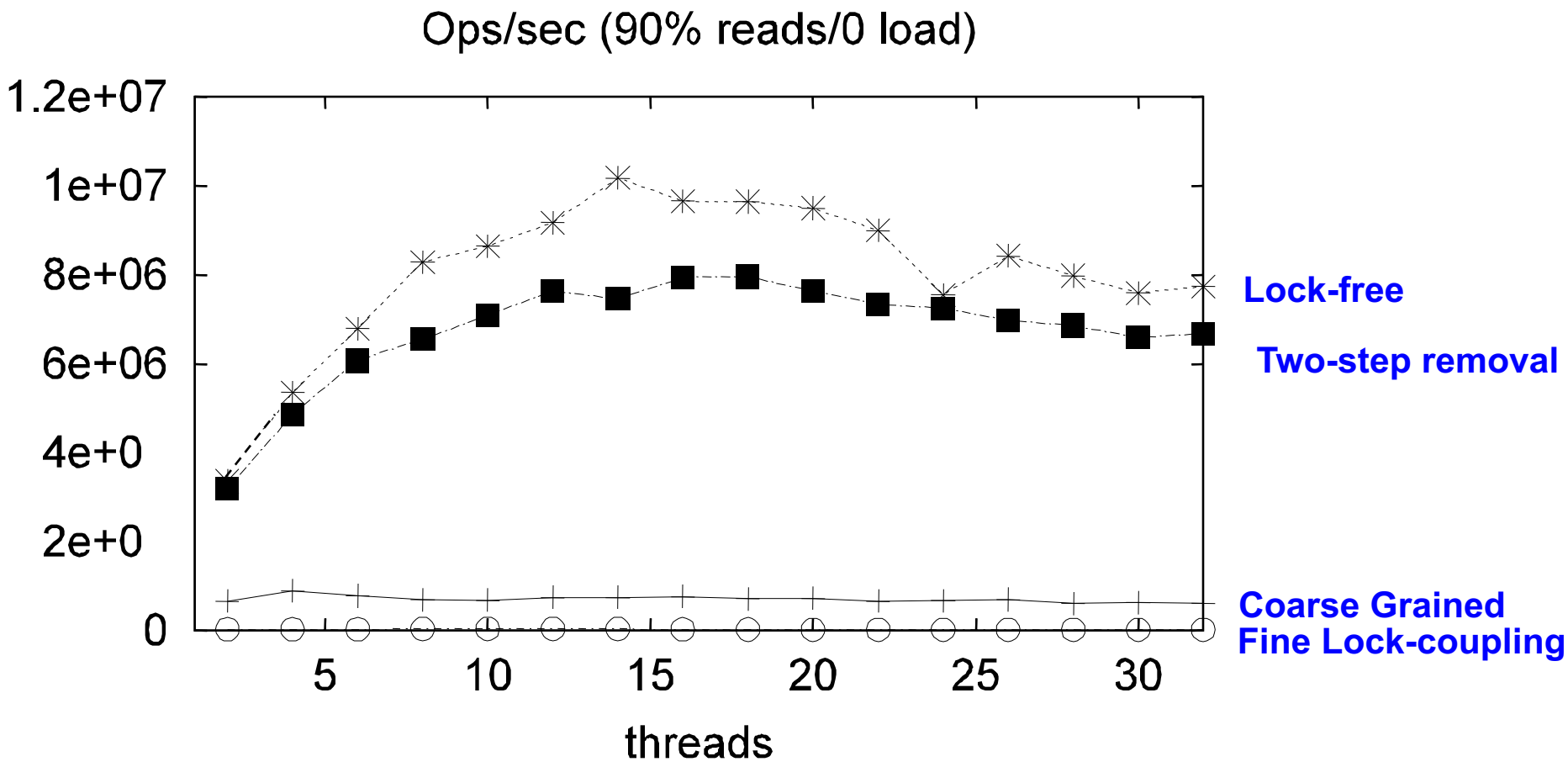
Performance

- Issues:
  - Lazy removal makes benchmarking traversals very tricky
  - Garbage collection makes benchmarking very tricky

Some performance results

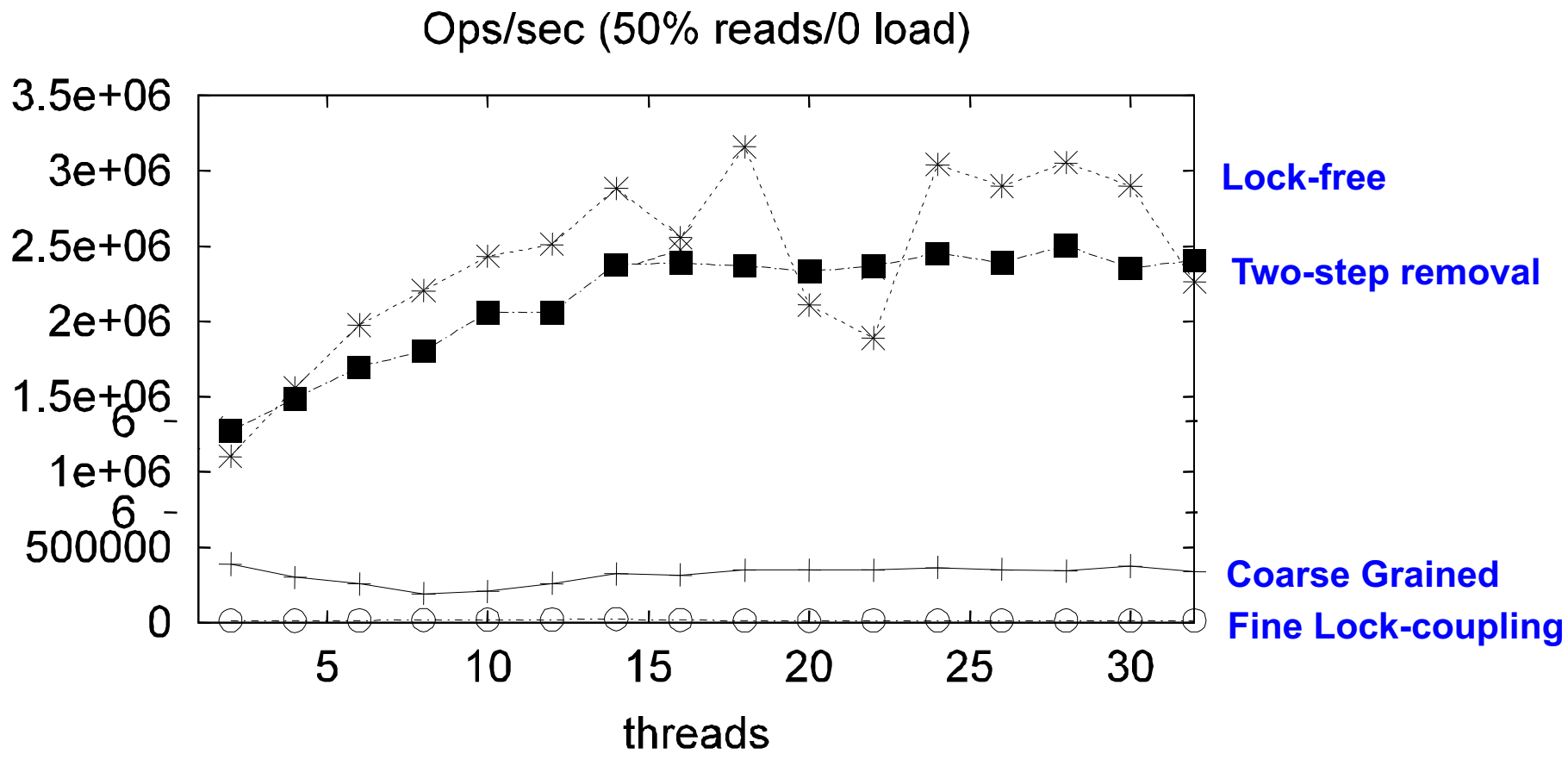
From: A Lazy Concurrent List-Based Set Algorithm: 2005 publication from the textbook authors research group

High Contains Ratio



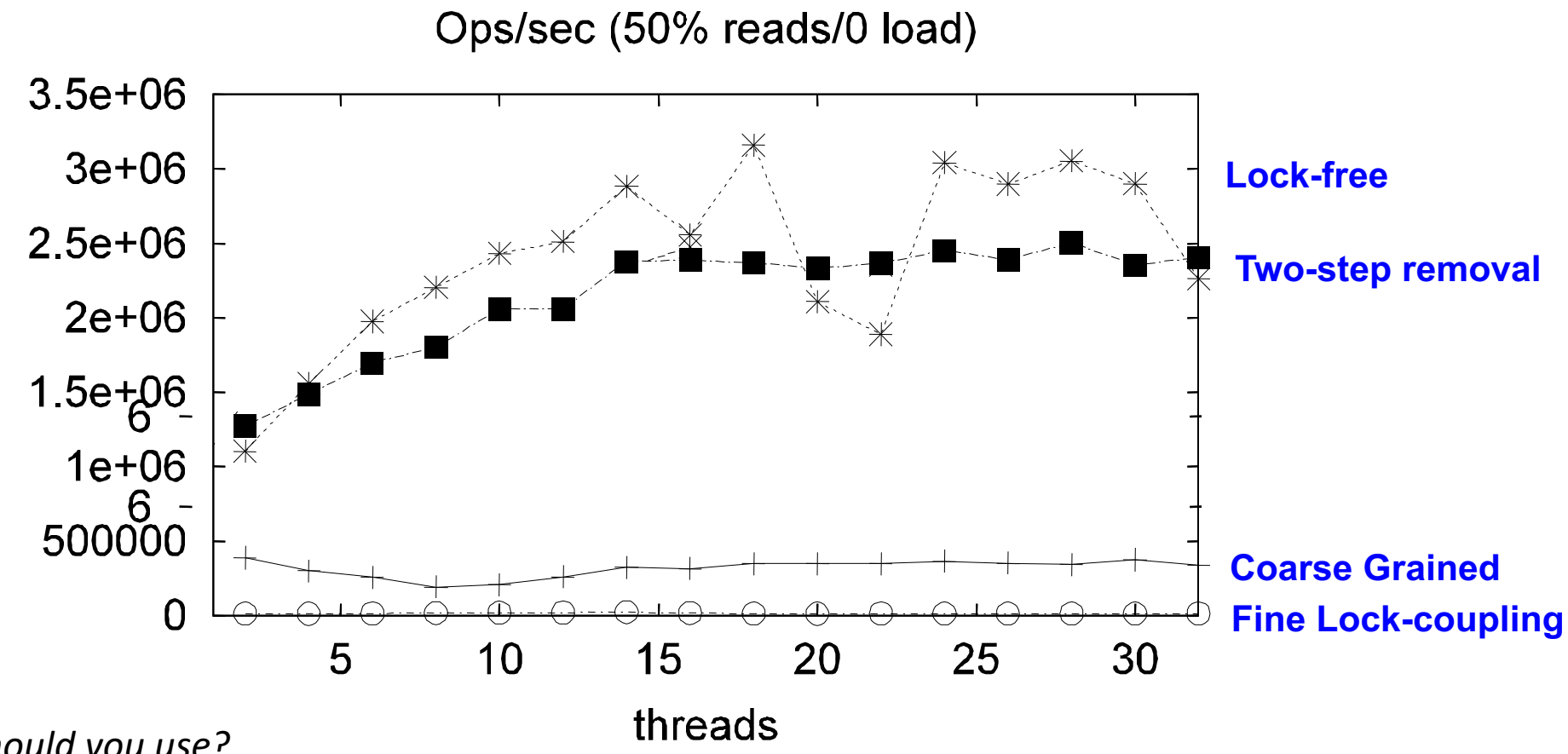
Low Contains Ratio

noisy!



Low Contains Ratio

noisy!



Which one should you use?

Schedule

- Finish up linked-list

- **Concurrent Queues**
  - Input/Output Queues
  - Synchronous Producer/Consumer Queue
  - Async Producer/Consumer Queue

Concurrent Queues

- New API
- List of items, accessed in a first-in first-out (FIFO) way
- *duplicates allowed*
- Methods
  - **`enq(x)`** put **`x`** in the list at the head
  - **`deq()`** remove the item at the tail of the queue and return it.
  - **`size()`** returns how many items are in the queue

Concurrent Queues

- General implementation given in Chapter 10 of the book.
- Similar types of reasoning as the linked list
  - Lots of reasoning about node insertion, node deletion
  - Using atomic RMWs (CAS) in clever ways

- We will think about specialized queues
  - Implementations can be simplified!

Input/Output Queues

- Queue in which multiple threads read (deq), or write (enq), but not both.

- Why would we want a thing?

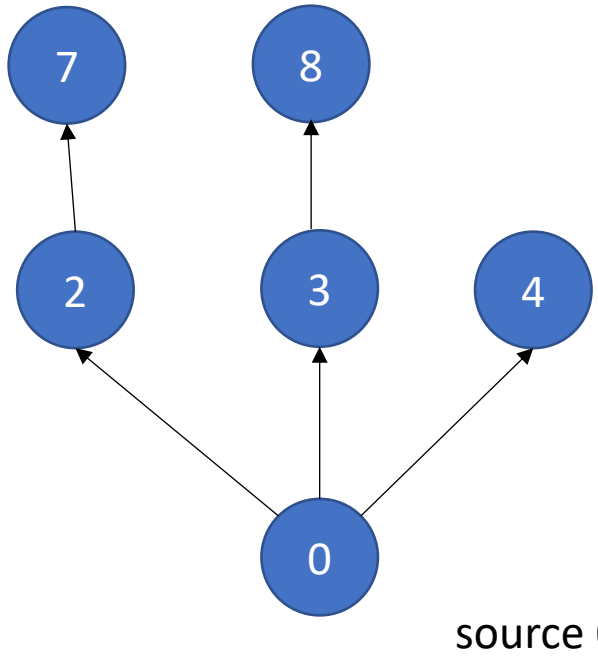
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

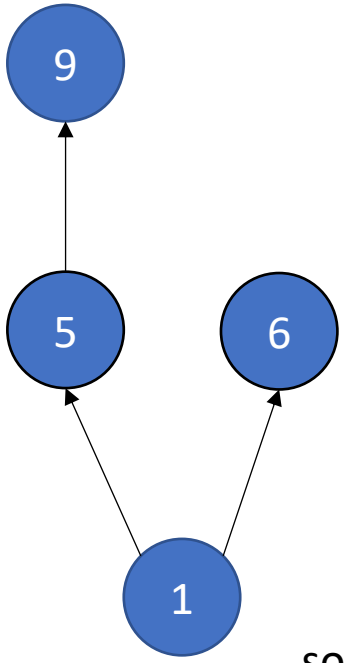
- Example: Information flow in graph applications:

Input/Output Queues

- Example: Information flow in graph applications:



source 0



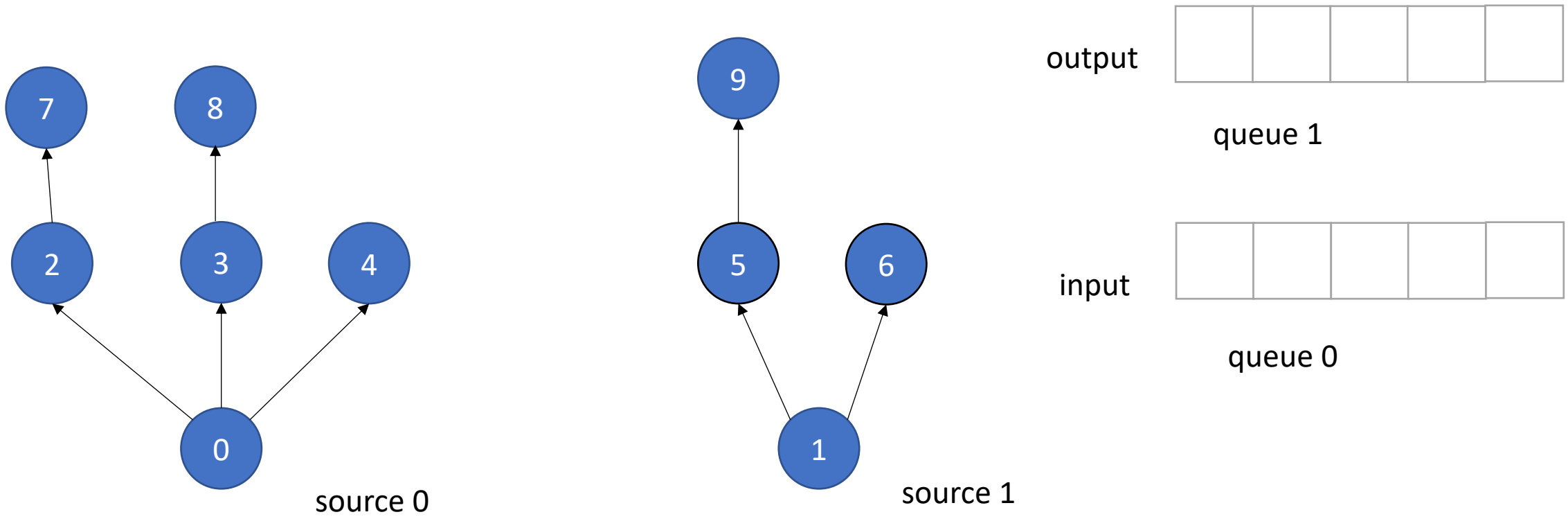
source 1

queue 1

queue 0

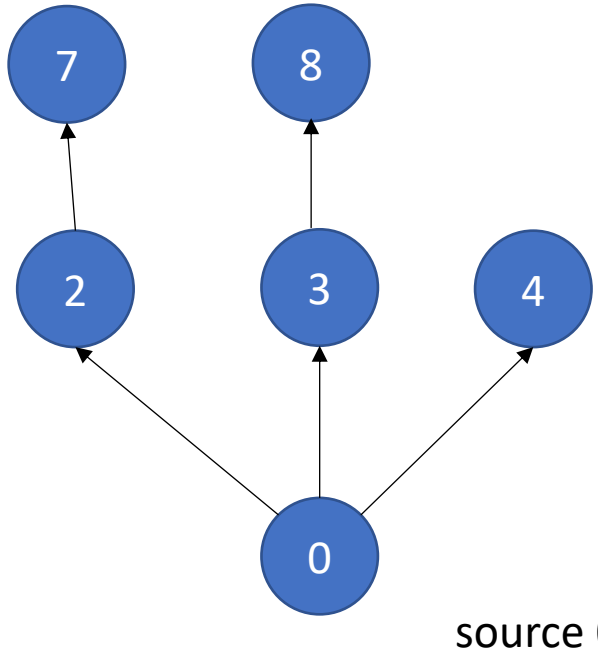
Input/Output Queues

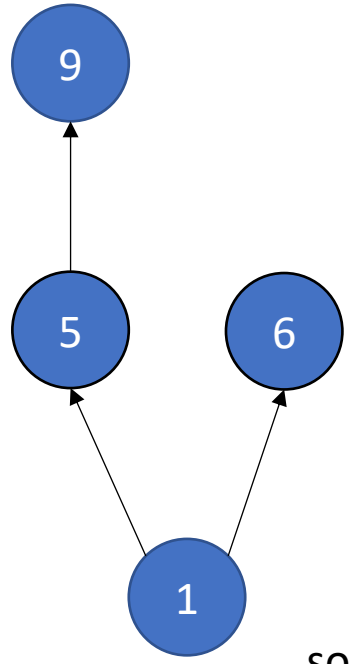
- Example: Information flow in graph applications:

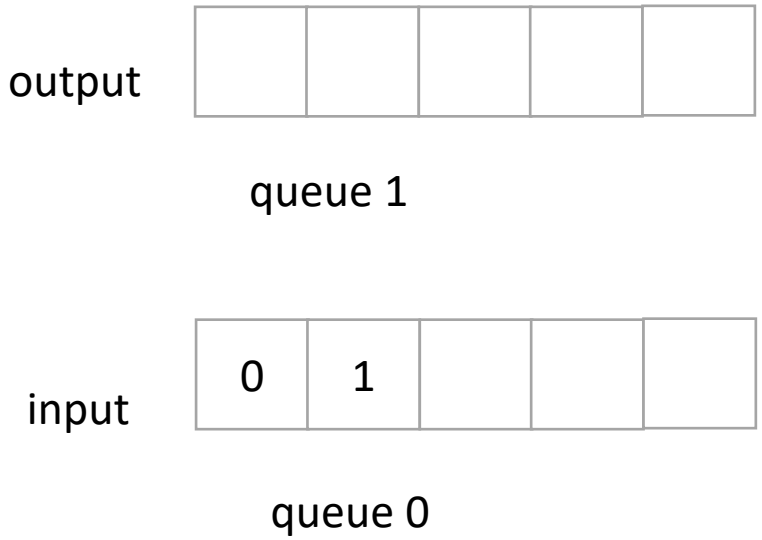


Input/Output Queues

- Example: Information flow in graph applications:

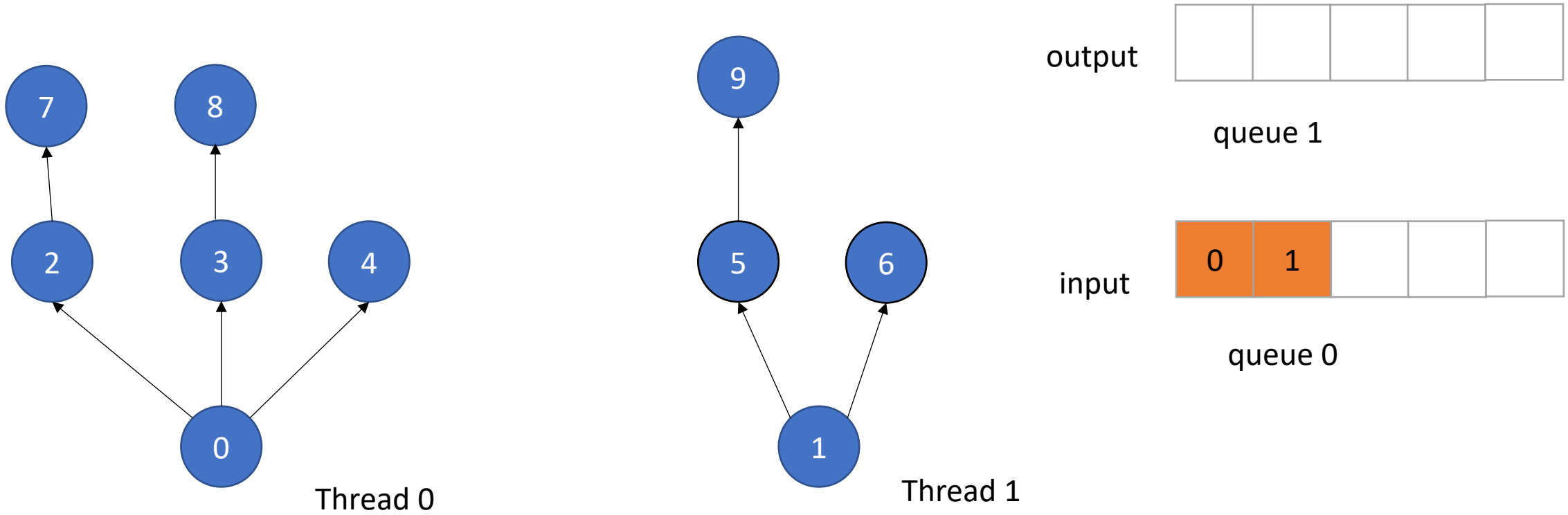






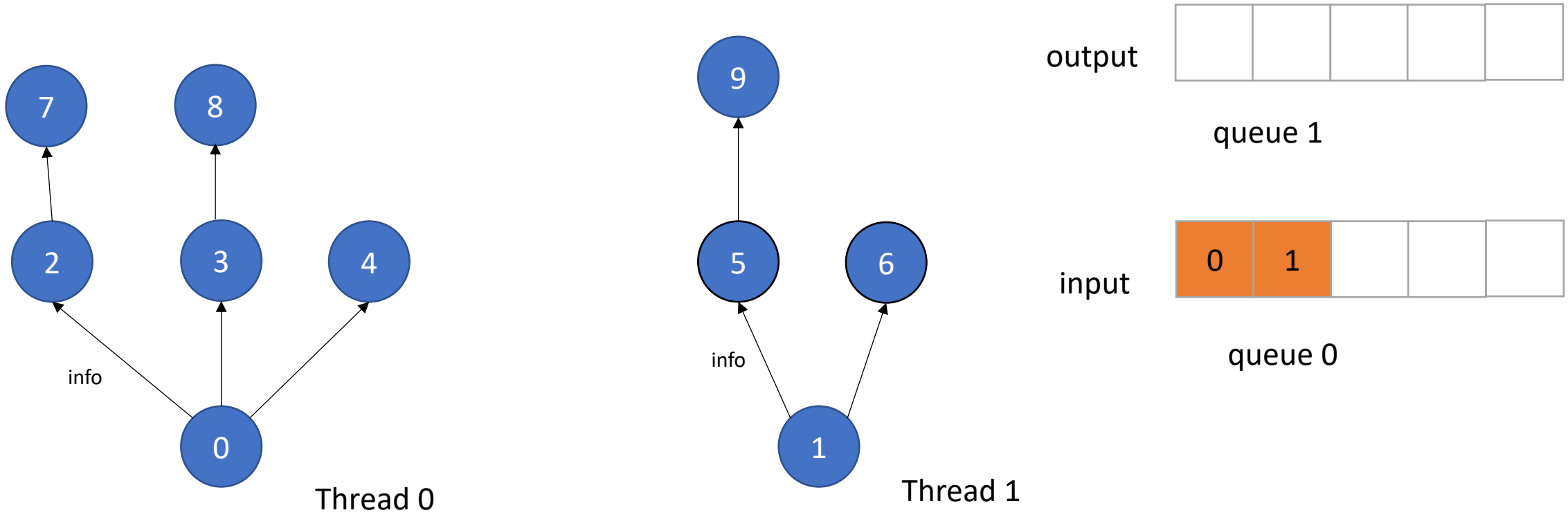
Input/Output Queues

- Example: Information flow in graph applications:



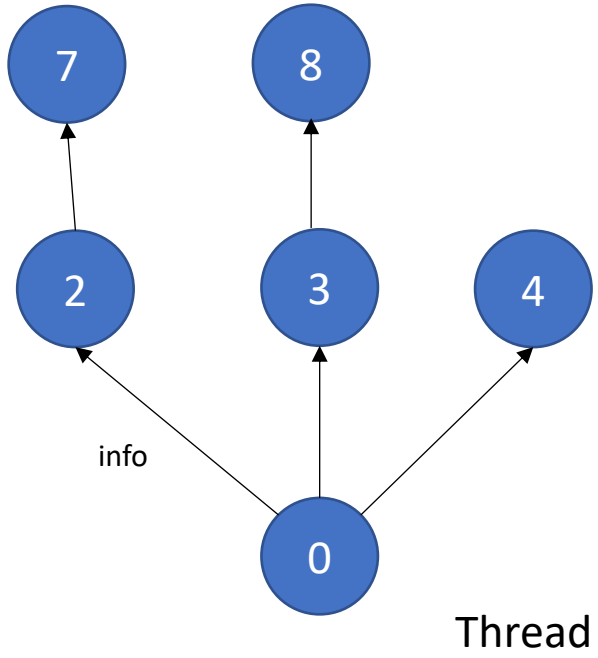
Input/Output Queues

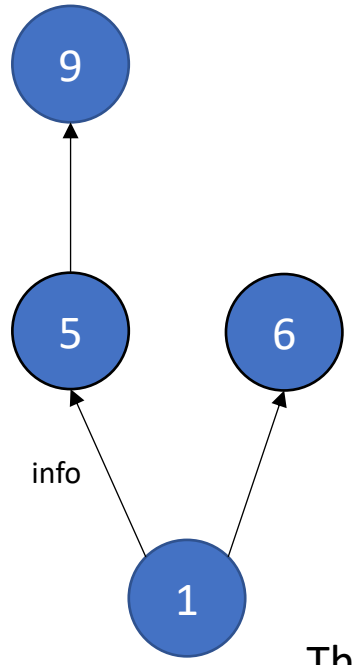
- Example: Information flow in graph applications:

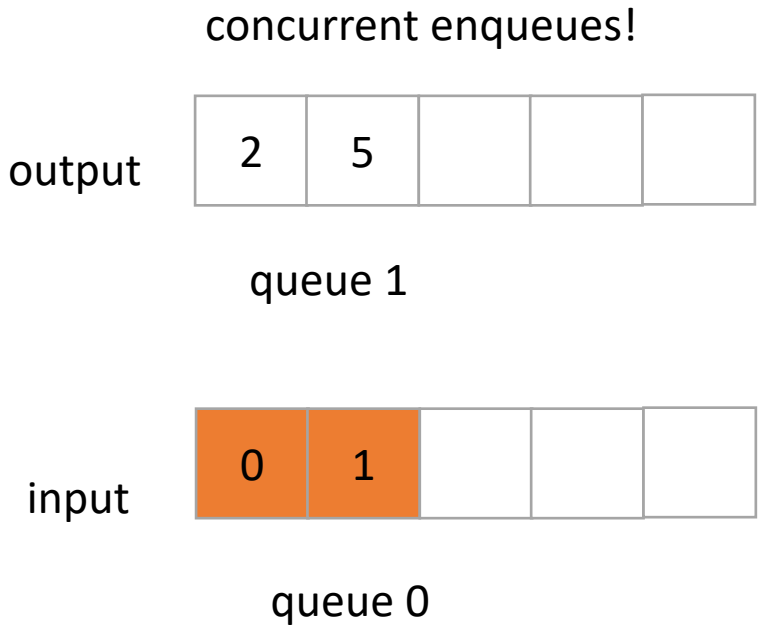


Input/Output Queues

- Example: Information flow in graph applications:

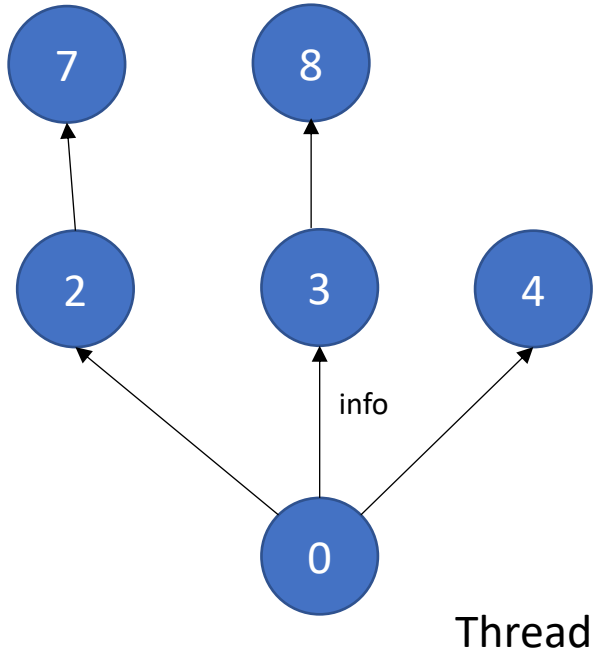


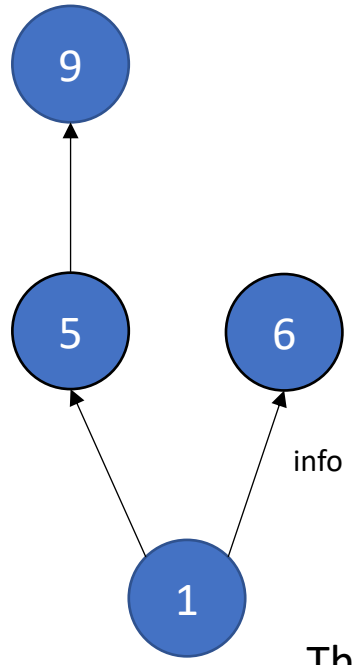


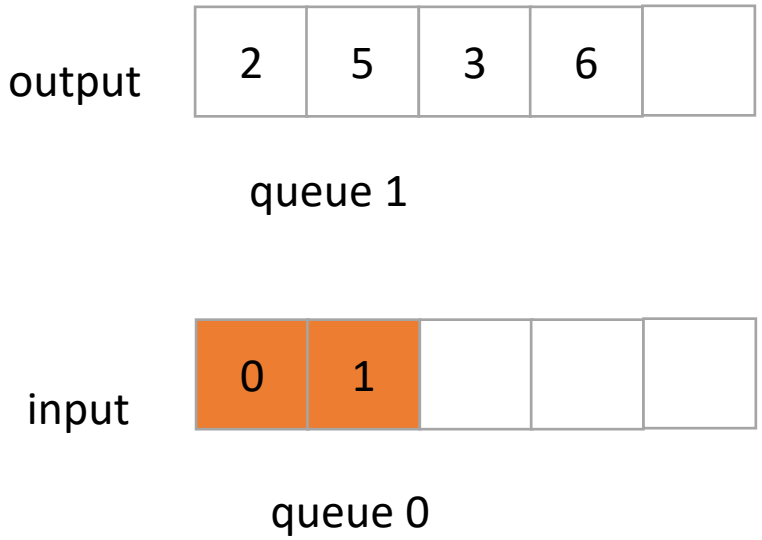


Input/Output Queues

- Example: Information flow in graph applications:

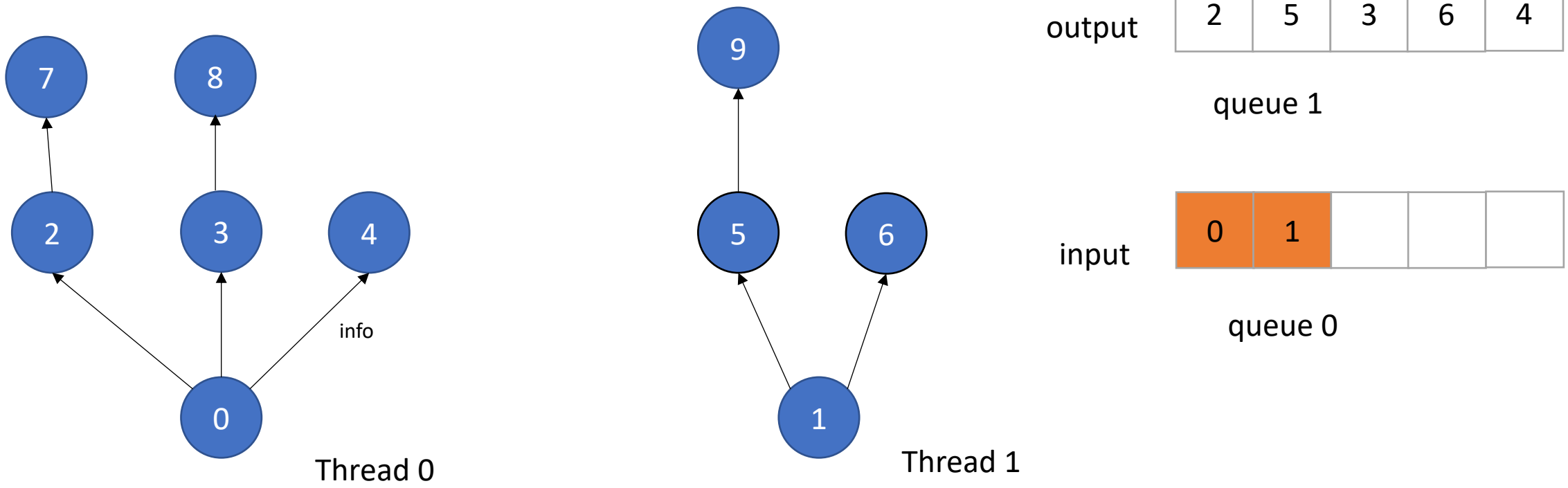






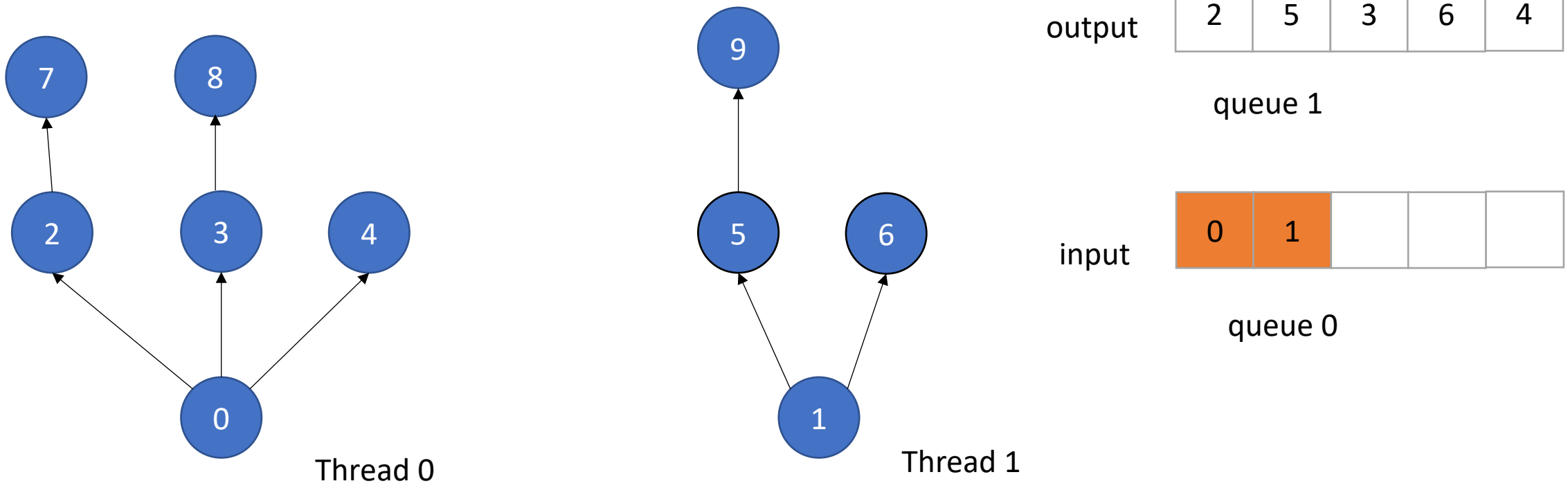
Input/Output Queues

- Example: Information flow in graph applications:



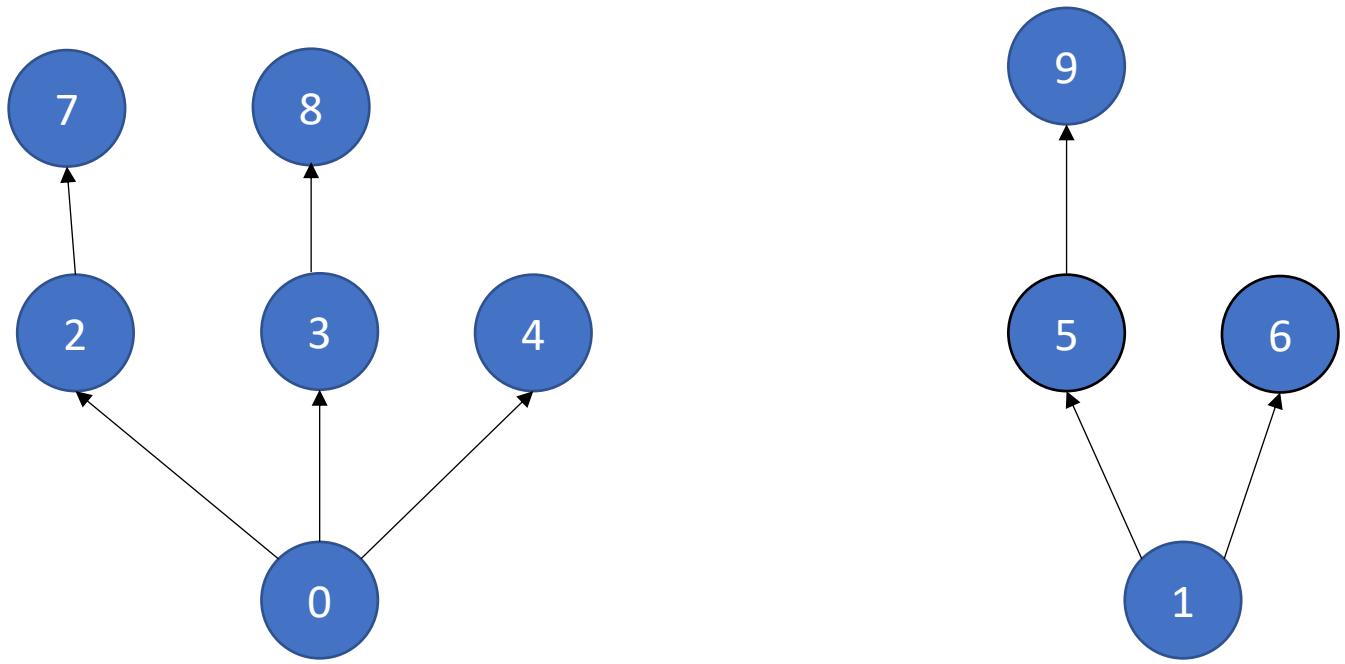
Input/Output Queues

- Example: Information flow in graph applications:



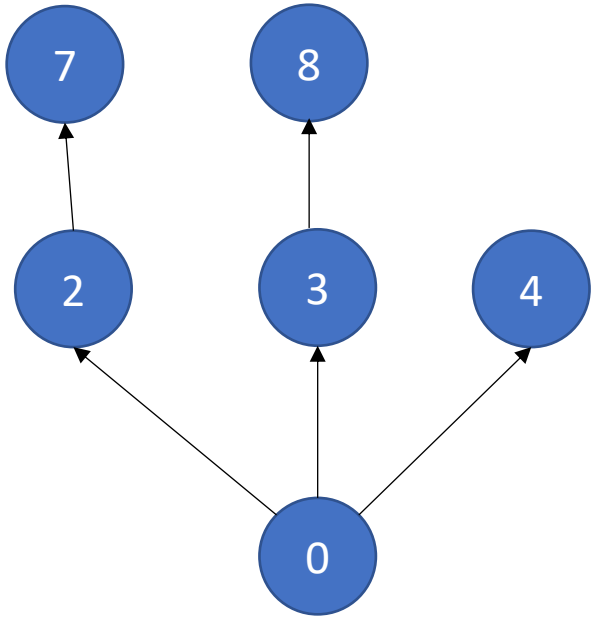
Input/Output Queues

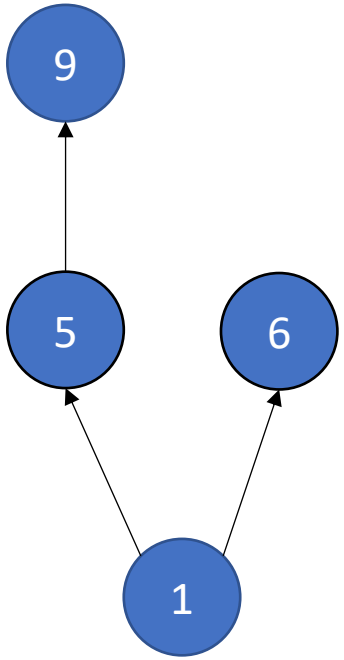
- Example: Information flow in graph applications:

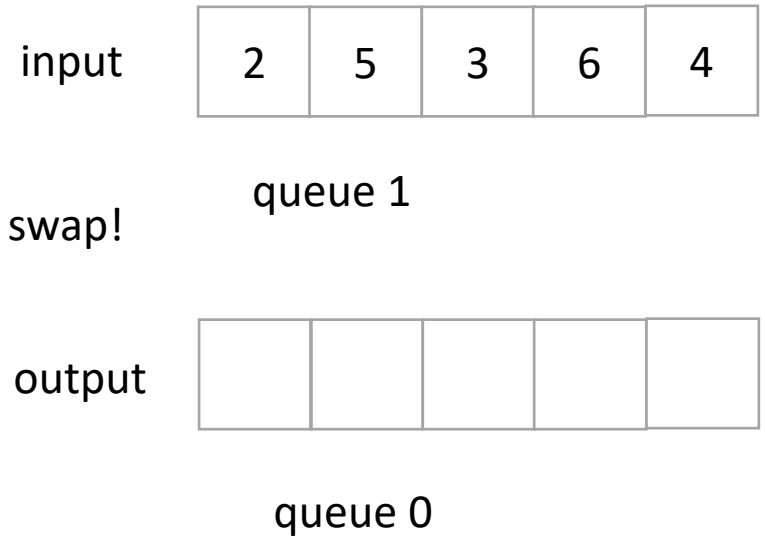


Input/Output Queues

- Example: Information flow in graph applications:

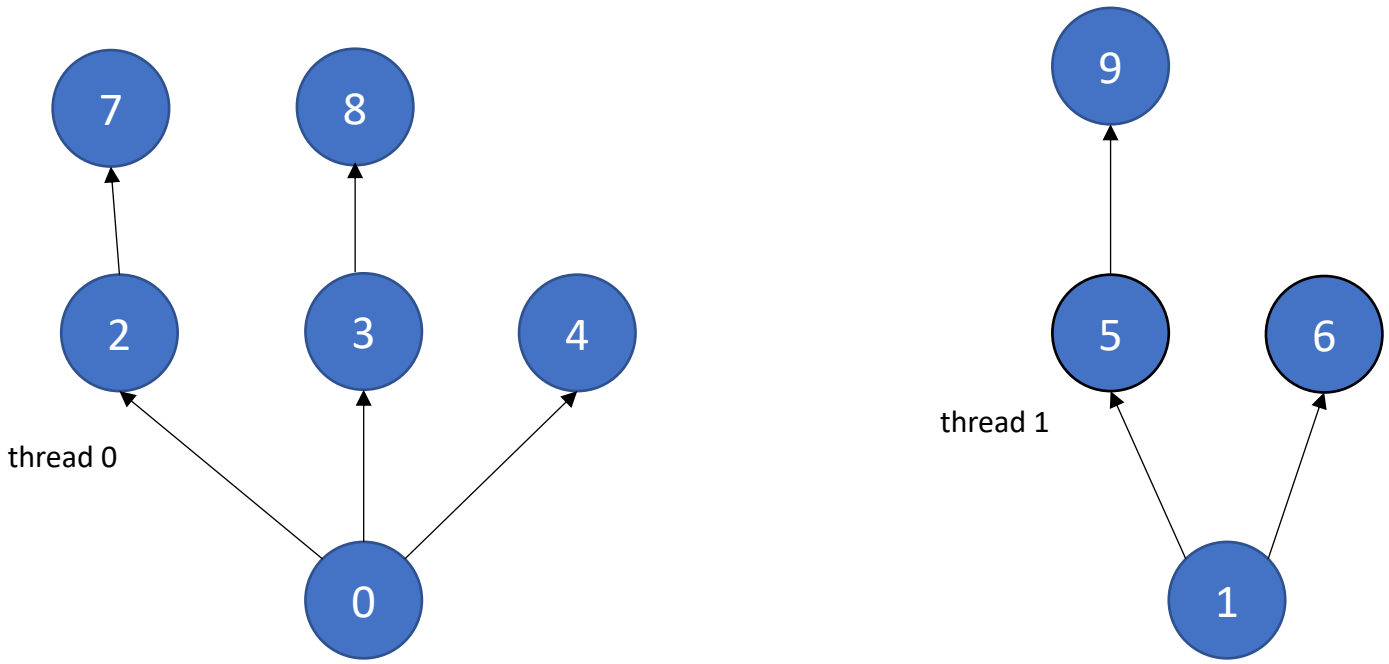






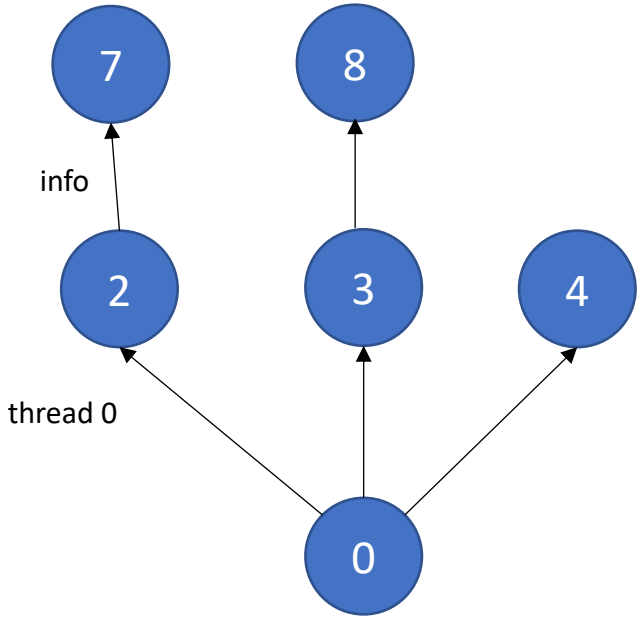
Input/Output Queues

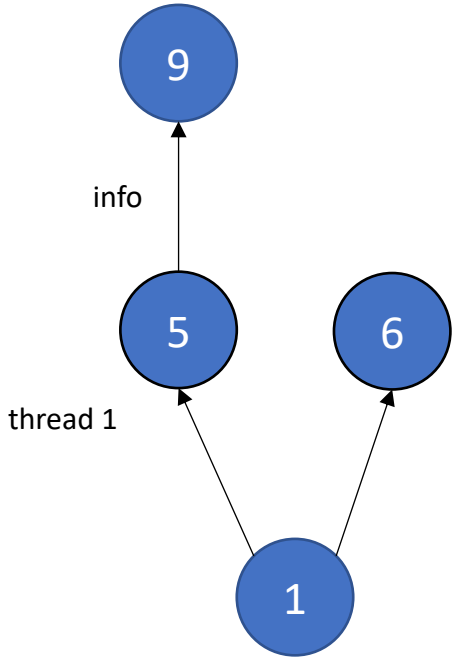
- Example: Information flow in graph applications:

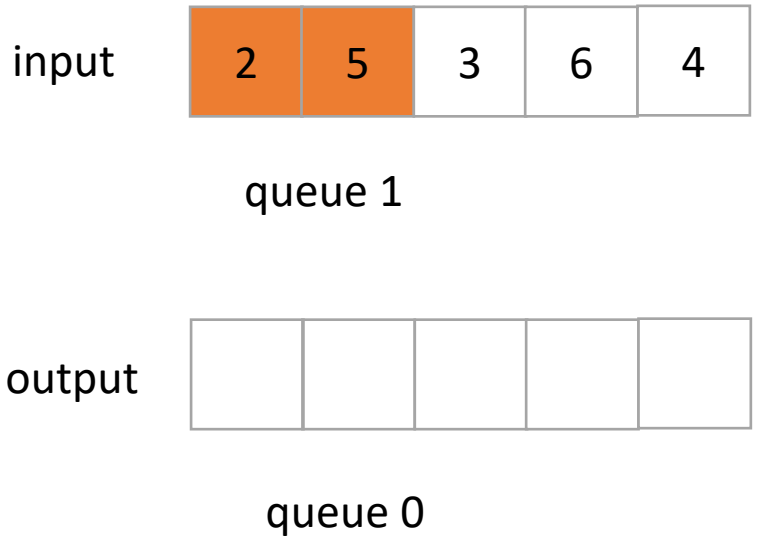


Input/Output Queues

- Example: Information flow in graph applications:

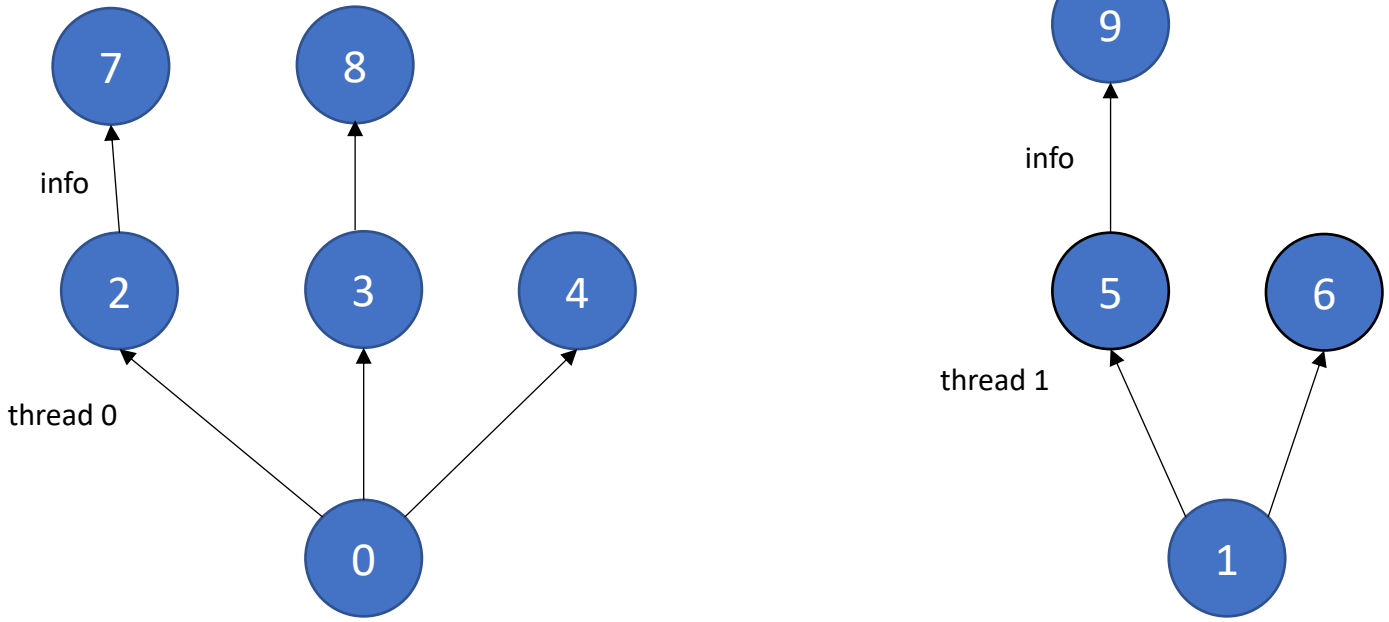






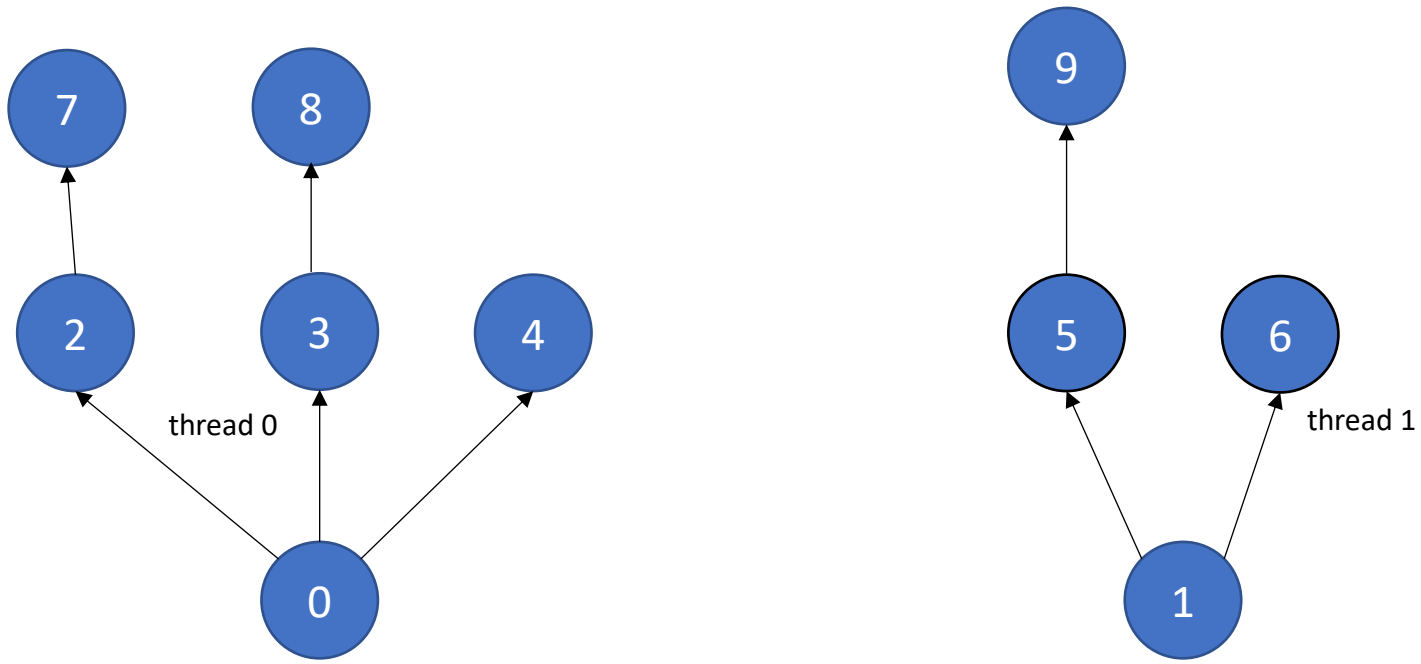
Input/Output Queues

- Example: Information flow in graph applications:



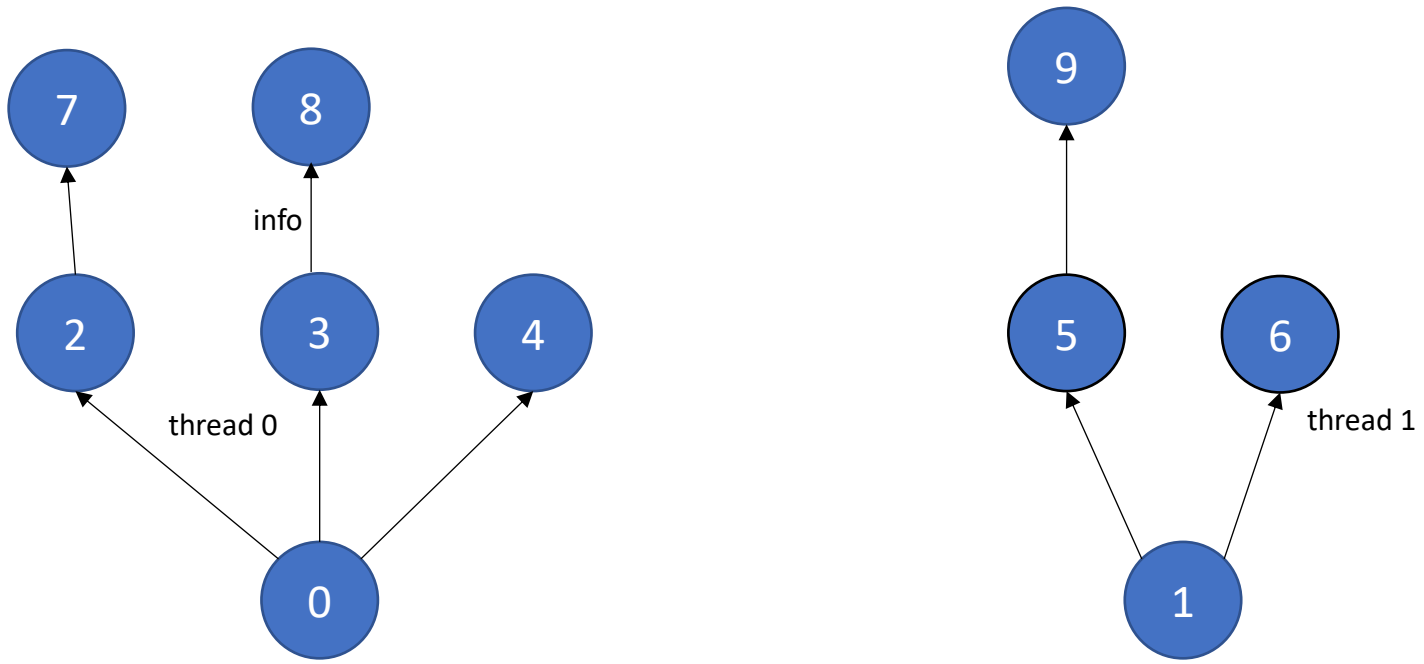
Input/Output Queues

- Example: Information flow in graph applications:



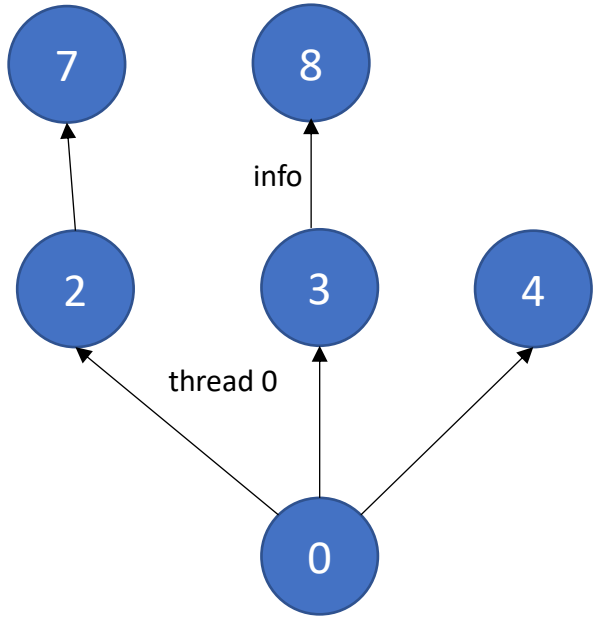
Input/Output Queues

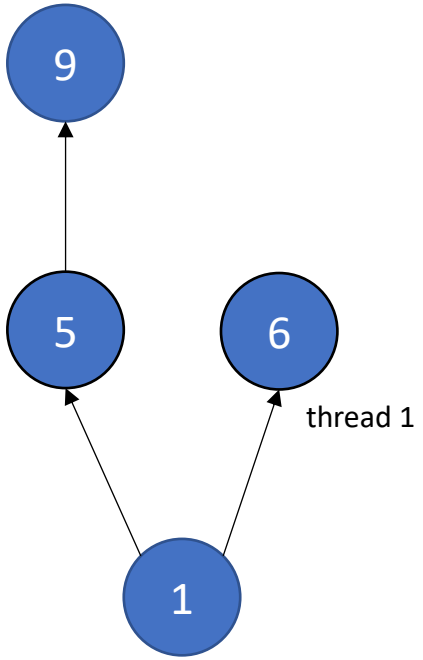
- Example: Information flow in graph applications:

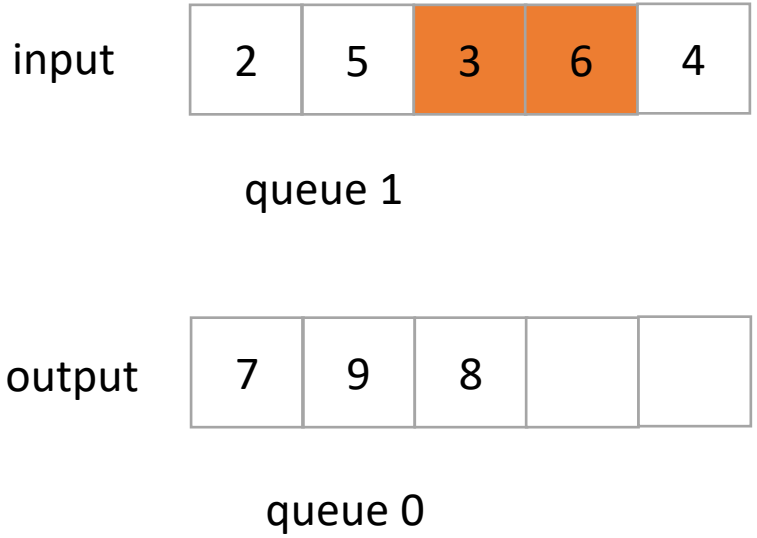


Input/Output Queues

- Example: Information flow in graph applications:

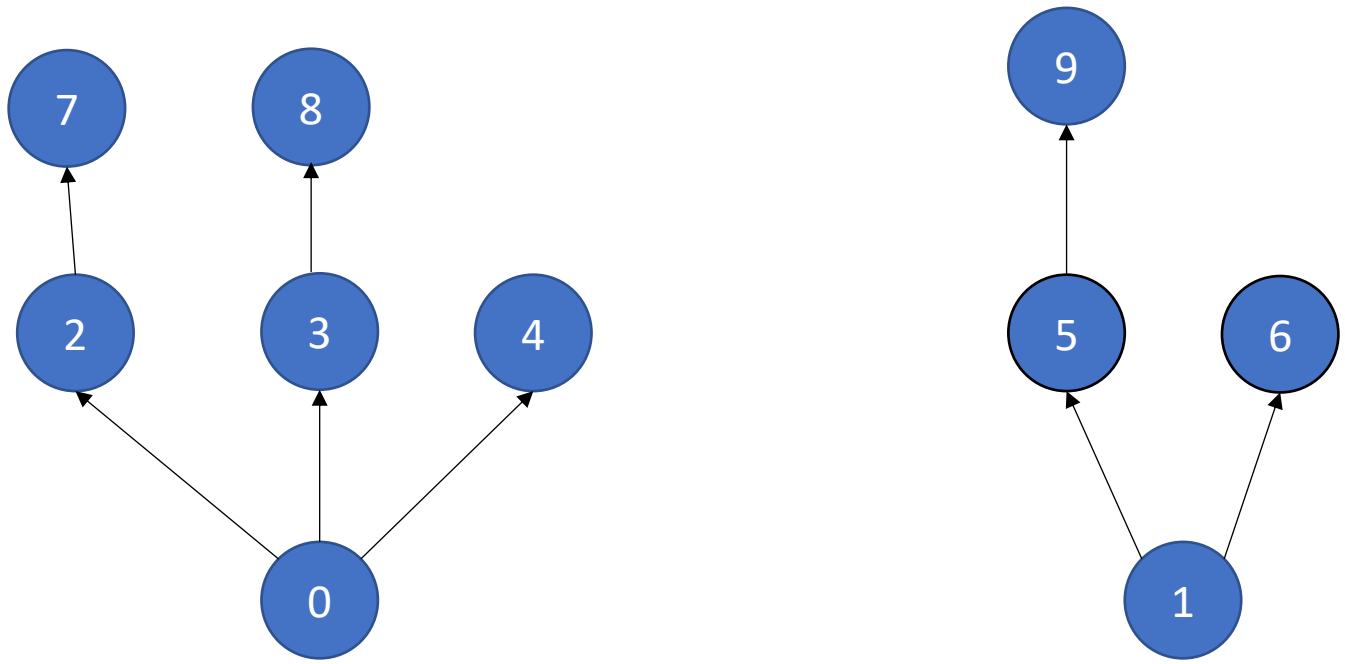


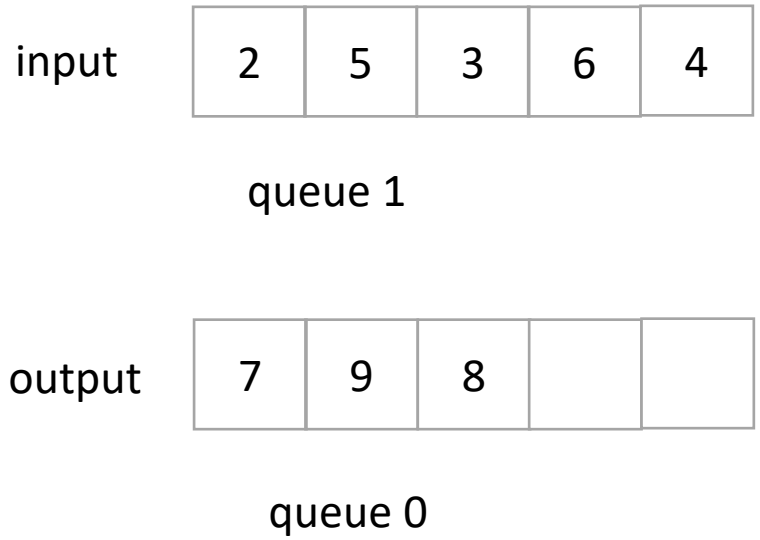




Input/Output Queues

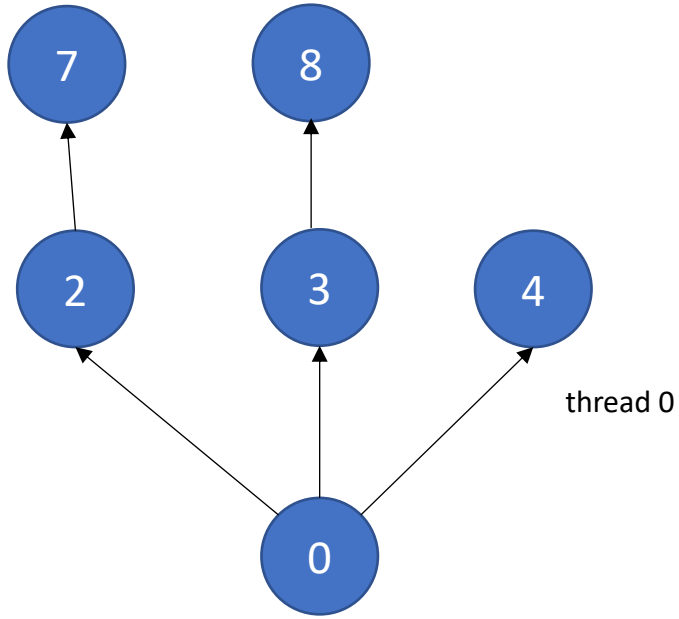
- Example: Information flow in graph applications:

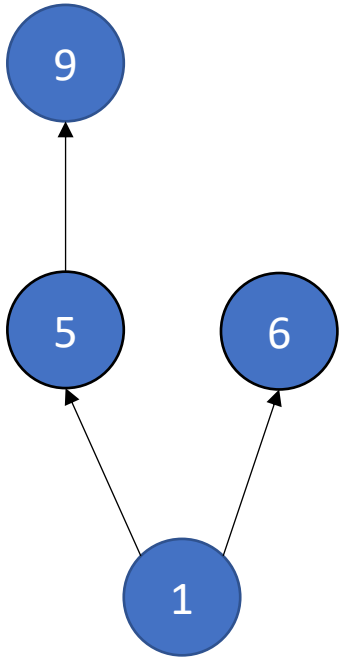


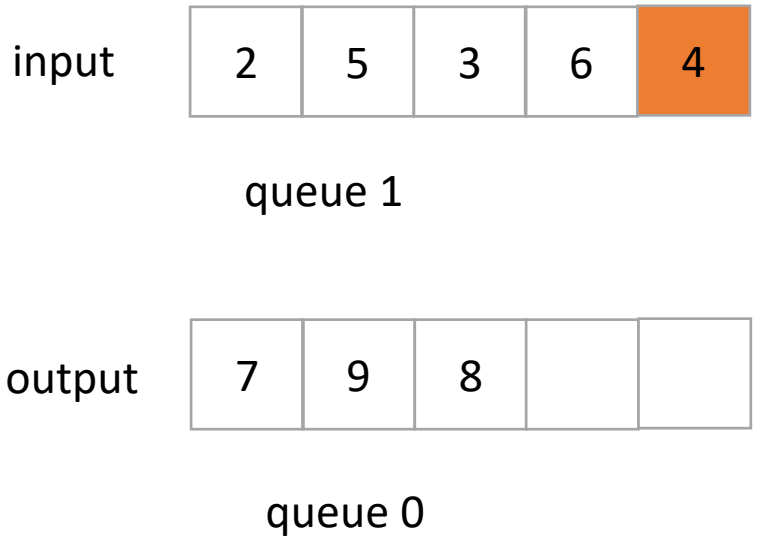


Input/Output Queues

- Example: Information flow in graph applications:

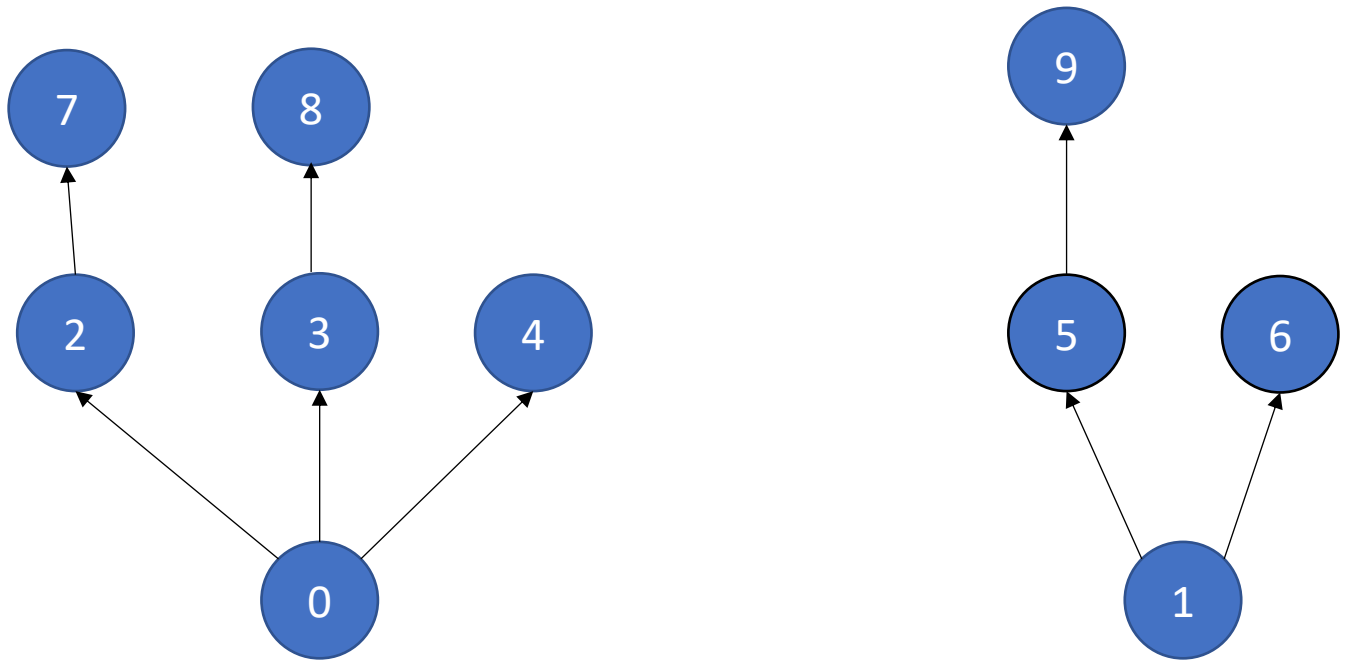


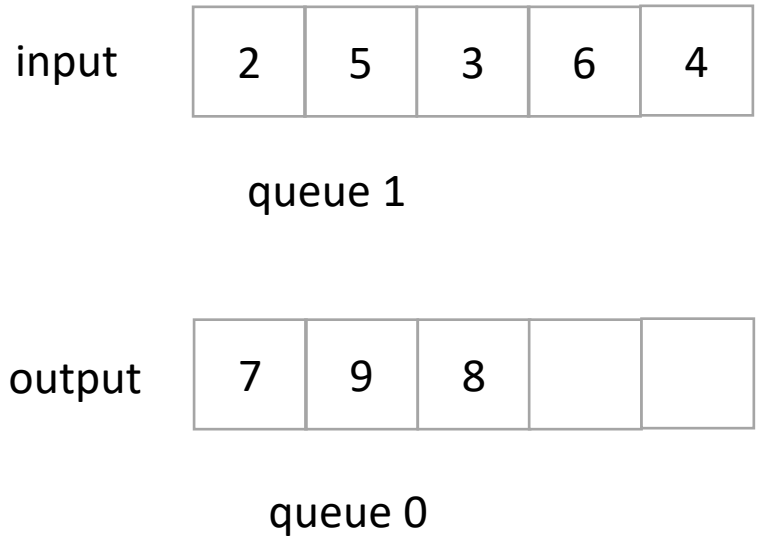




Input/Output Queues

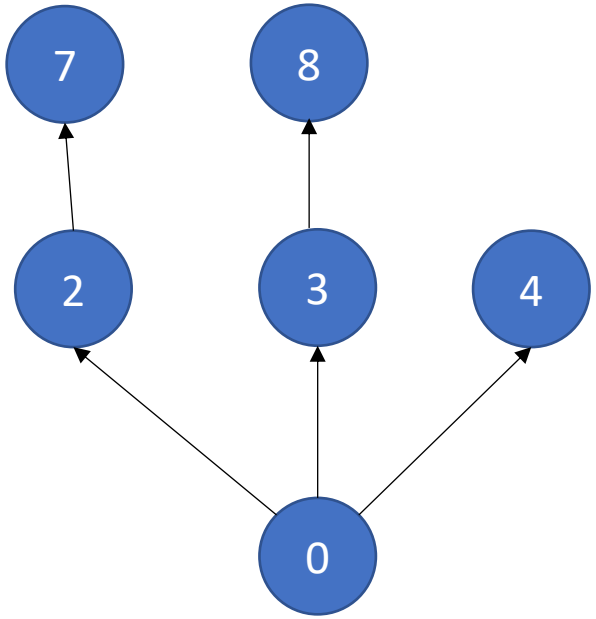
- Example: Information flow in graph applications:

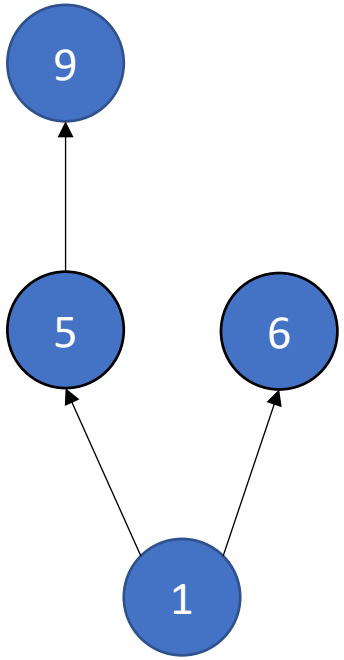




Input/Output Queues

- Example: Information flow in graph applications:





and so on...

output

queue 1

input

7	9	8		

queue 0

Implementation

Implementation

Allocate a contiguous array

Pros:
?

Cons:
?

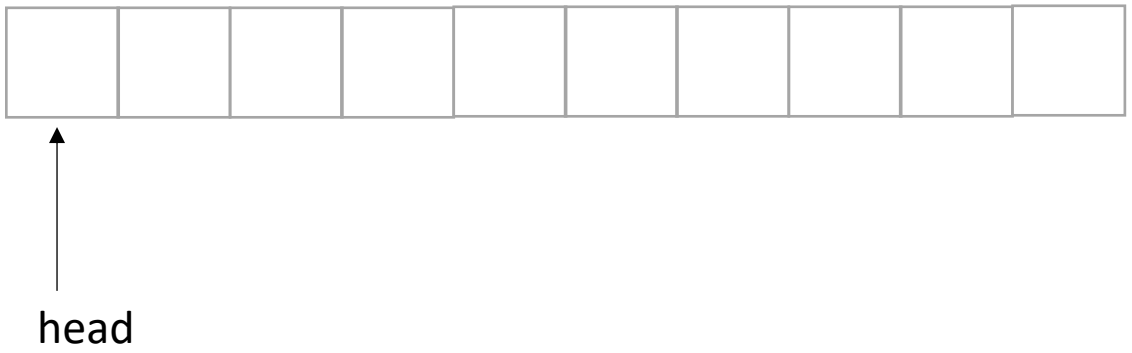
Implementation

Allocate a contiguous array

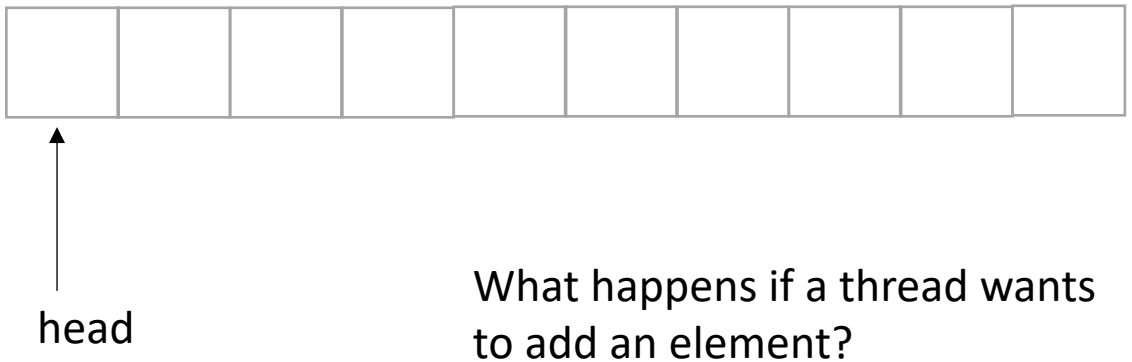
Pros:
+ fast!
+ we can use indexes instead of addresses

Cons:
- need to reason about overflow!

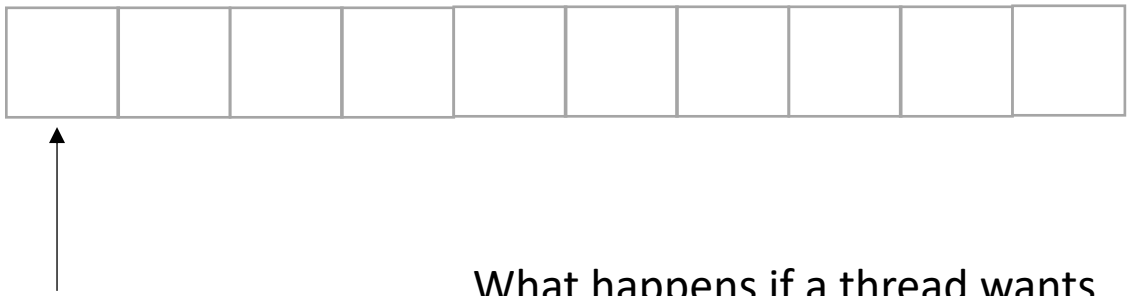
Implementation



Implementation



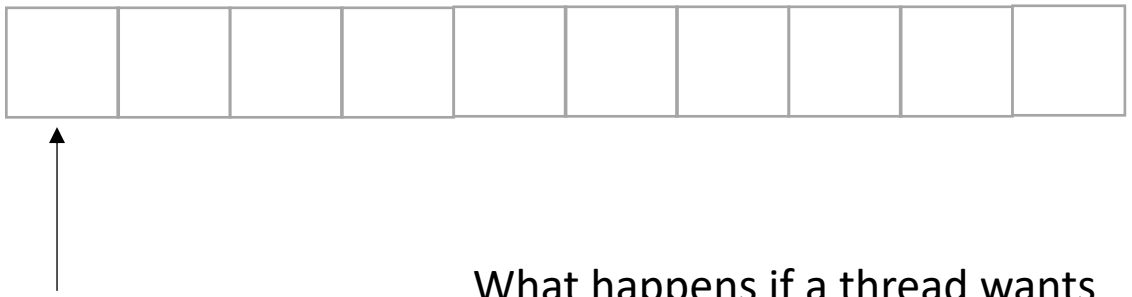
Implementation



What happens if a thread wants to add an element?

Think sequentially:

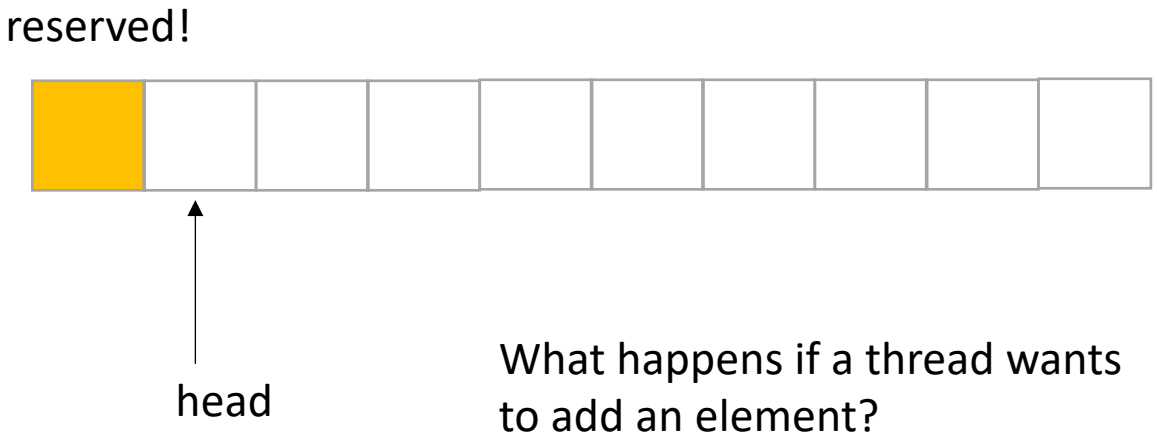
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment head

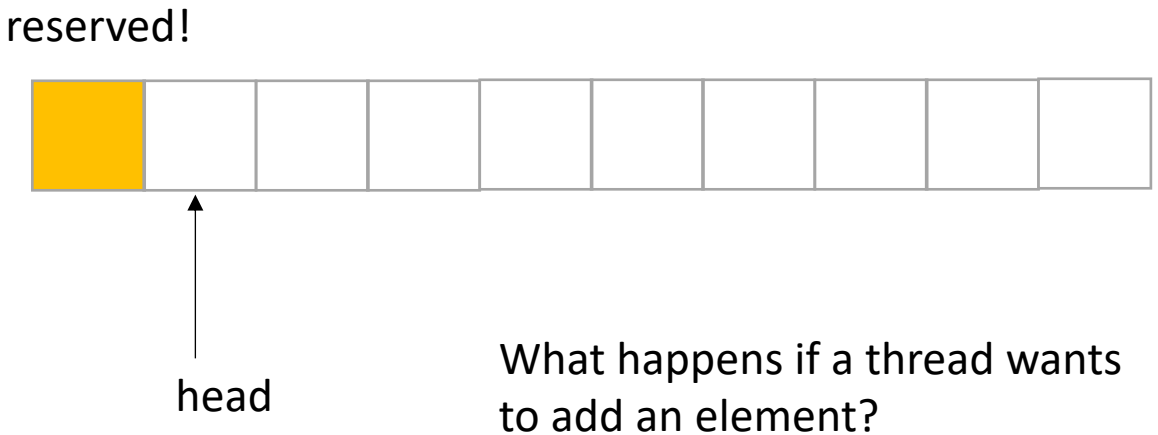
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment tail

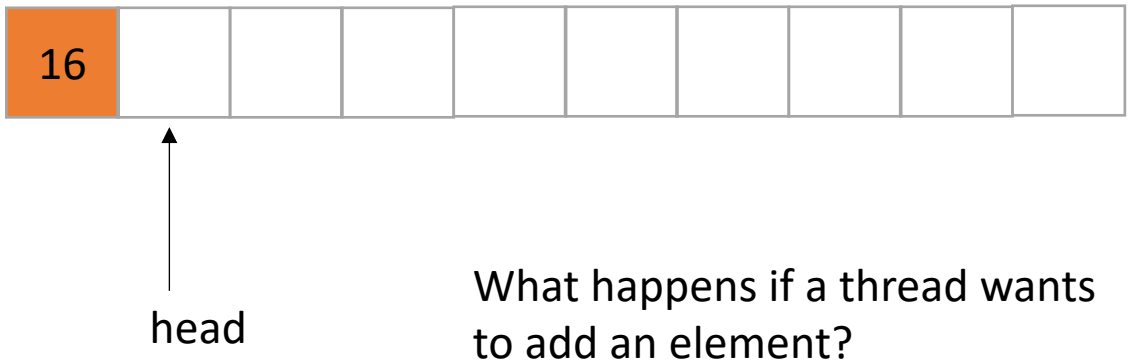
Implementation



What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment head
* add the element

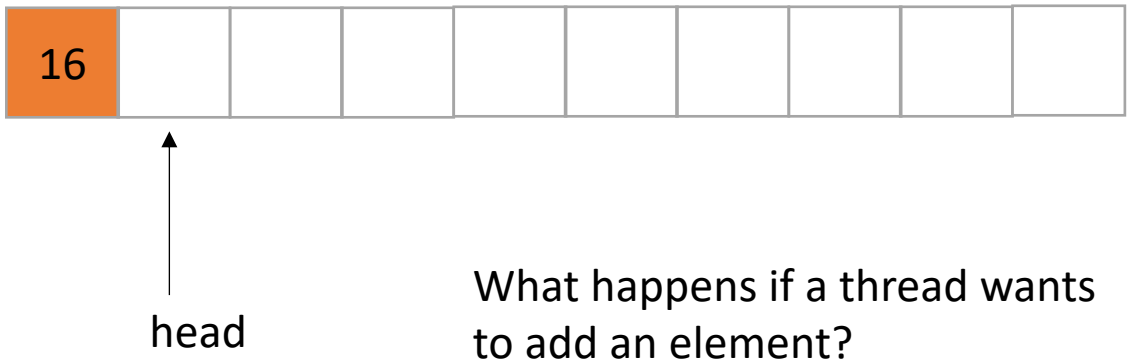
Implementation



What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment head
* add the element

Implementation

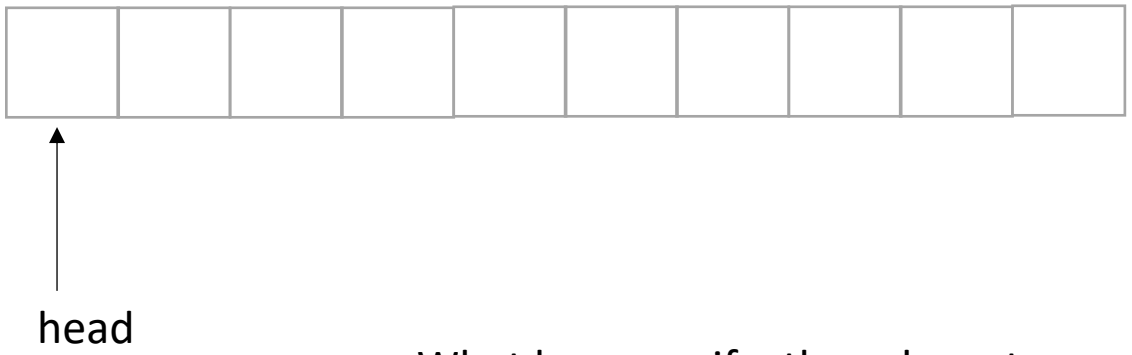


What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment head
* add the element

done!

Implementation

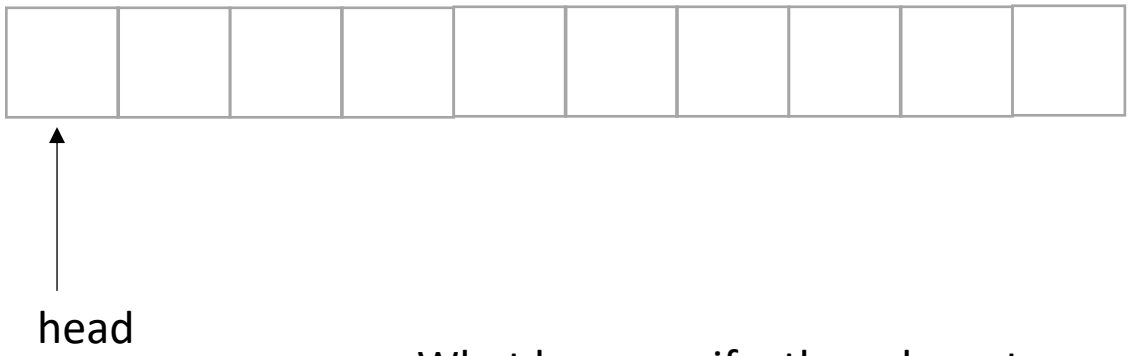


What happens if a thread wants
to add an element?

Think concurrently:

Two threads cannot reserve the same space!
We've seen this before

Implementation

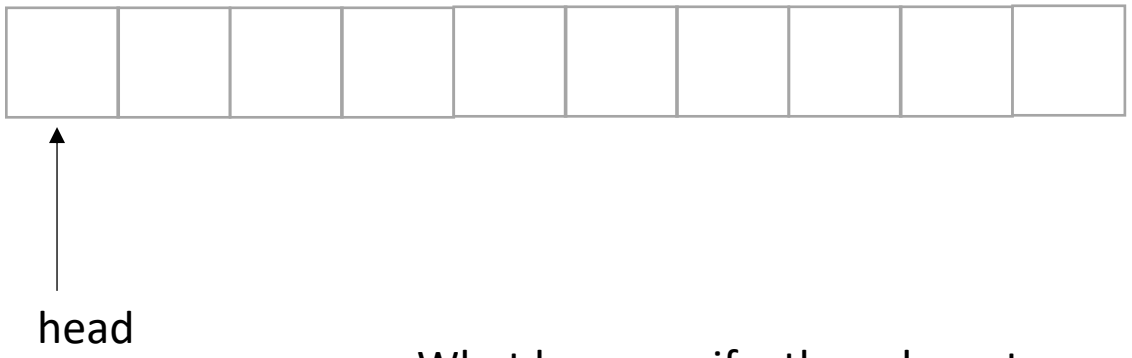


What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&head, 1);
```

Implementation



Thread 0:
enq(6);

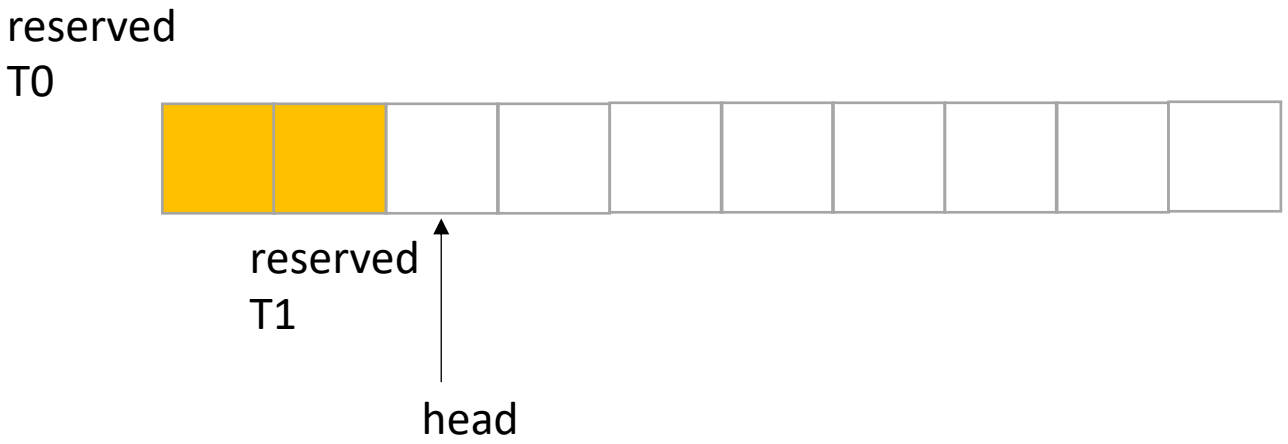
Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&head, 1);
```

Implementation



Thread 0:
enq(6);

Thread 1:
enq(7);

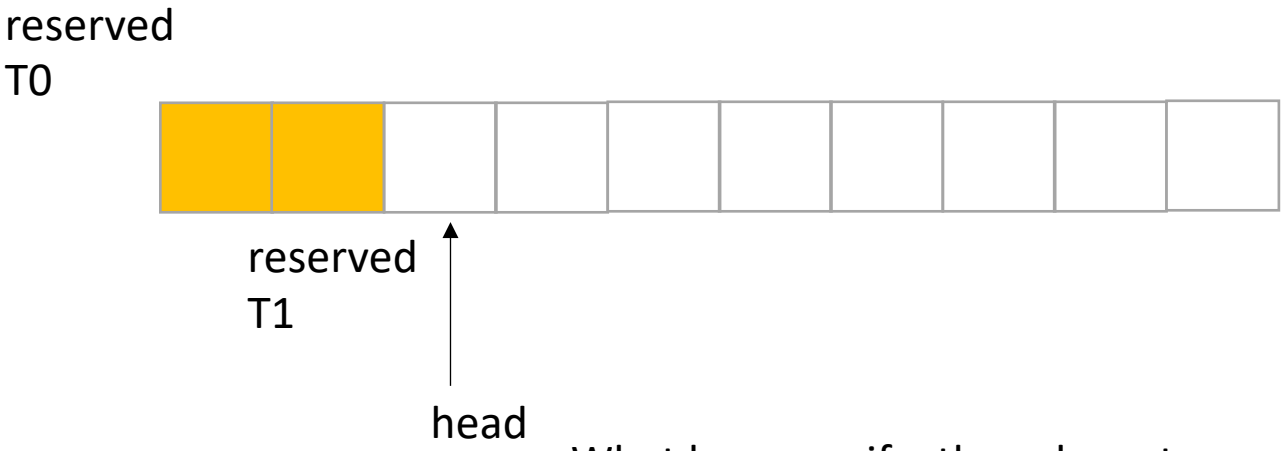
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&head, 1);
```

Implementation

does it matter which order threads add their data?



What happens if a thread wants to add an element?

Think concurrently:

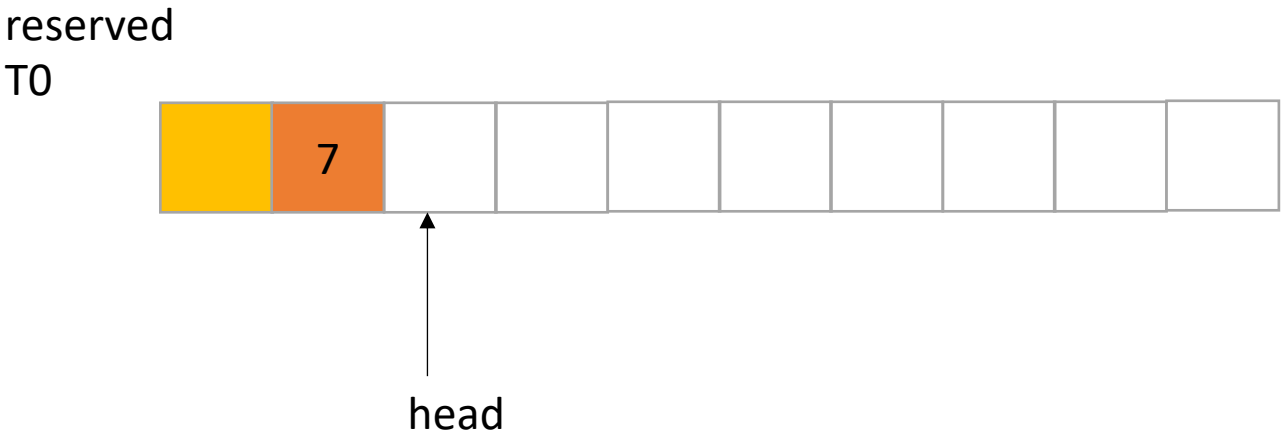
```
reserved_index = atomic_fetch_add(&head, 1);
```

Thread 0:
enq(6);

Thread 1:
enq(7);

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

What happens if a thread wants to add an element?

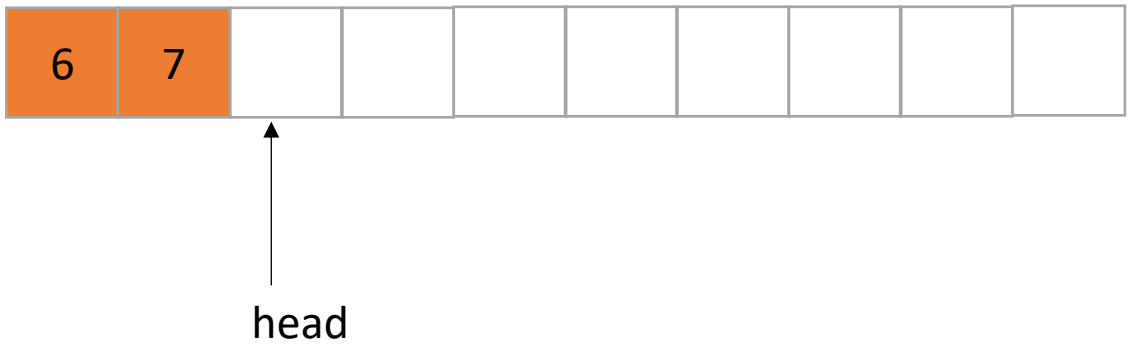
Think concurrently:

```
reserved_index = atomic_fetch_add(&head, 1);
```

Implementation

does it matter which order threads add their data? No! Because there are no deqs!

reserved
T0



Thread 0:
enq(6);

Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

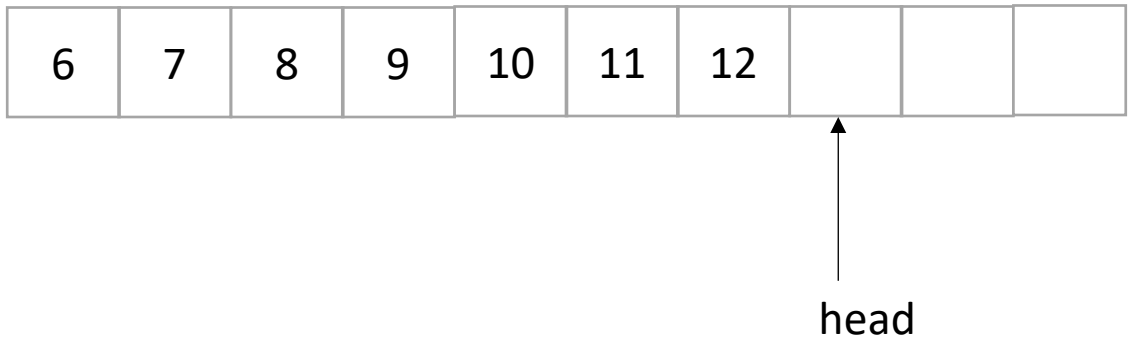
```
reserved_index = atomic_fetch_add(&head, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int head;
    int list[SIZE];

  public:
    InputOutputQueue() {
        head = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&head, 1);
        list[reserved_index] = x;
     }

     int size() {
        return head.load();
     }
 }
```

How to protect against overflows?

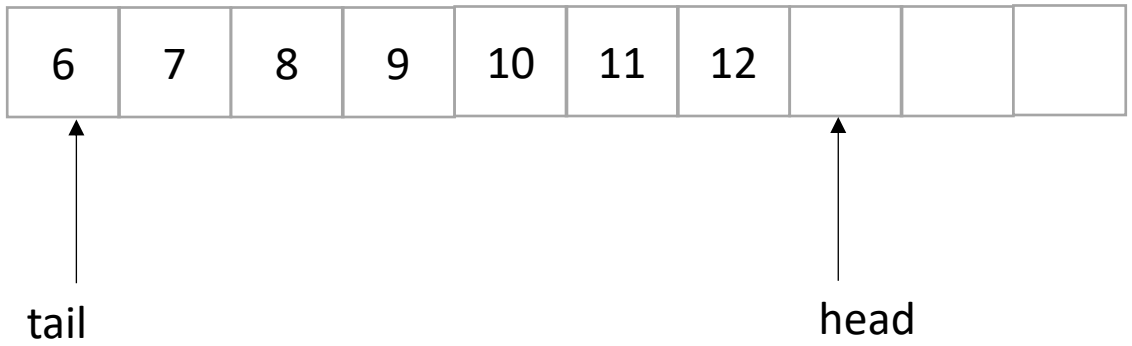
What about Input?

- Now we only do deqs



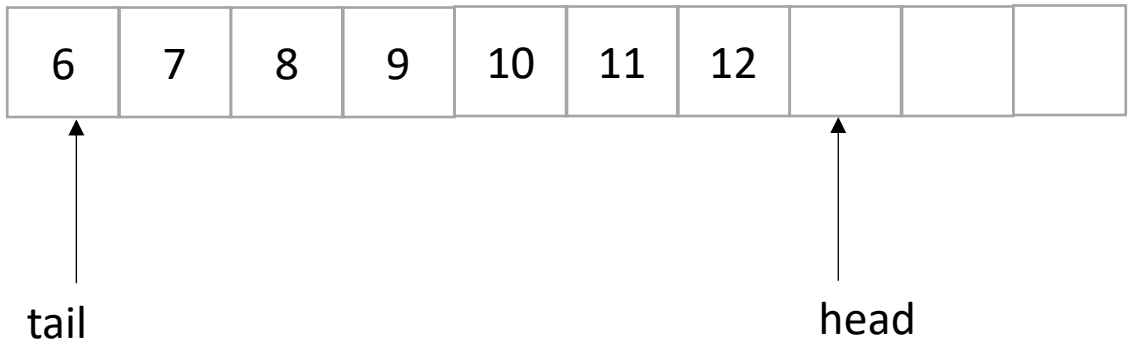
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



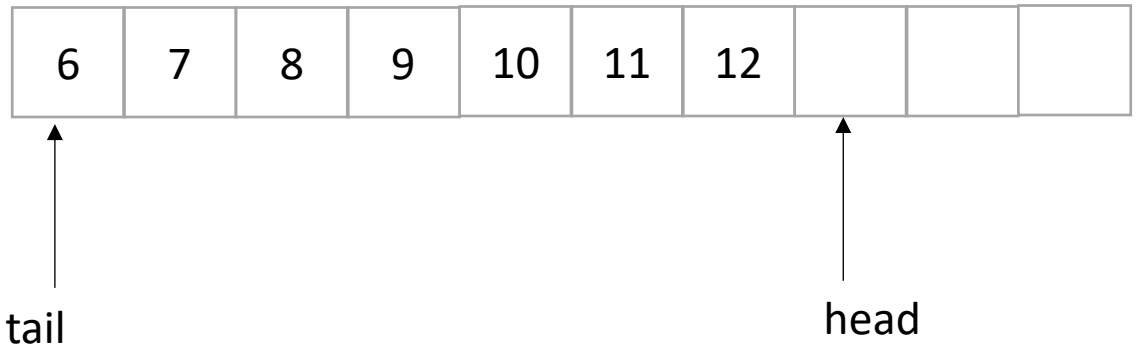
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&tail, 1);
```

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

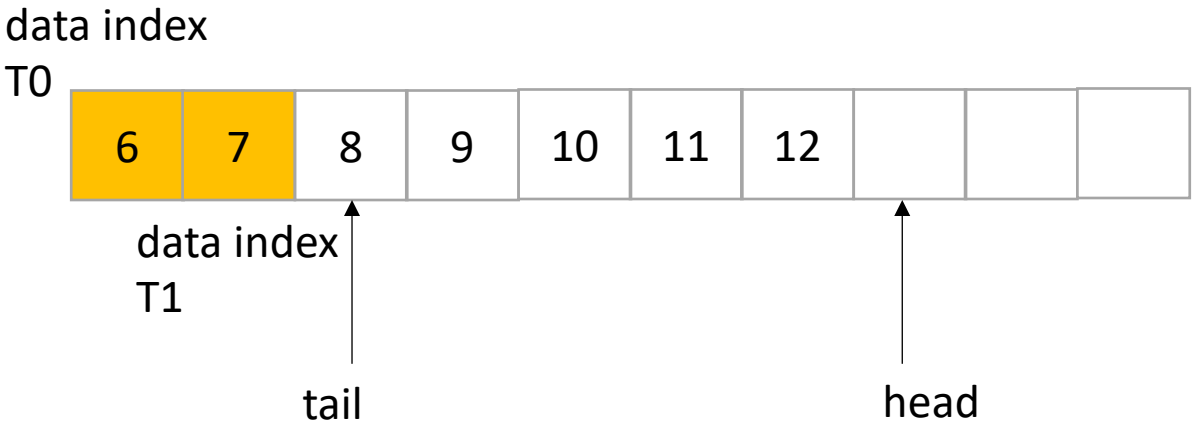
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&tail, 1);
```

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

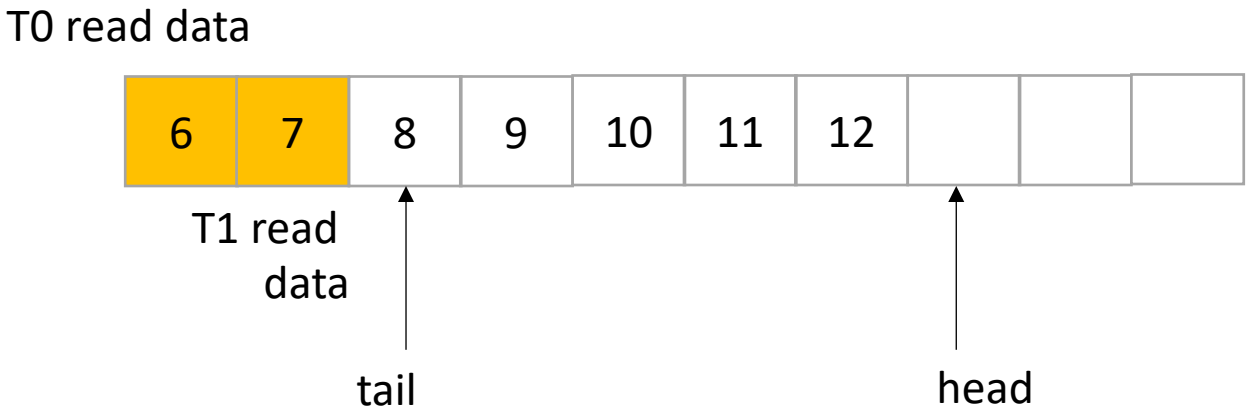
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&tail, 1);
```

What about Input?

- Now we only do deqs



Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

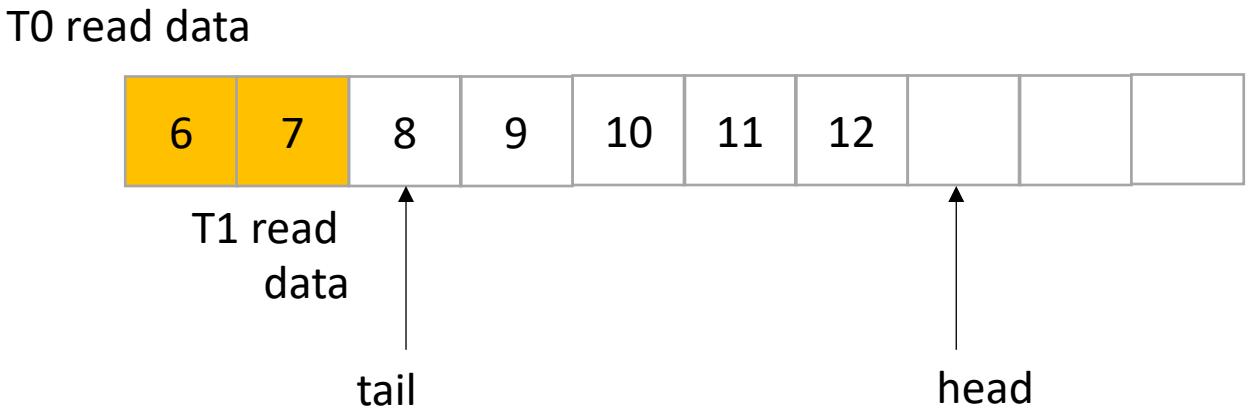
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&tail, 1);
```

What about Input?

- Now we only do deqs



How to implement a stack?

Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&tail, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int list[SIZE];

  public:
    InputOutputQueue() {
        head = tail = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&head, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&tail, 1);
       return list[reserved_index];
     }

     int size() {
        return head.load();
     }
 }
```

```
class InputOutputQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int list[SIZE];

  public:
    InputOutputQueue() {
        head = tail = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&head, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&tail, 1);
       return list[reserved_index];
     }

     int size() {
        return head.load();
     }
 }
```

how about size?

```
class InputOutputQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int list[SIZE];

  public:
    InputOutputQueue() {
        head = tail = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&head, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&tail, 1);
       return list[reserved_index];
     }

     int size() {
        return head.load() - tail.load();
     }
 }
```

how about size?

how do we reset?

```
class InputOutputQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int list[SIZE];

  public:
    InputOutputQueue() {
        head = tail = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&head, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&tail, 1);
       if (reserved_index > SIZE) throw exception
       return list[reserved_index];
     }

     int size() {
        return head.load() - tail.load();
     }
 }
```

how about size?

how do we reset?

does the list need to be atomic?

Schedule

- Finish up linked-list

- **Concurrent Queues**
  - Input/Output Queues
  - **Synchronous Producer/Consumer Queue**
  - Async Producer/Consumer Queue

5 minute break

Producer Consumer Queues

- 1 enq, 1 deq
  - enq’er cannot deq
  - deq cannot enq

- Example: printf:
  - your program equeues values to print
  - the terminal process dequeues values and prints them

Synchronous Producer Consumer Queues

- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

Synchronous Producer Consumer Queues

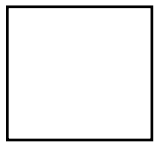
- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

- enq does not return until value is deq’ed

Synchronous Producer Consumer Queues

Producer Thread

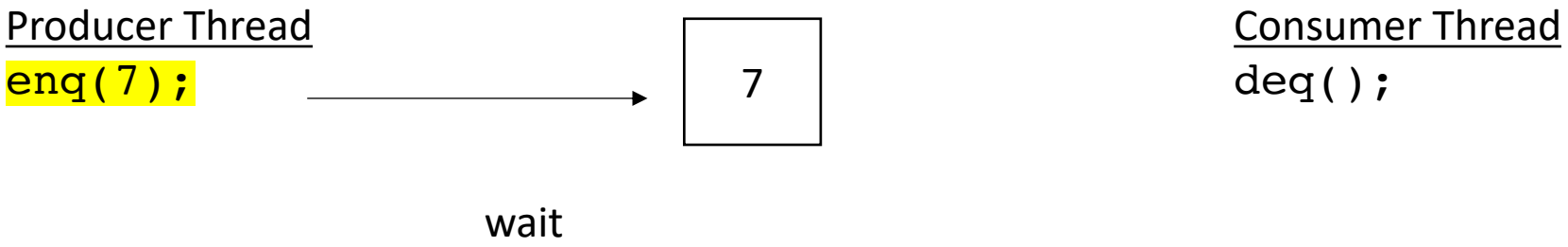
```
enq(7);
```



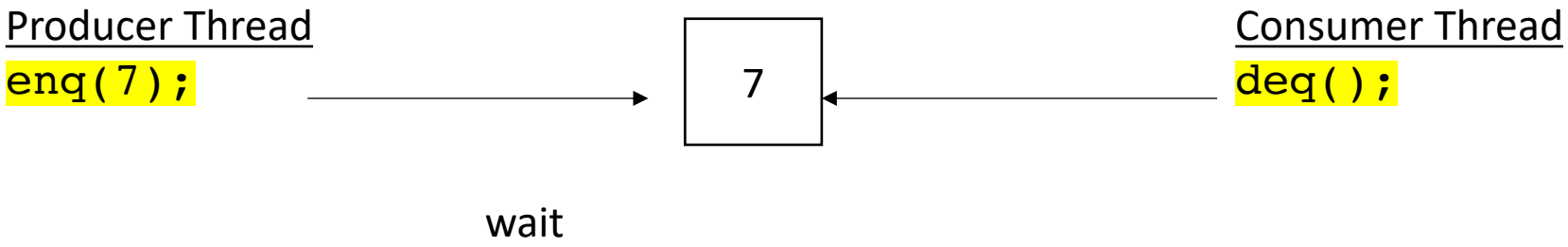
Consumer Thread

```
deq();
```

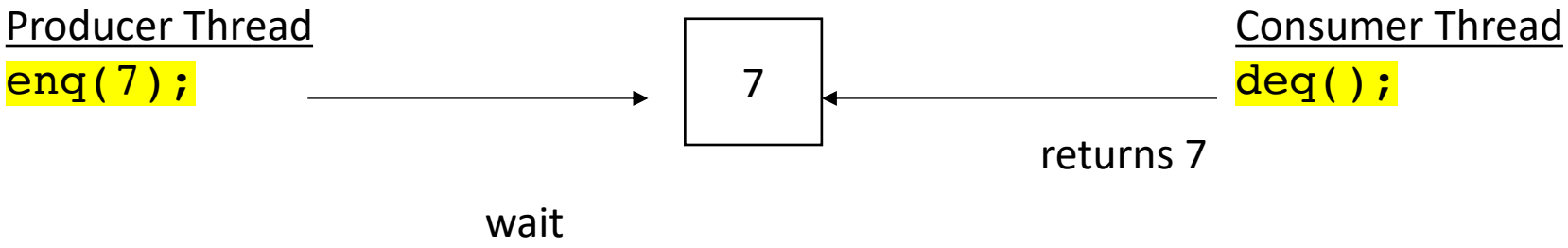
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues



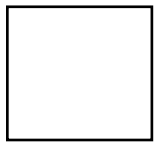
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

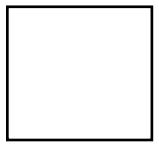
```
deq();
```

both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
sleep();
enq(7);
```



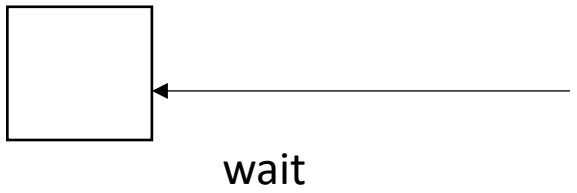
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
sleep();
enq(7);
```

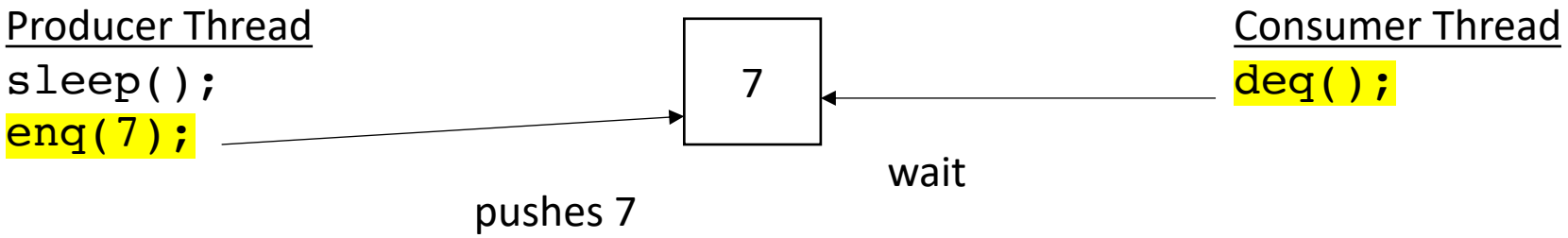


wait

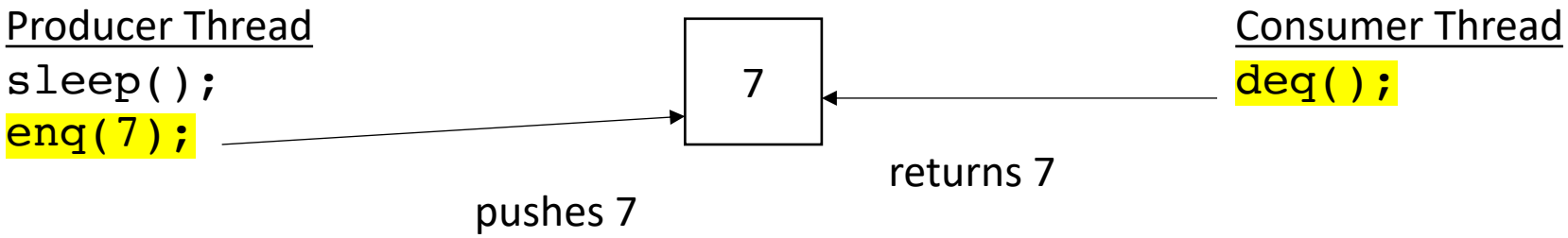
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

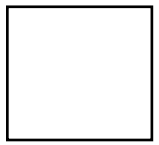


They both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```

Consumer Thread

```
deq();
```

can the consumer just read?

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```

Consumer Thread

```
deq();
```

can the consumer just read?
Needs to wait for a value to appear

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```

Consumer Thread

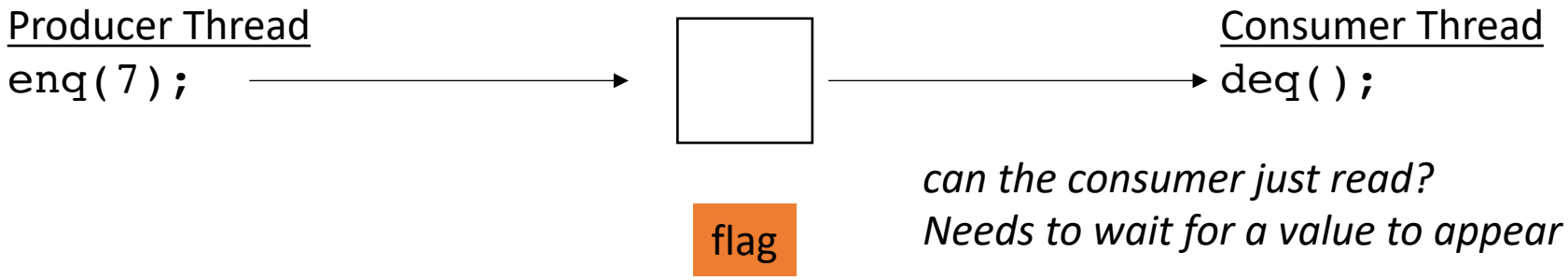
```
deq();
```

flag

can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

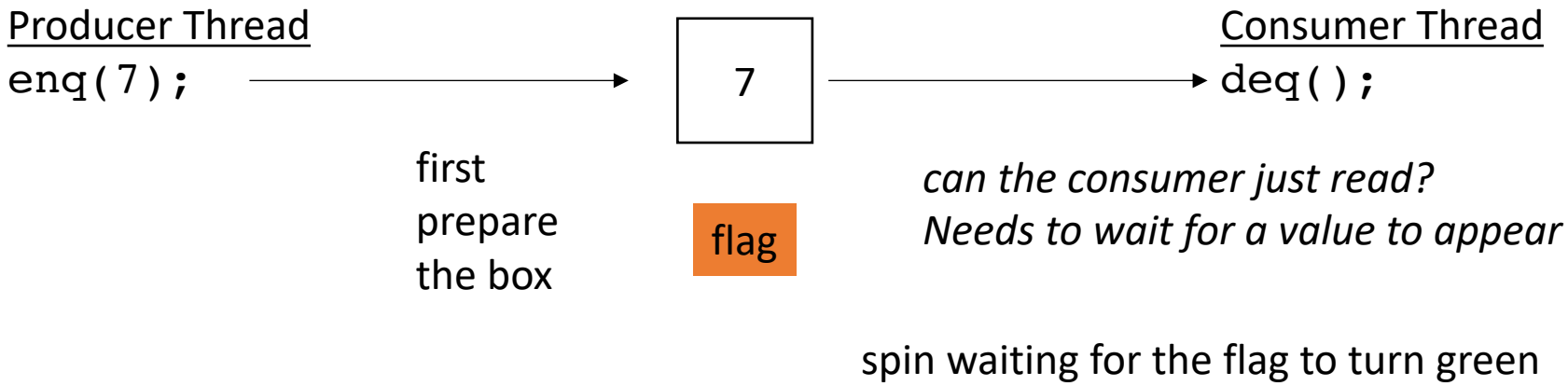
Synchronous Producer Consumer Queues



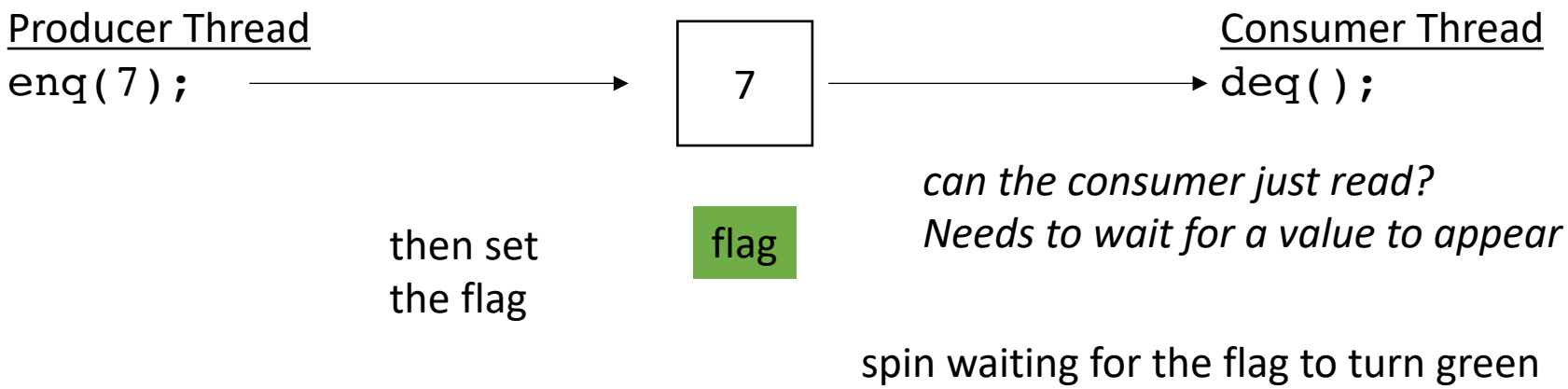
can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

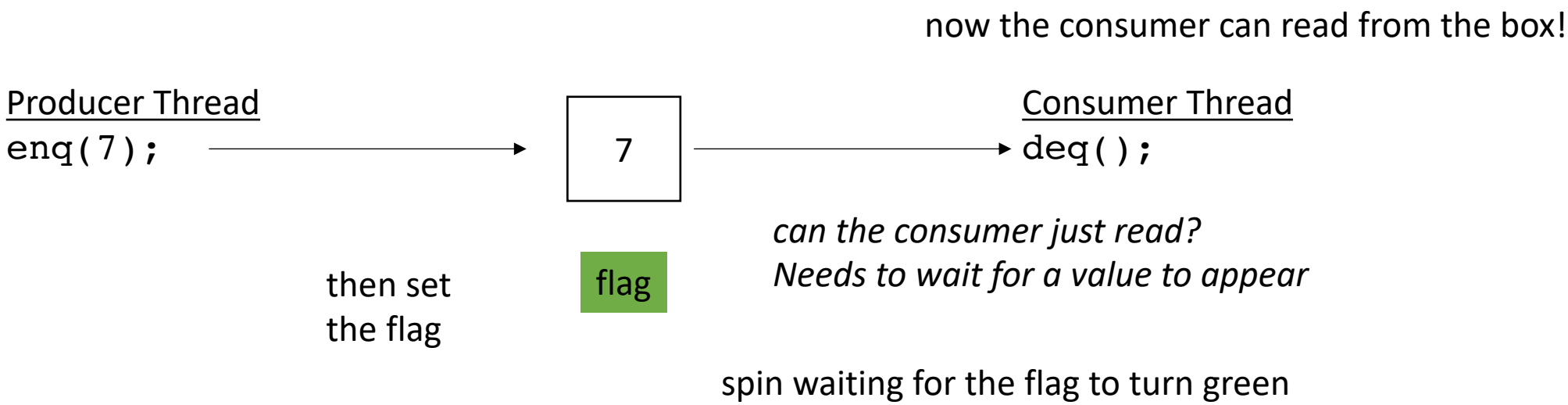
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues



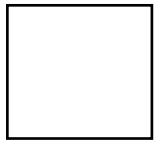
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

Consumer Thread

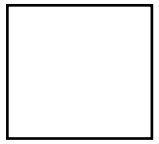
```
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

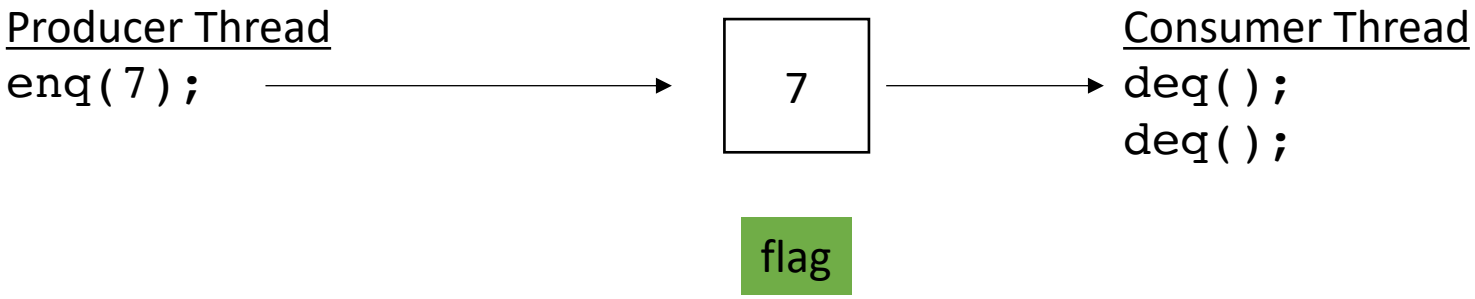
Consumer Thread

```
deq();
deq();
```

what happens when there are two deqs?

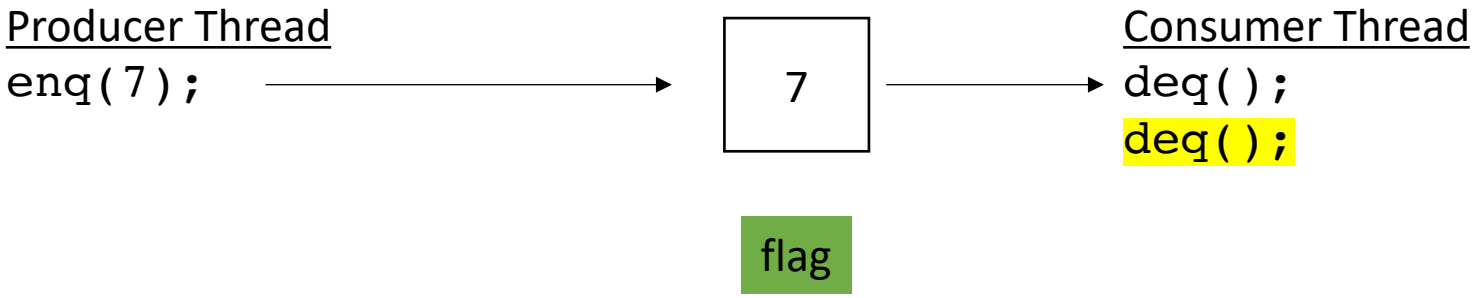
```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues




```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues

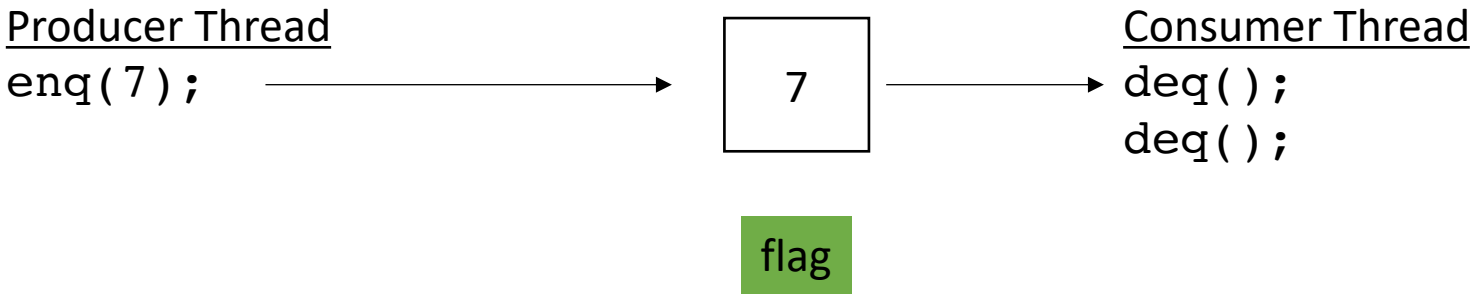



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

what happens in the next deq?

How to fix?

Synchronous Producer Consumer Queues

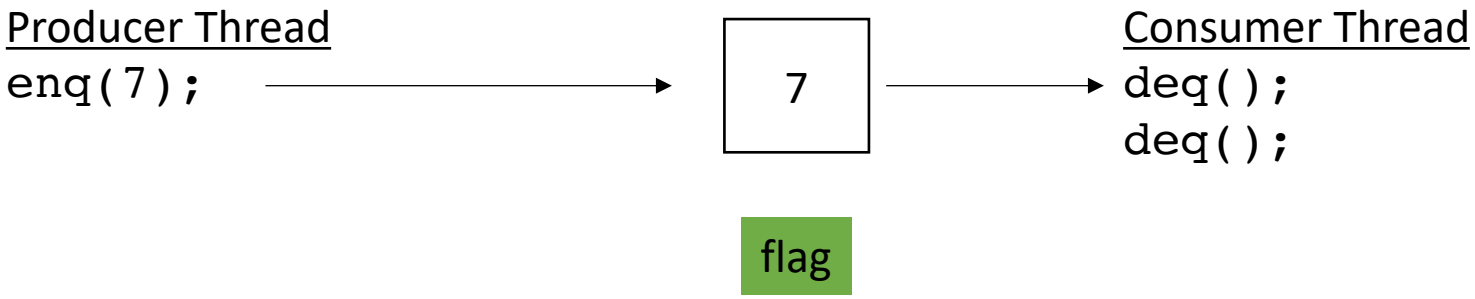



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

what happens in the next deq?

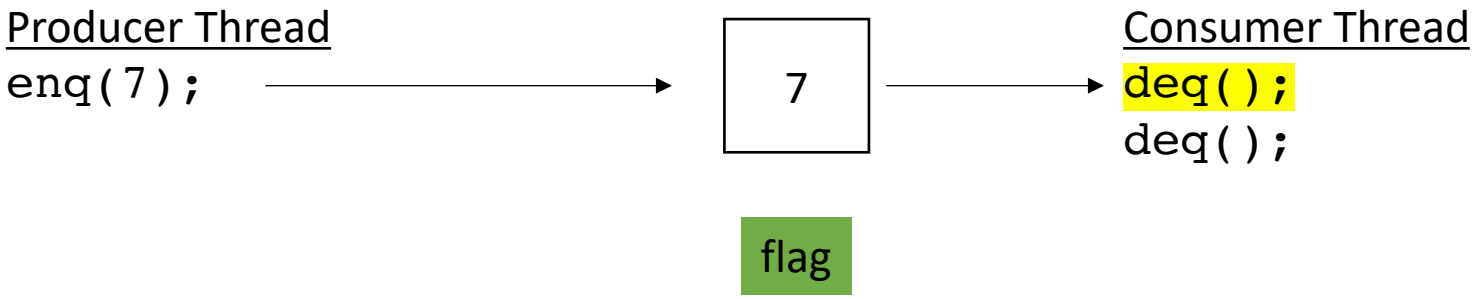
How to fix?

Synchronous Producer Consumer Queues



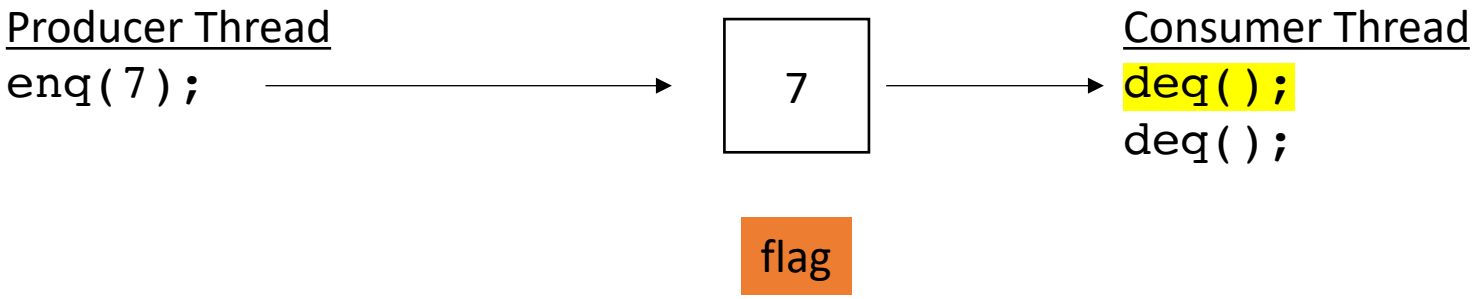

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues



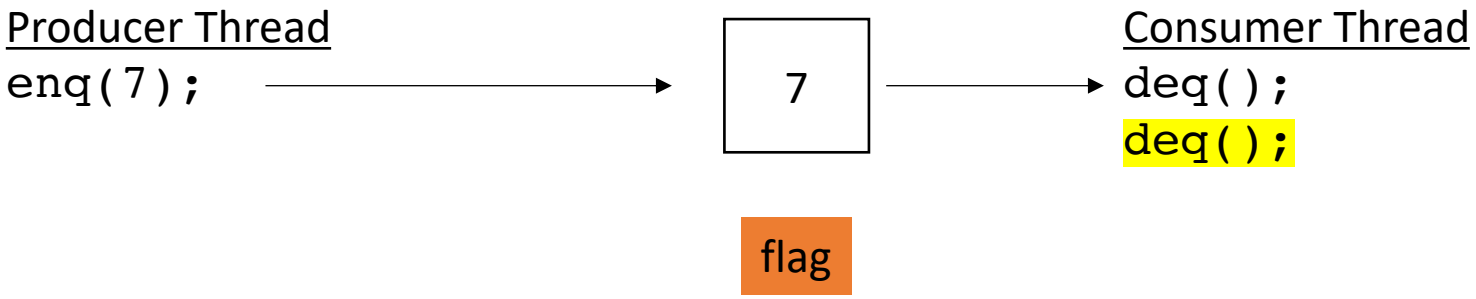

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues




```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues




```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

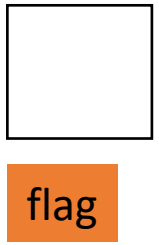
waiting like we are supposed to

Synchronous Producer Consumer Queues

reset (now with extra enq)

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

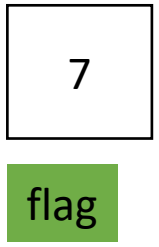
extra enq

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

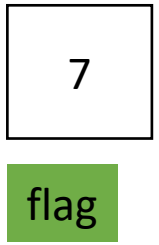
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

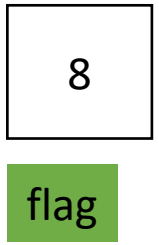
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

7 was dropped!

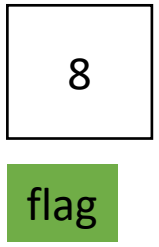
how to fix?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

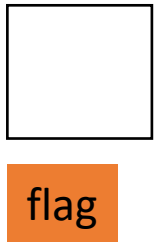
7 was dropped!

how to fix?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

reset



Producer Thread

```
enq(7);
enq(8);
```

Consumer Thread

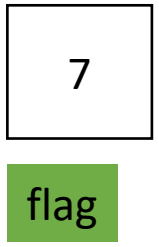
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

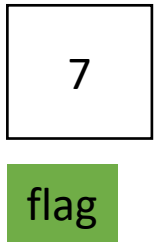
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

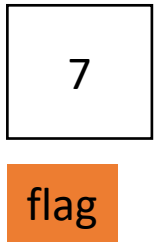
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

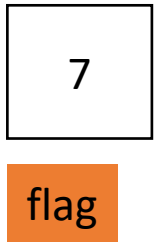
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

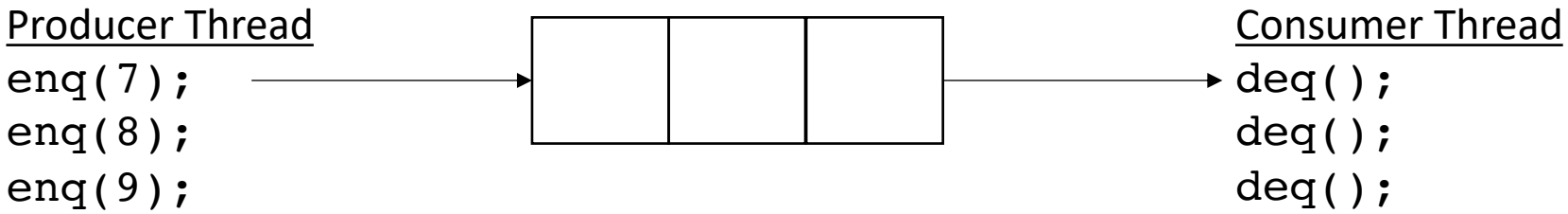
Schedule

- Finish up linked-list

- **Concurrent Queues**
  - Input/Output Queues
  - Synchronous Producer/Consumer Queue
  - **Async Producer/Consumer Queue**

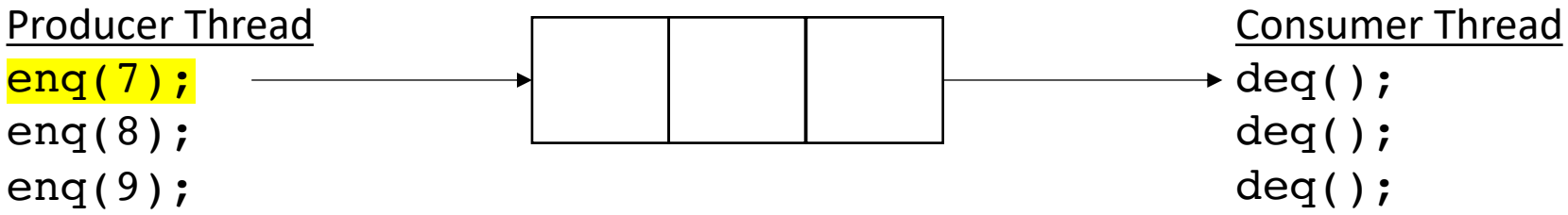
Producer Consumer Queues

- Asynchronous:



Producer Consumer Queues

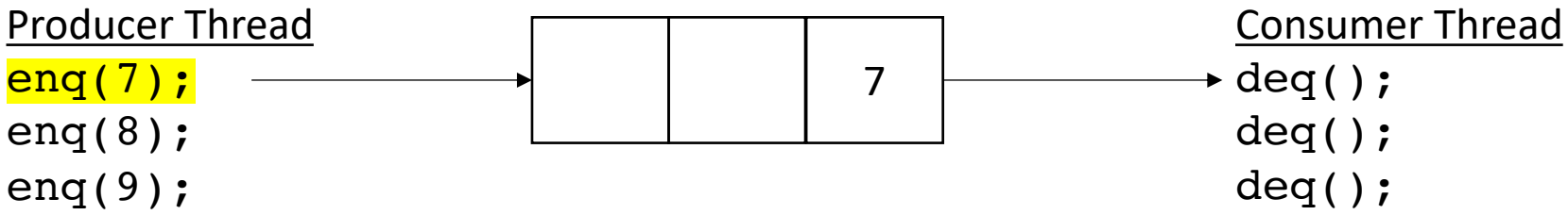
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

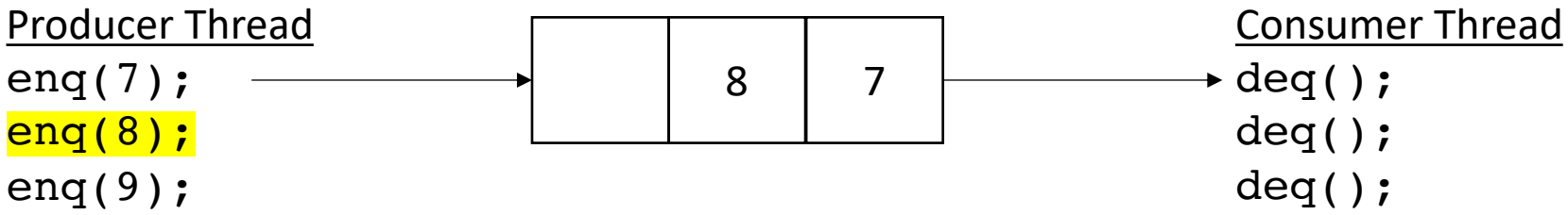
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

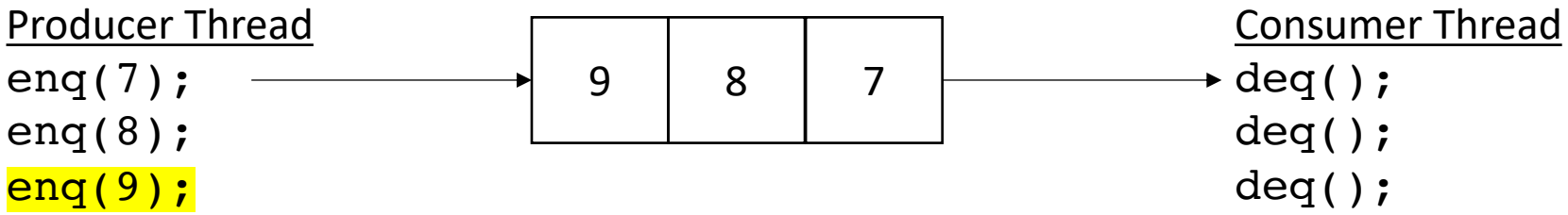
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

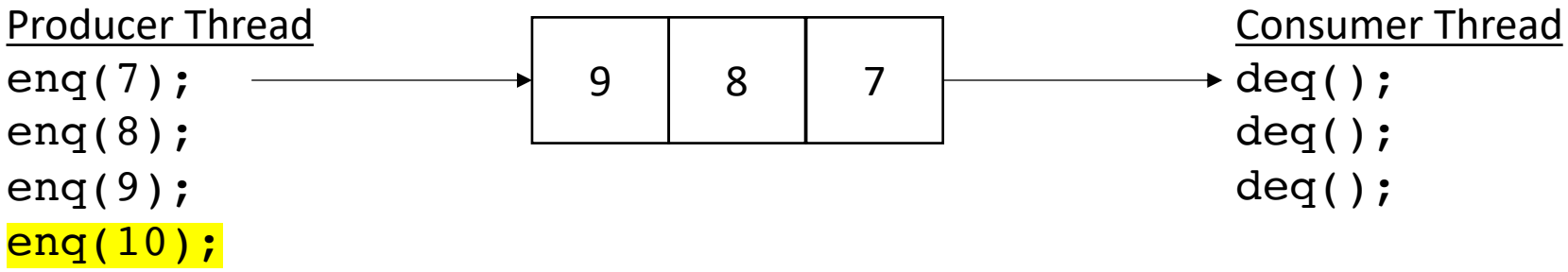
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

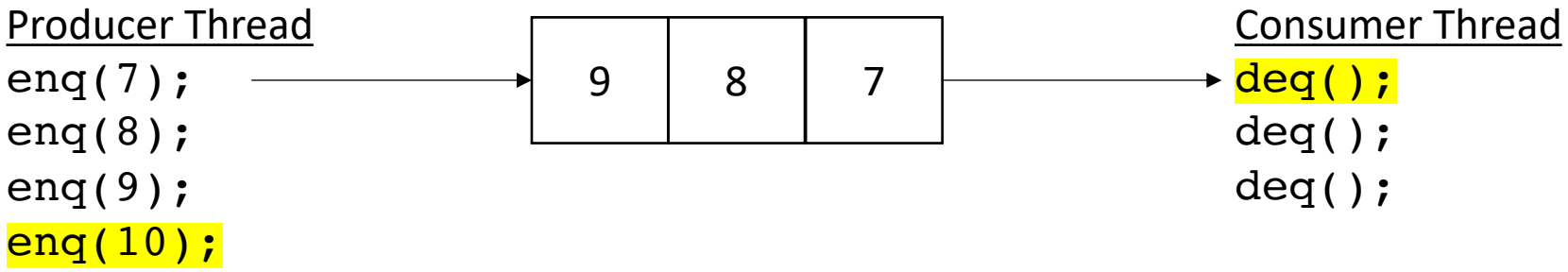


no waiting for producer (while there is room)

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



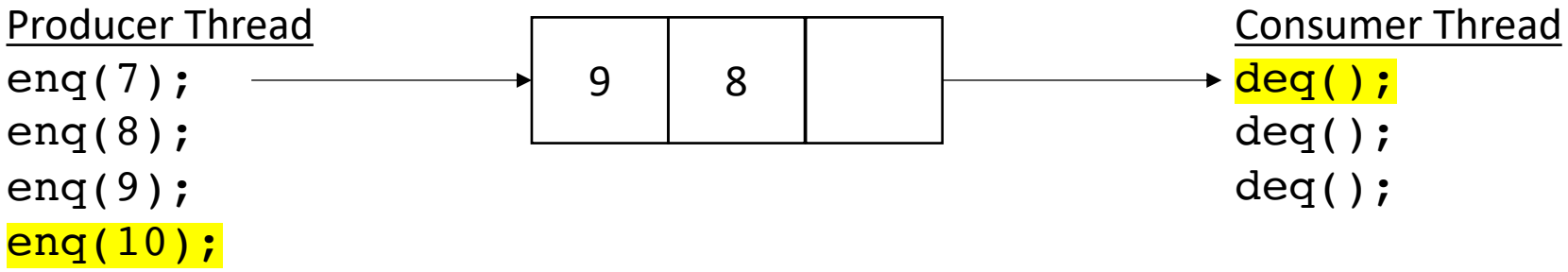
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



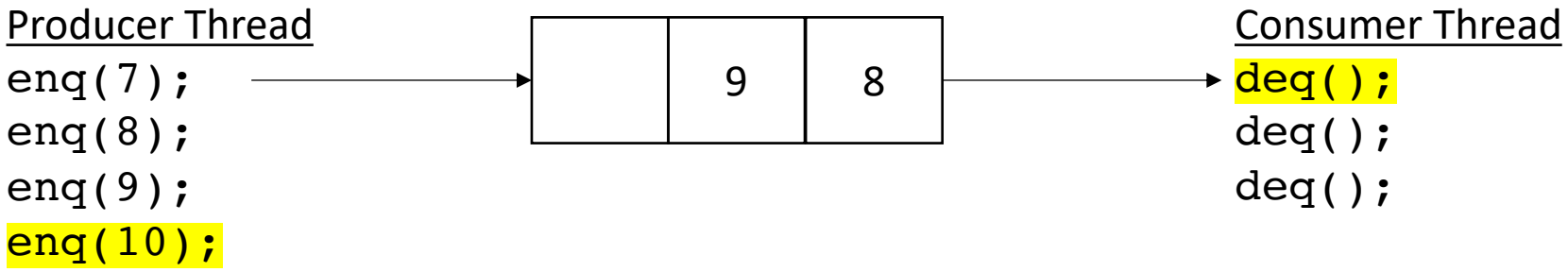
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



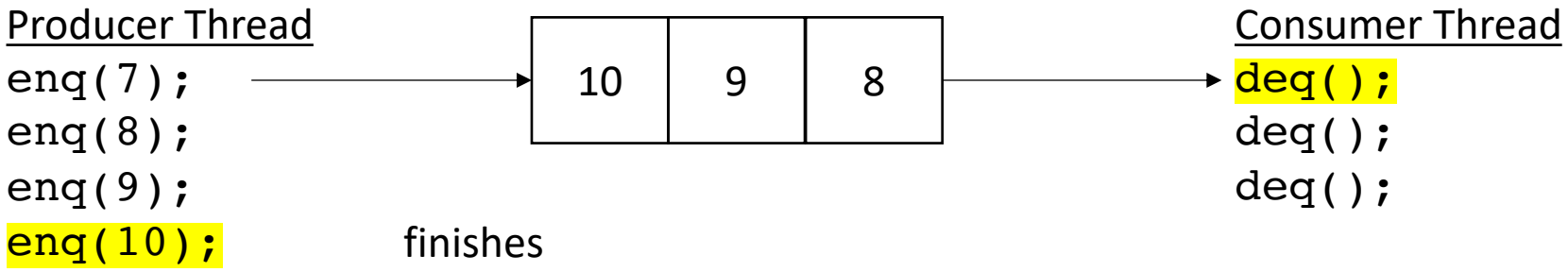
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



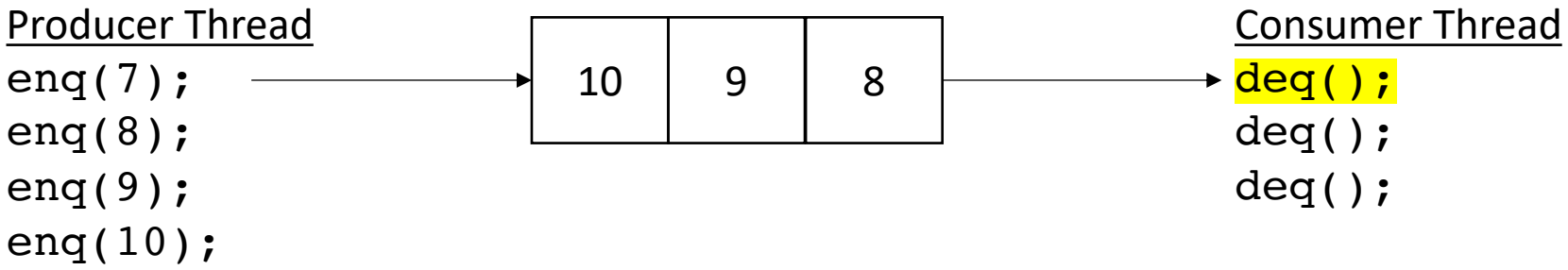
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



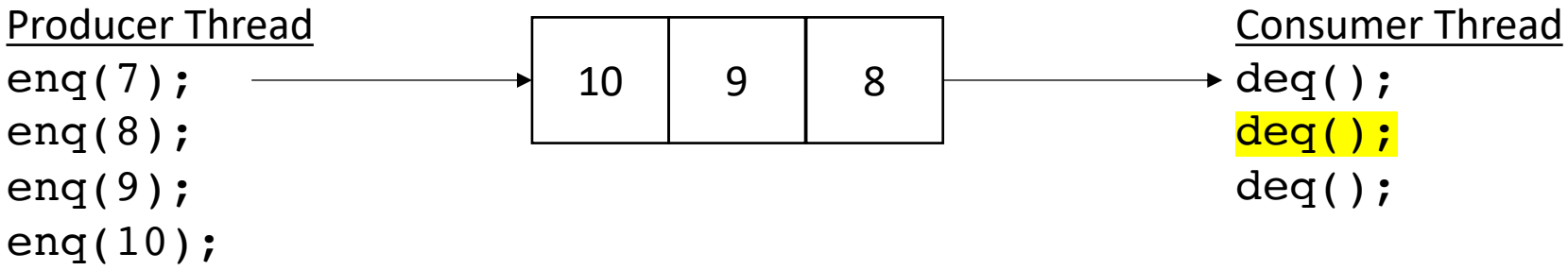
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



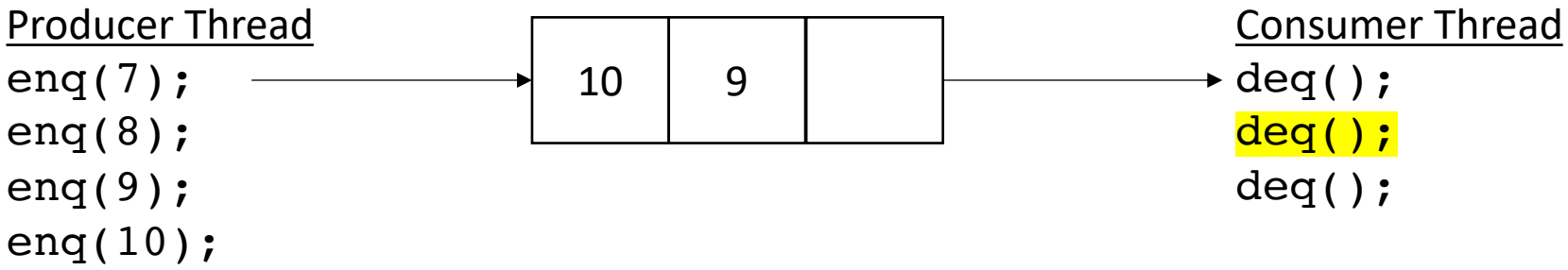
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



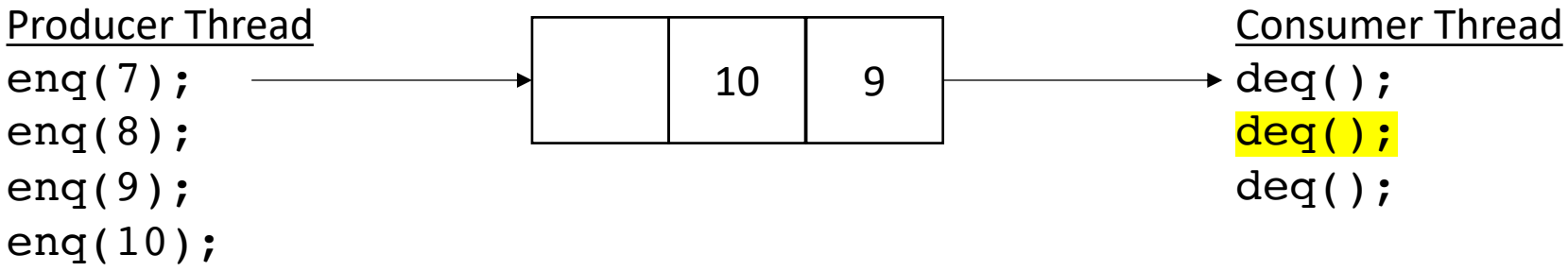
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



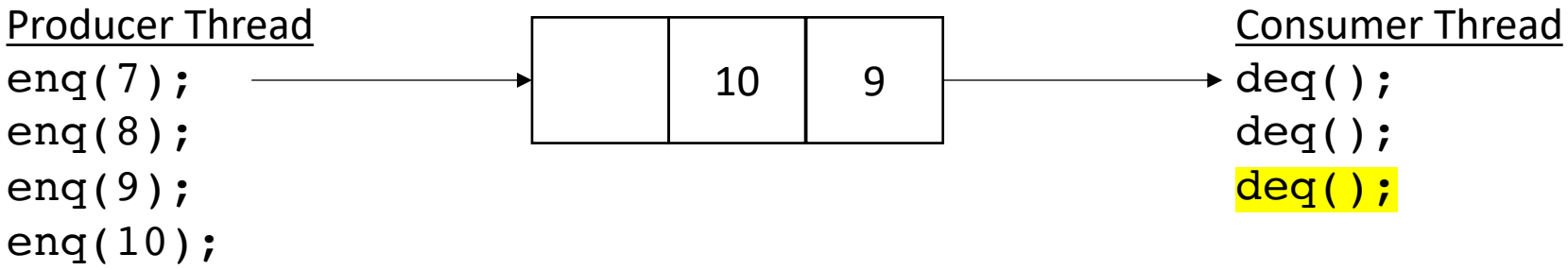
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

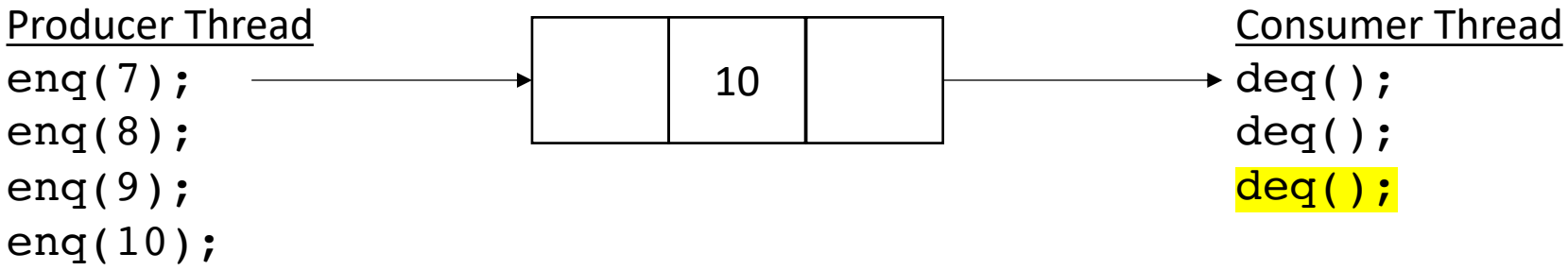
- Asynchronous:



returns 9

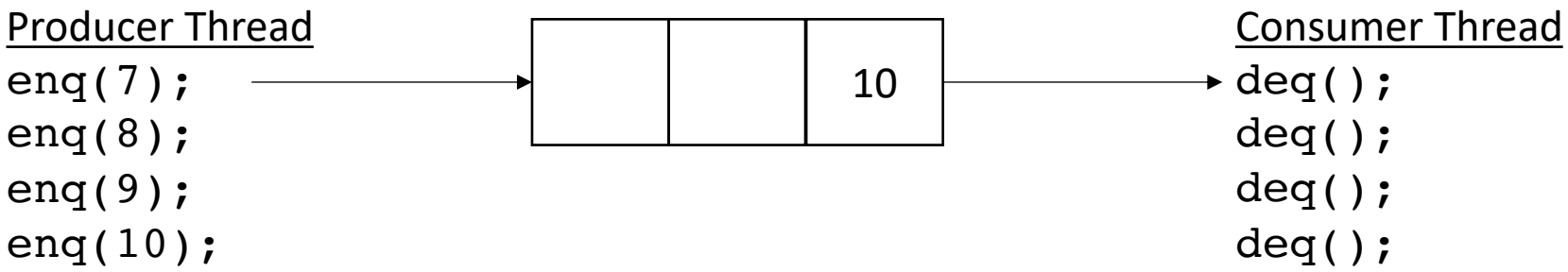
Producer Consumer Queues

- Asynchronous:



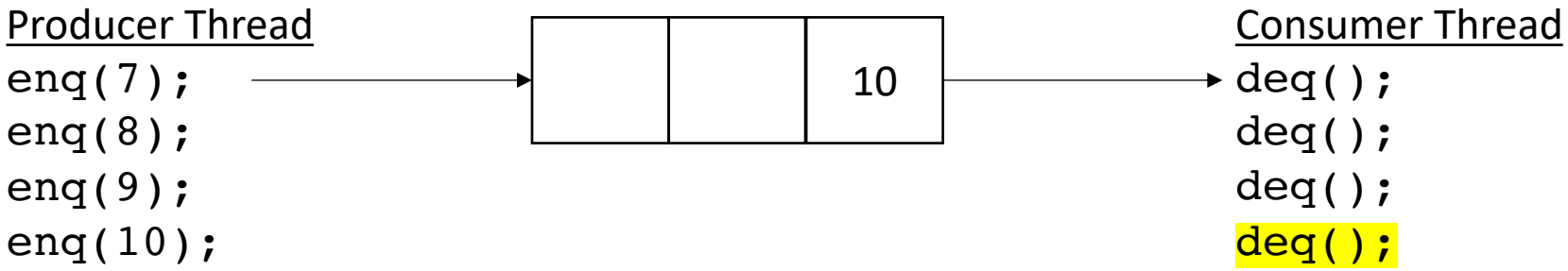
Producer Consumer Queues

- Asynchronous:



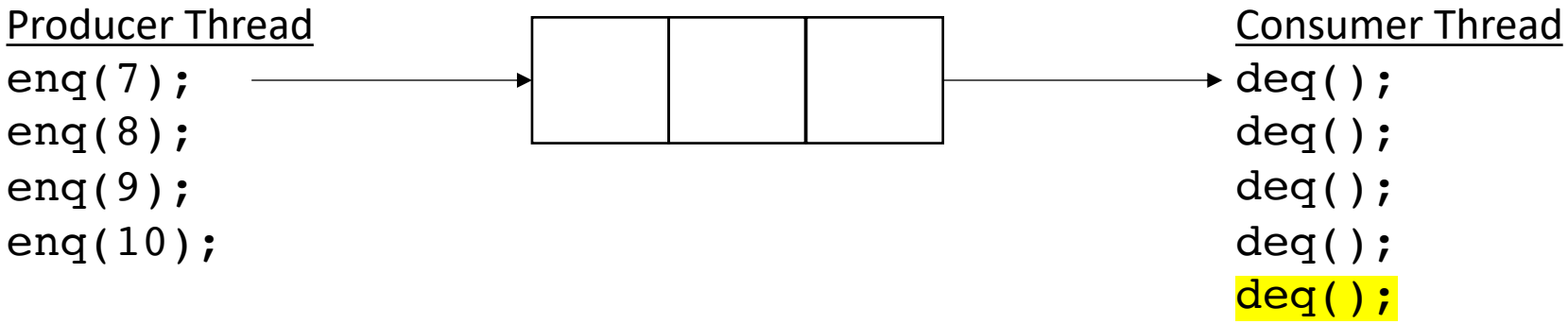
Producer Consumer Queues

- Asynchronous:



Producer Consumer Queues

- Asynchronous:



blocks when there is nothing in the queue

Producer Consumer Queues

- How do we implement it?

Producer Consumer Queues

- Start with a fixed size array

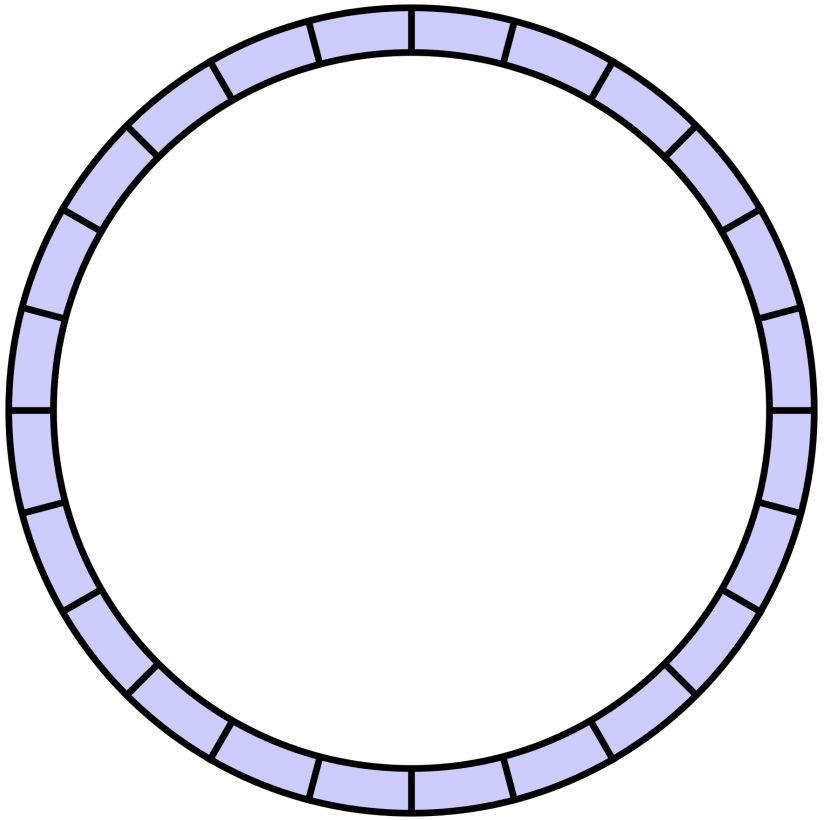
Producer Consumer Queues

- Start with a fixed size array

We will use what is called a *circular buffer method*

Producer Consumer Queues

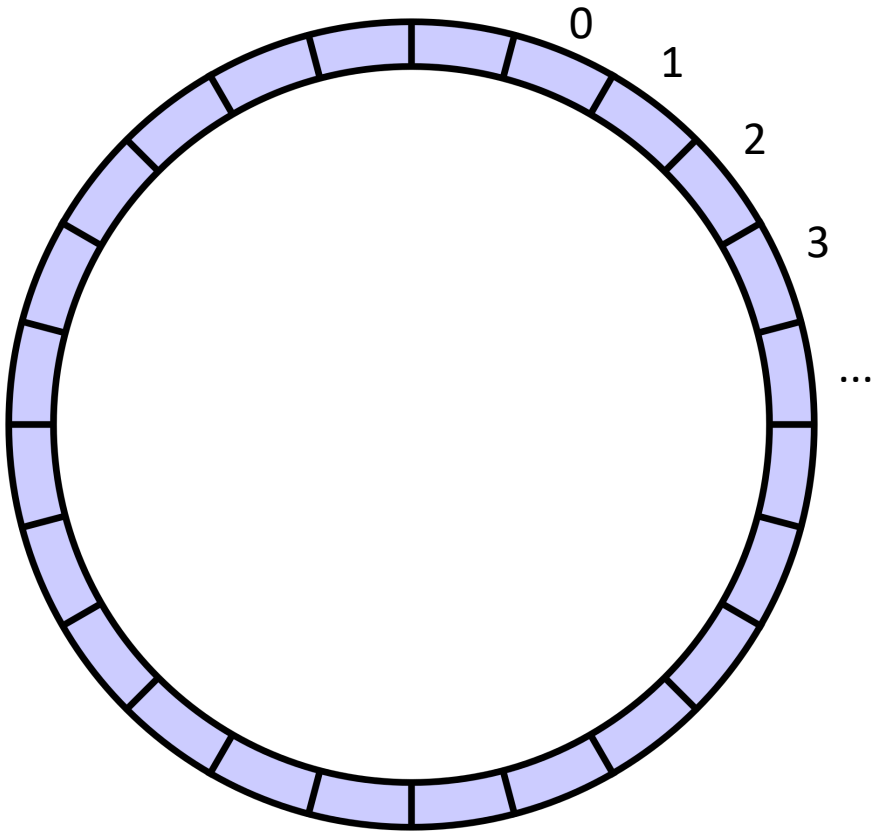
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

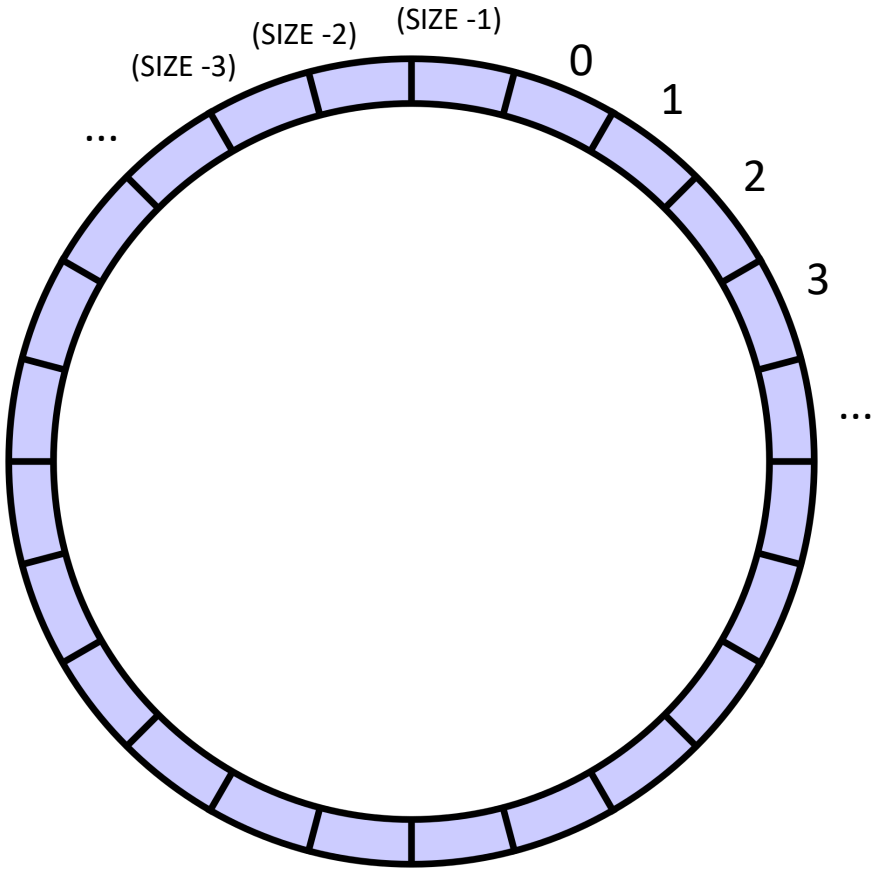
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

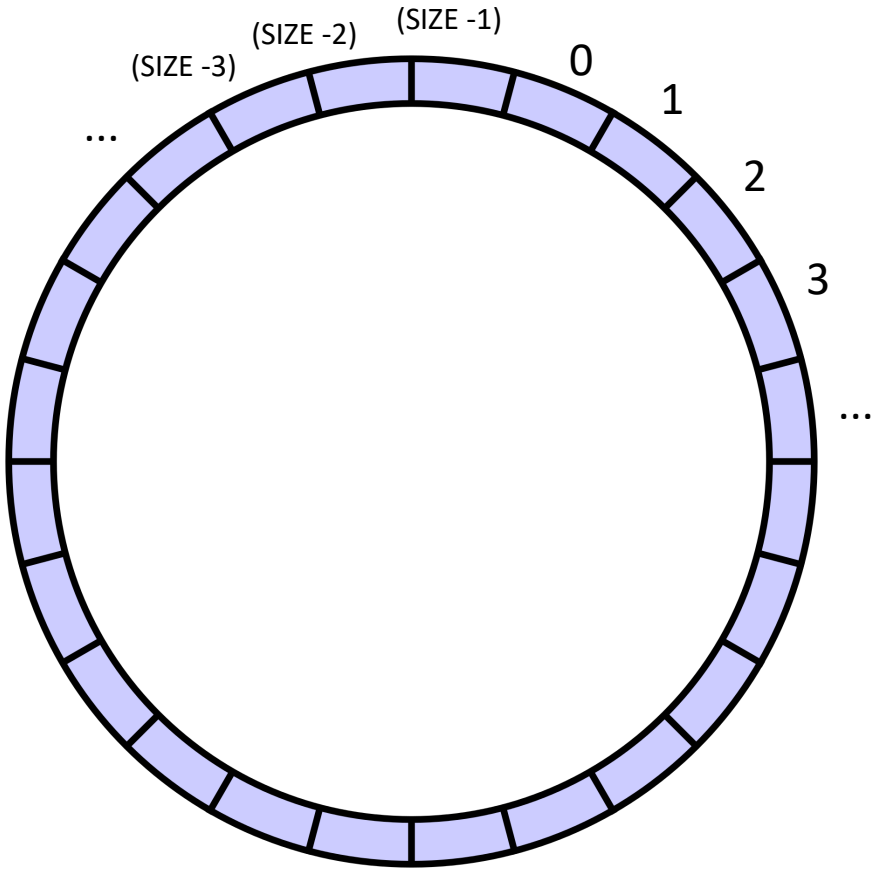


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

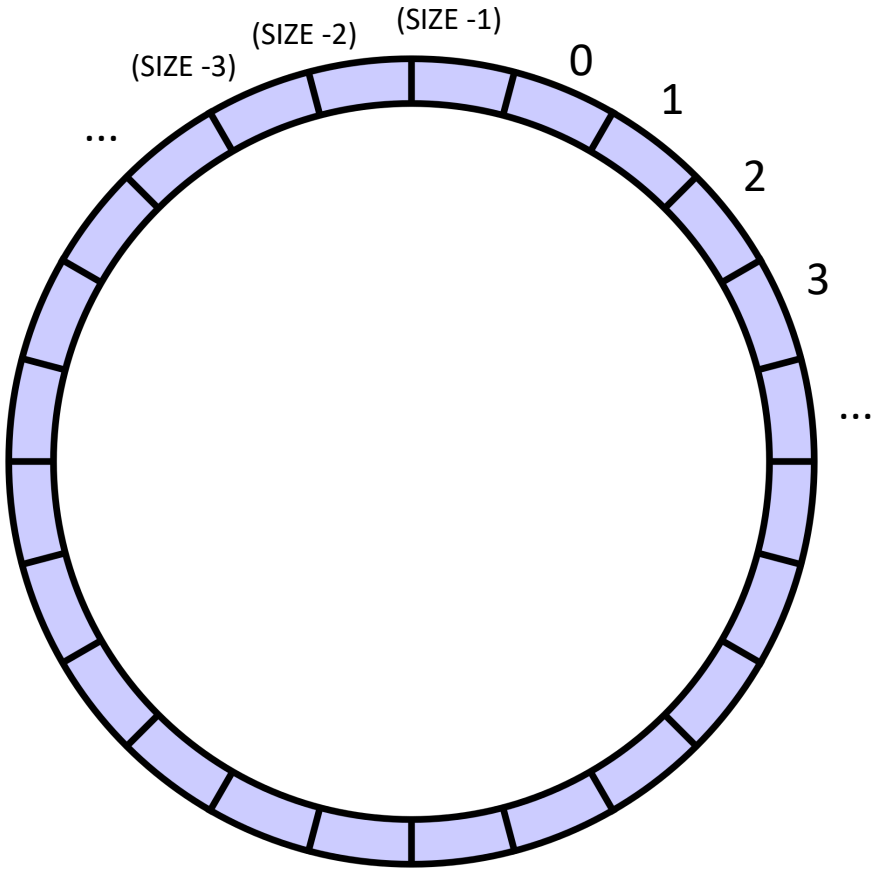
we will assume modular arithmetic:

if x = (SIZE - 1) then
x + 1 == 0;

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

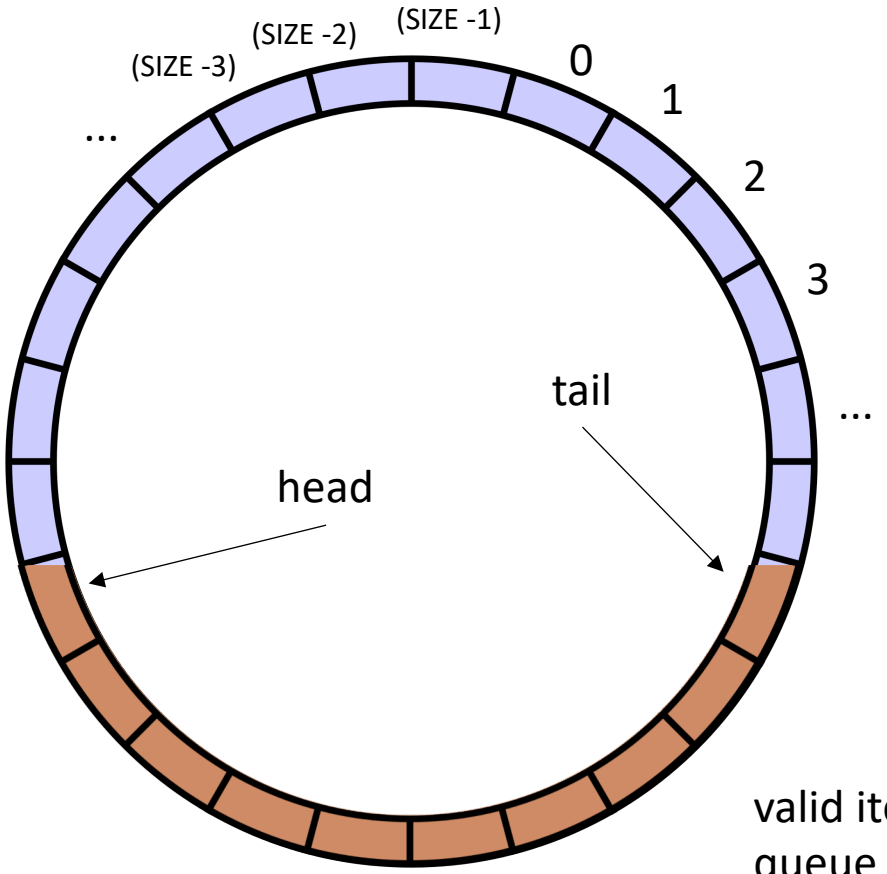
Two variables to keep track of where to deq and enq:

head and tail

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail:

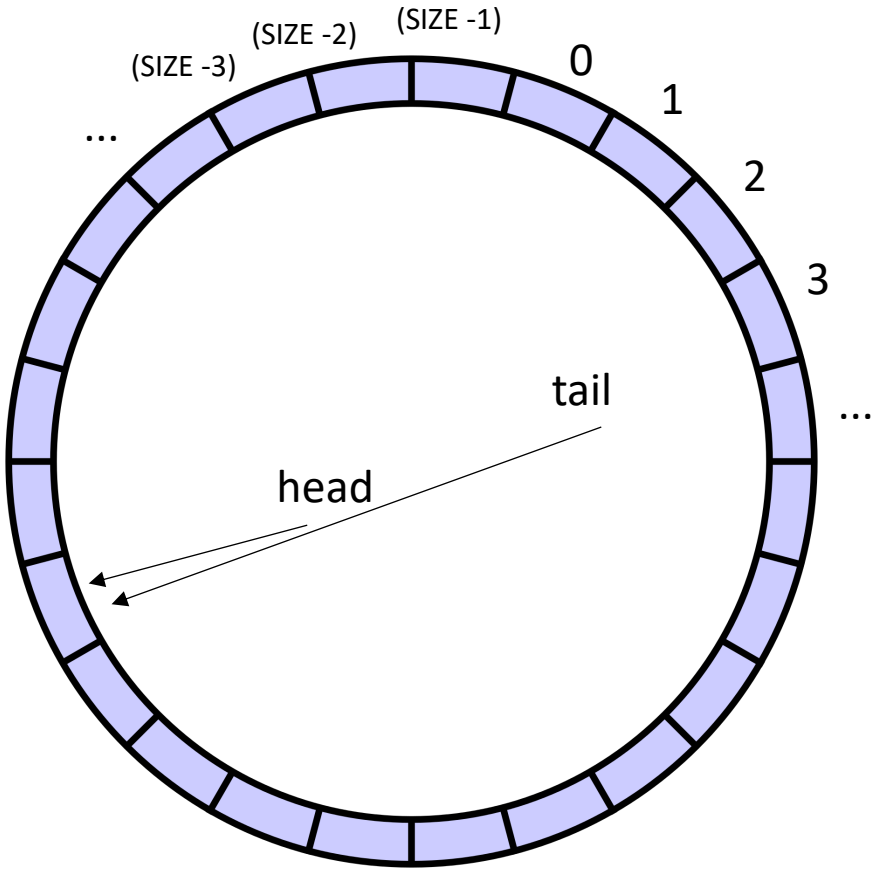
enq to the head, deq from the tail

valid items in the queue

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

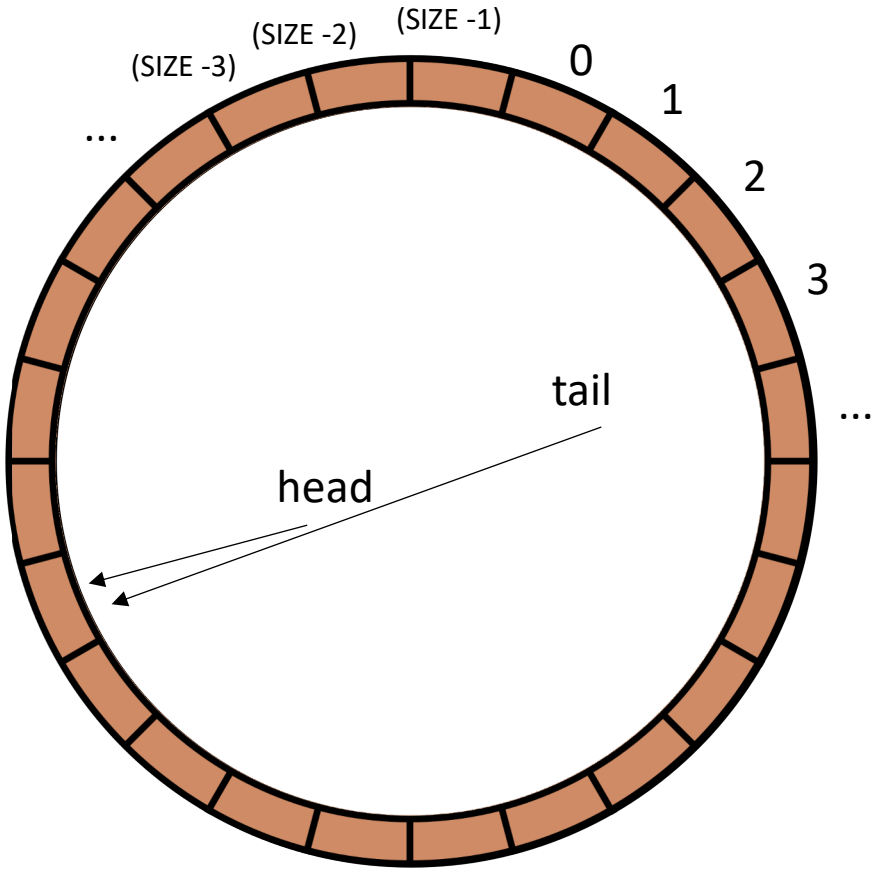
head and tail

Empty queue is when head == tail

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

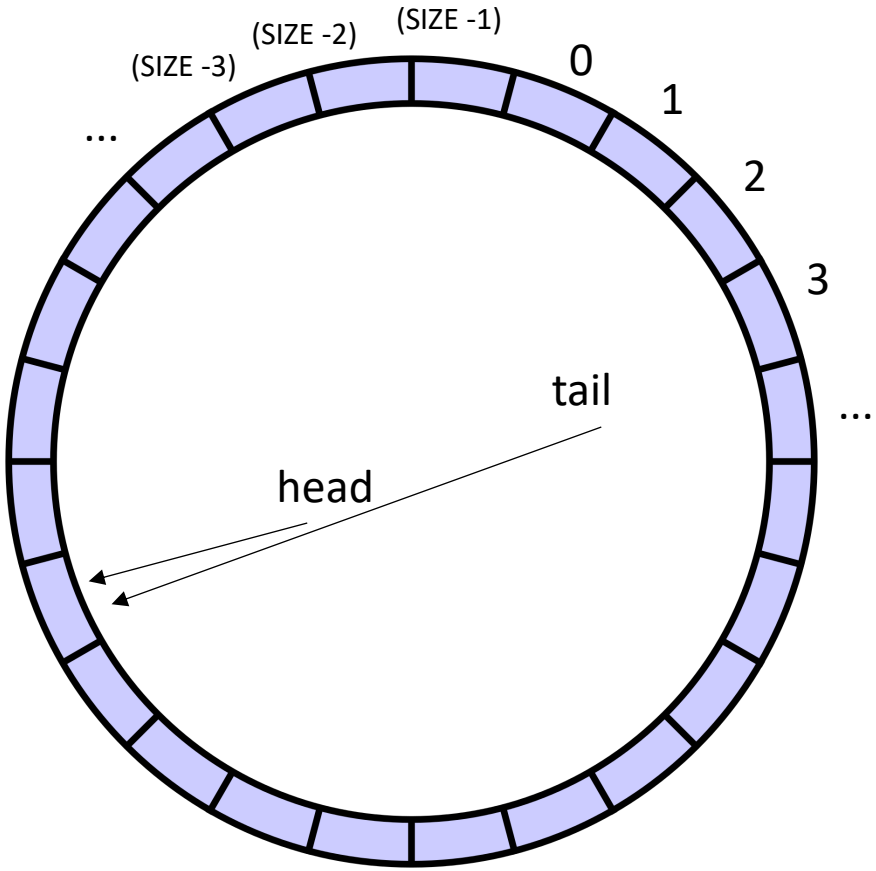
Empty queue is when
head == tail

Full queue is when
head == tail?

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when
head == tail

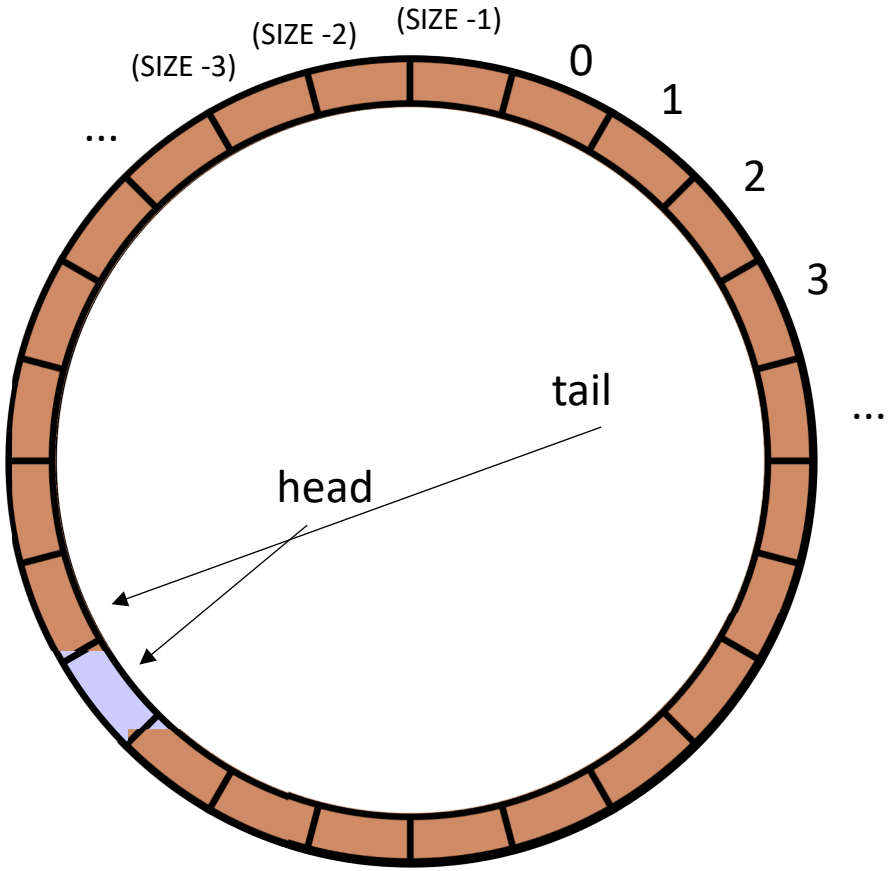
Full queue is when
head == tail?

but then
how to tell
full queue from
empty?

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

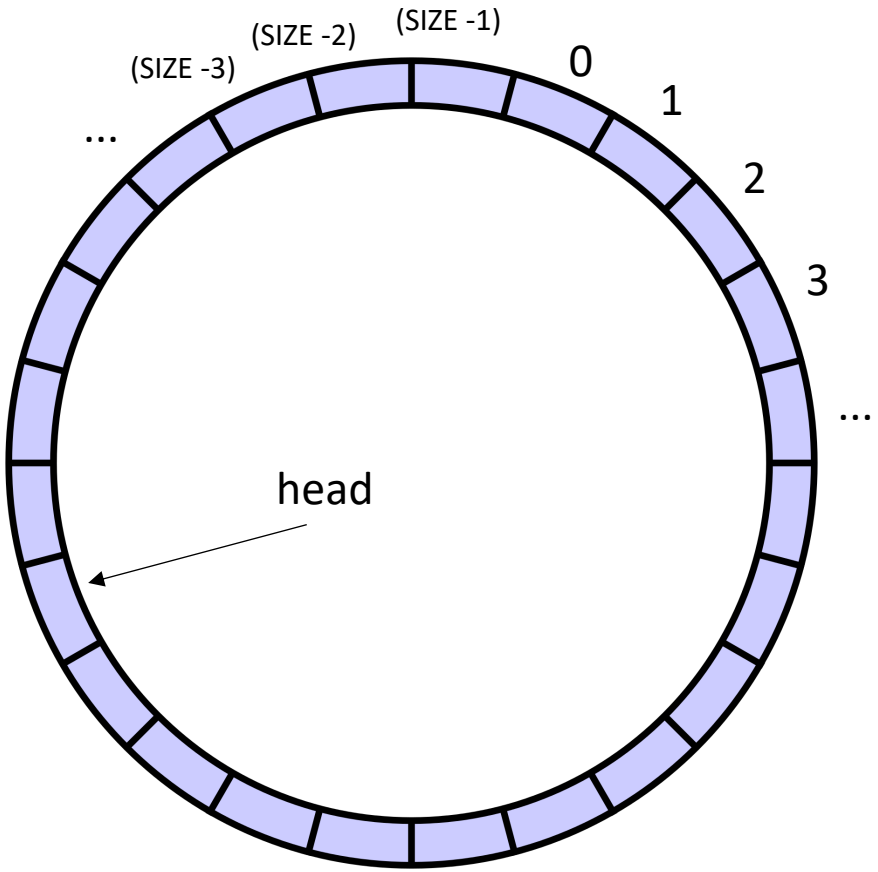
head and tail

Empty queue is when
head == tail

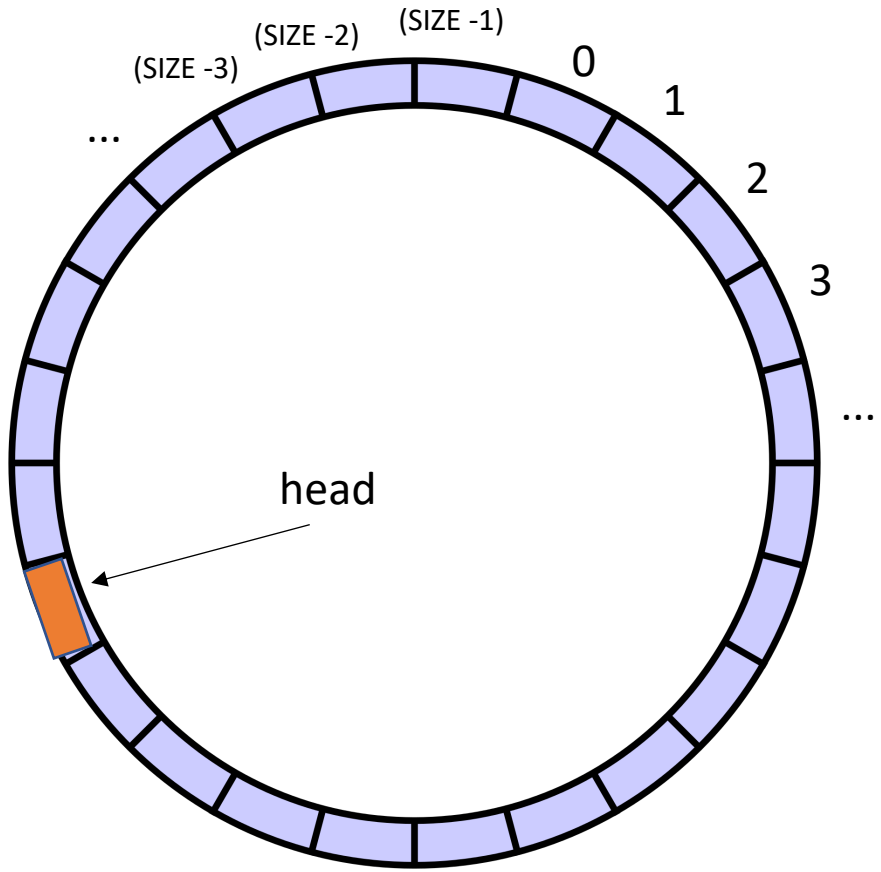
Full queue is when
head + 1 == tail

wasting one location, but its okay...

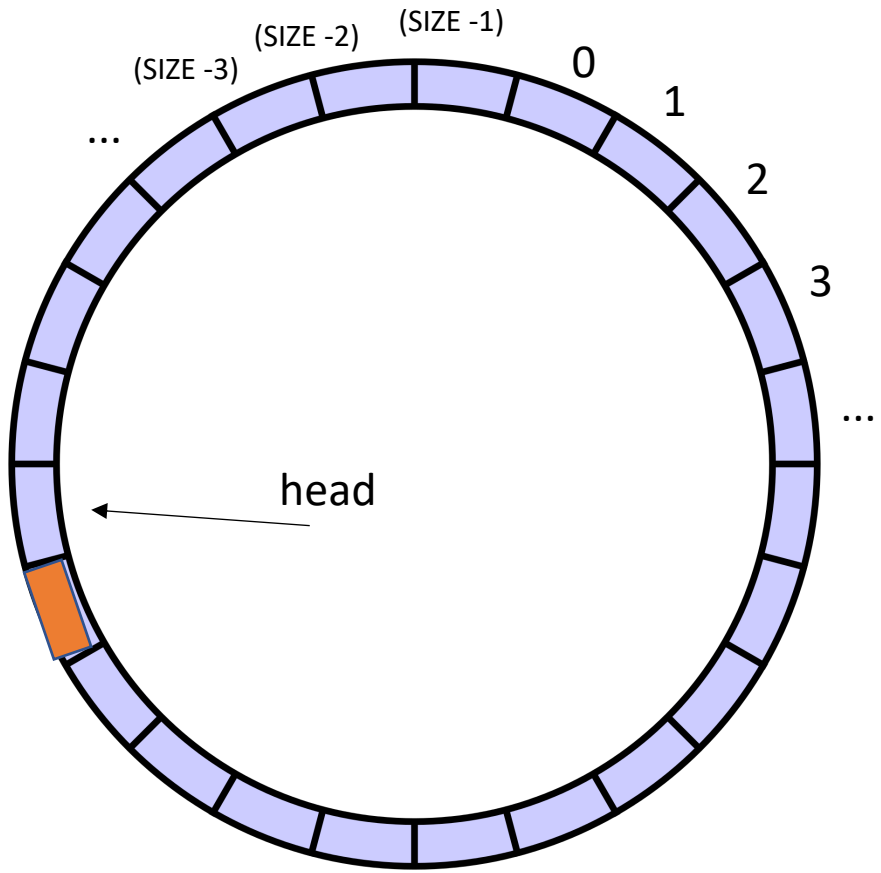
conceptually it is a circle



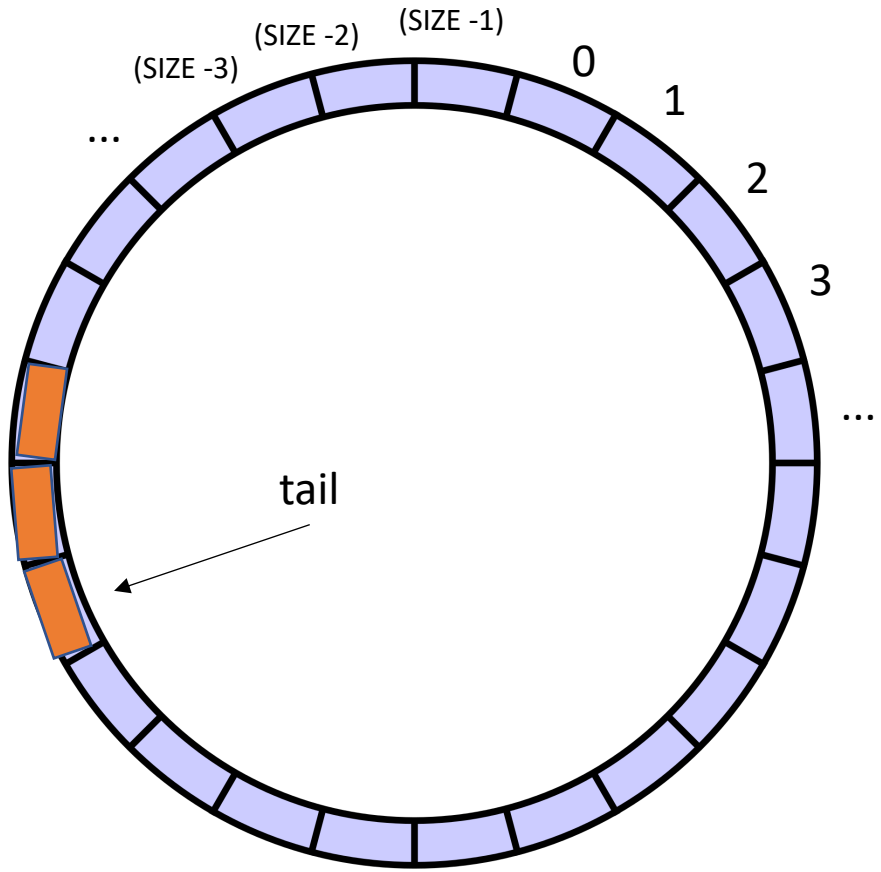

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



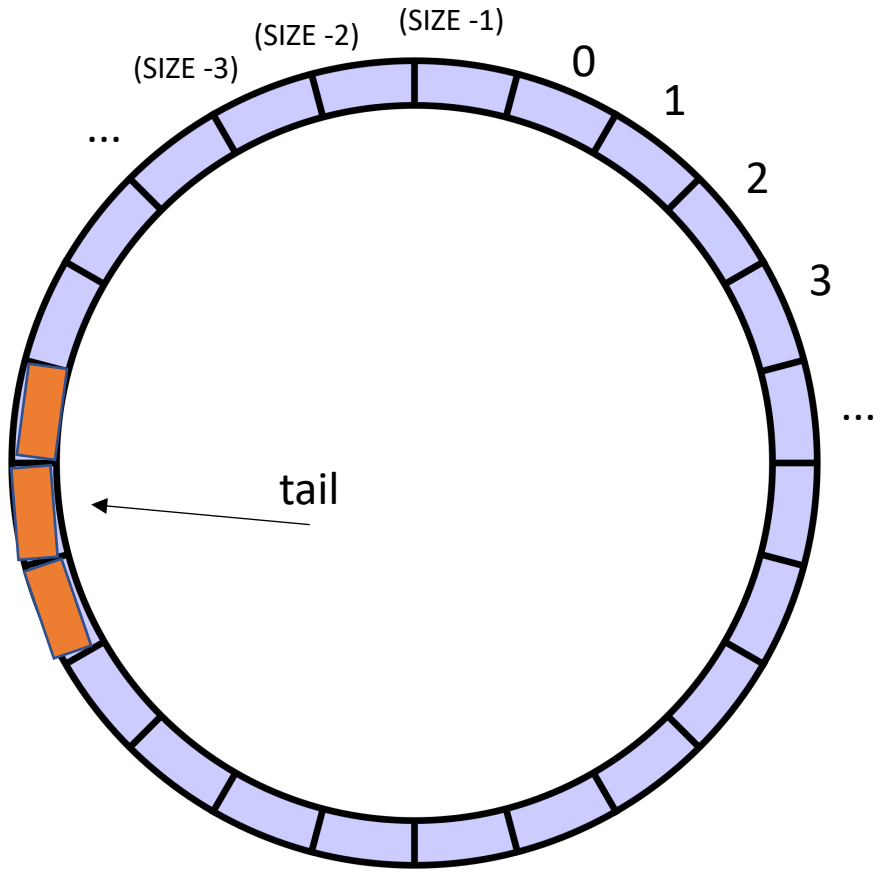

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```

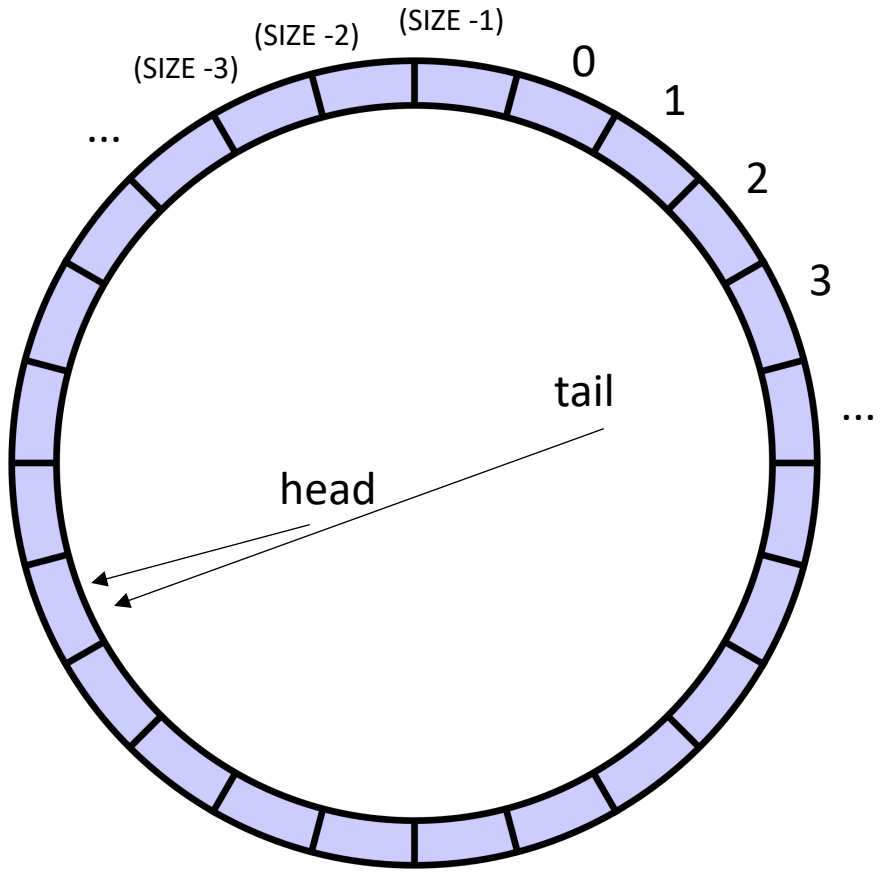



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```



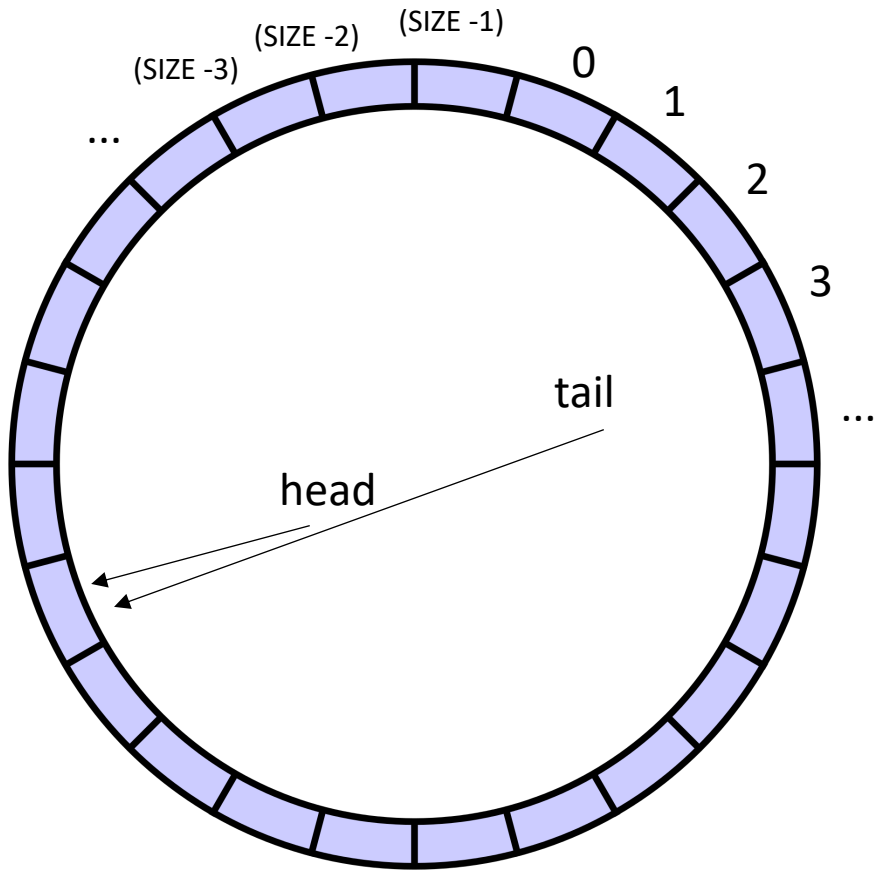

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

This looks like the two threads don't even share head and tail! What is missing?

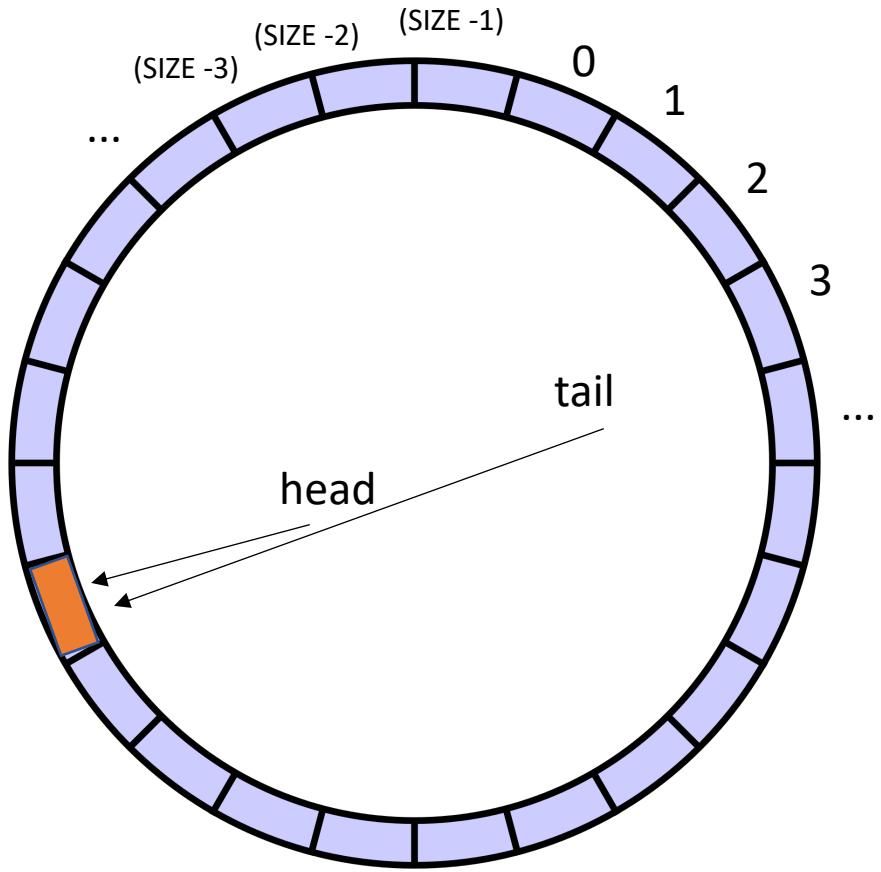



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

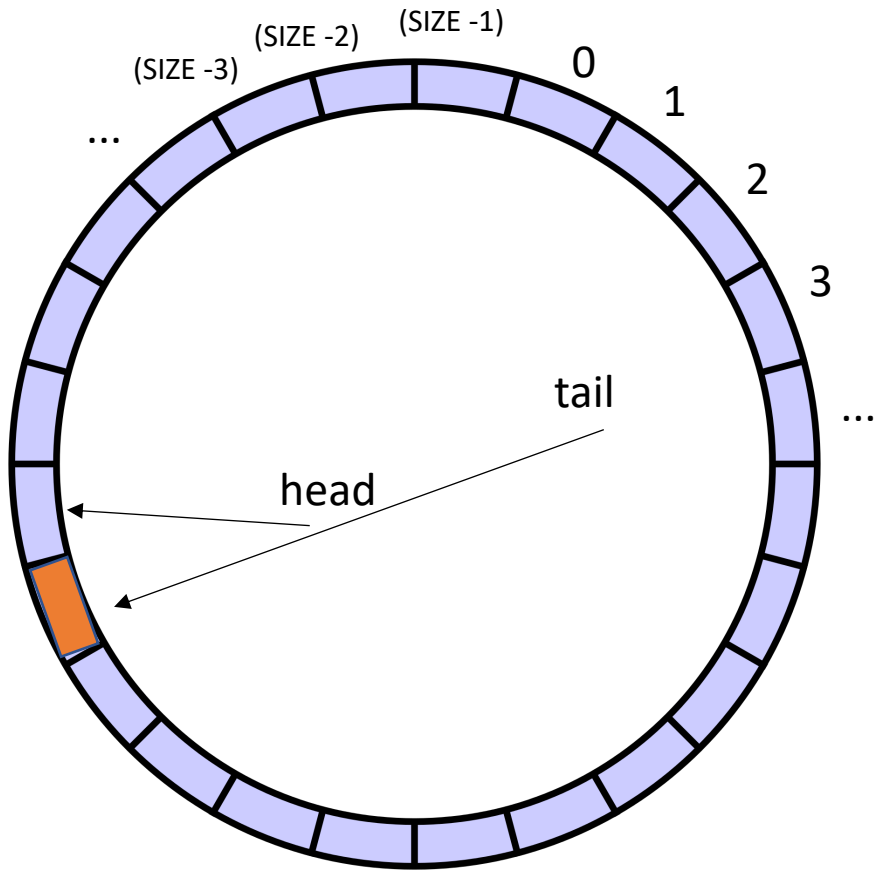
what happens if we try to dequeue here?



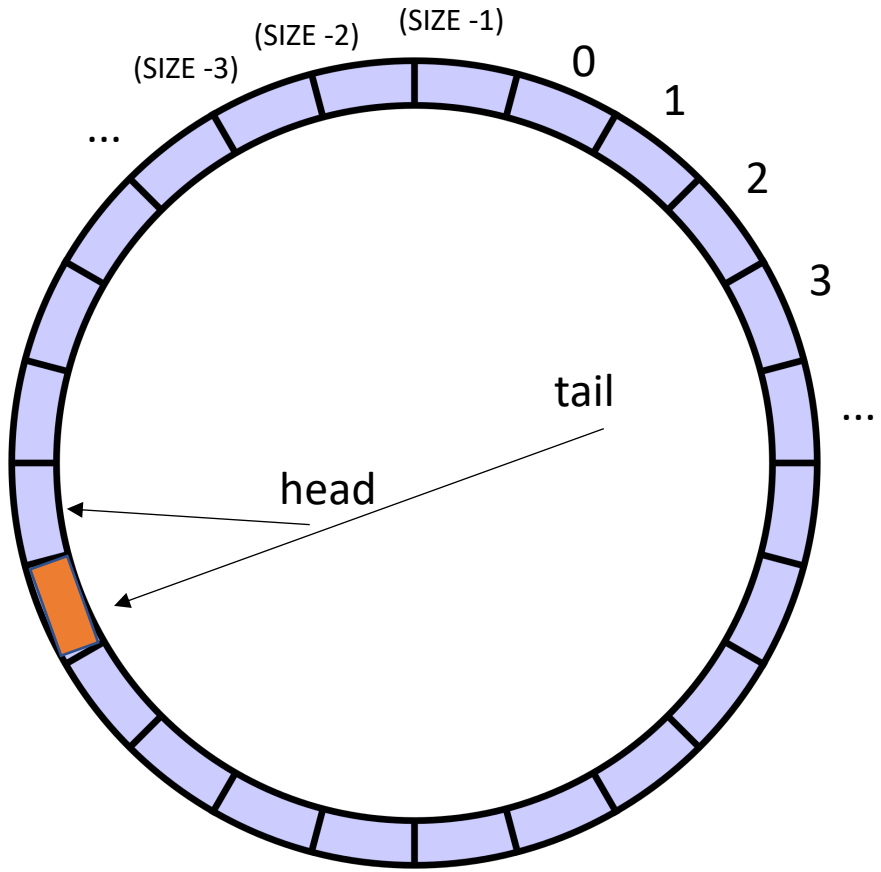

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```



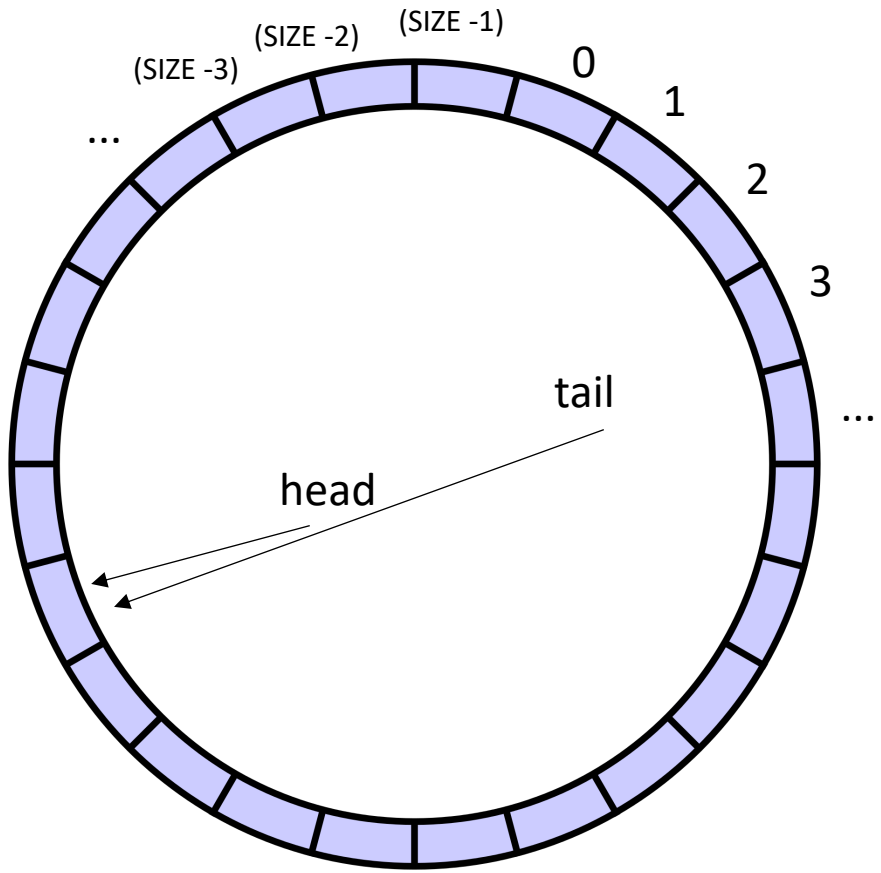

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```



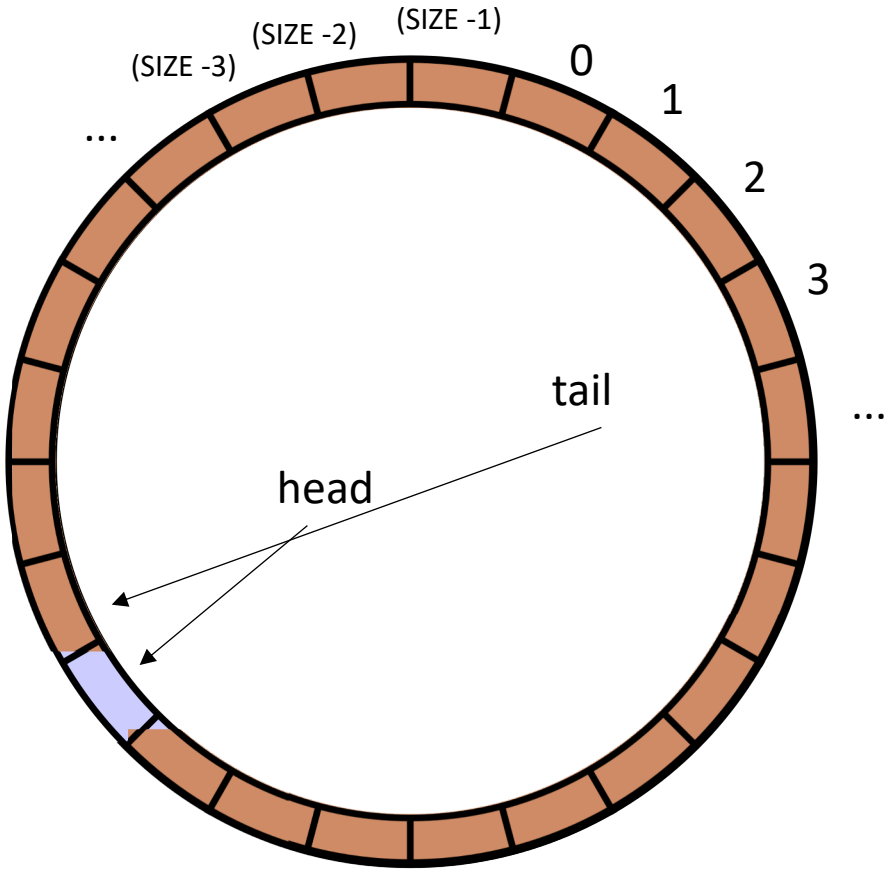

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```



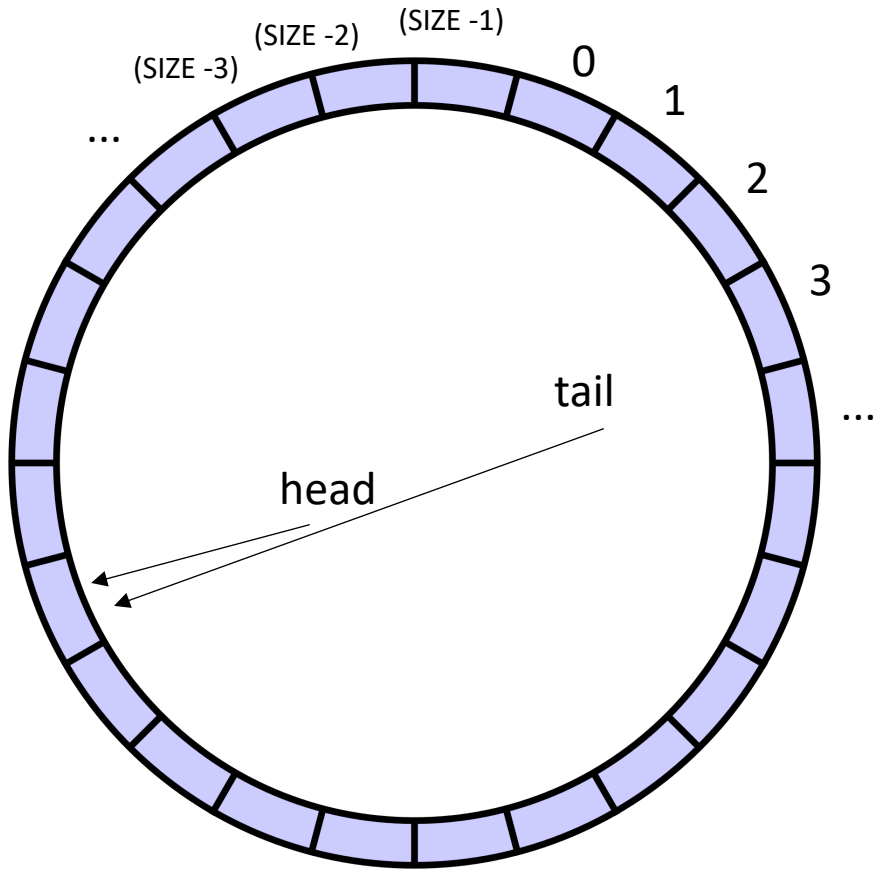

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```

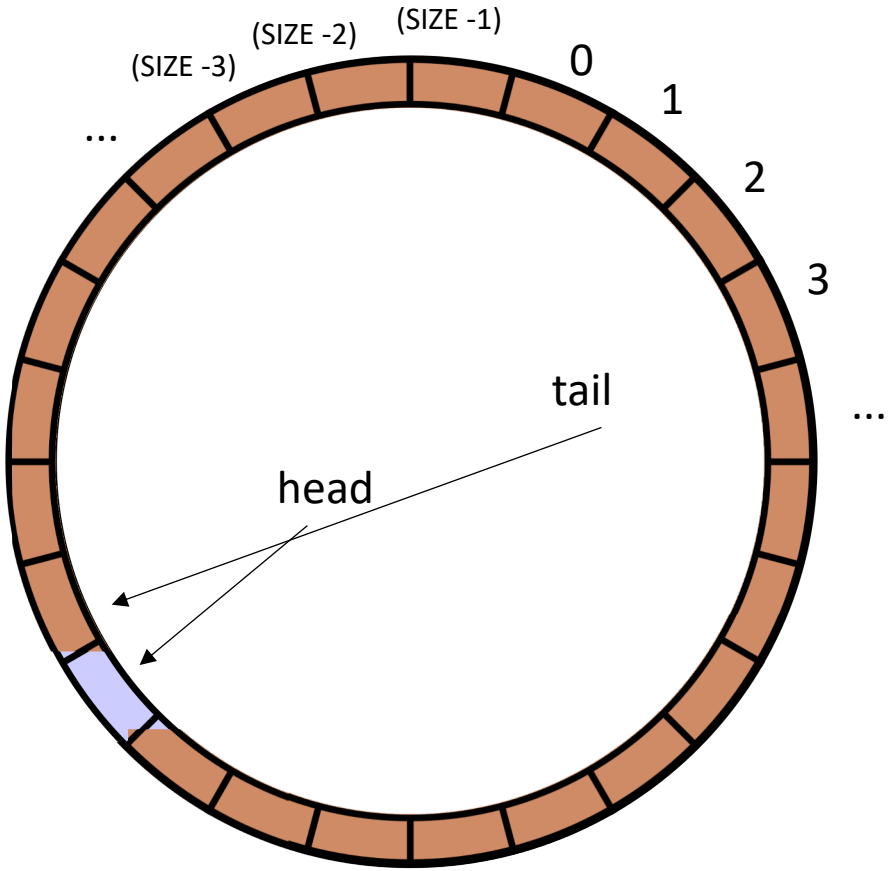
similarly for enqueue

but why can't we enqueue?




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```

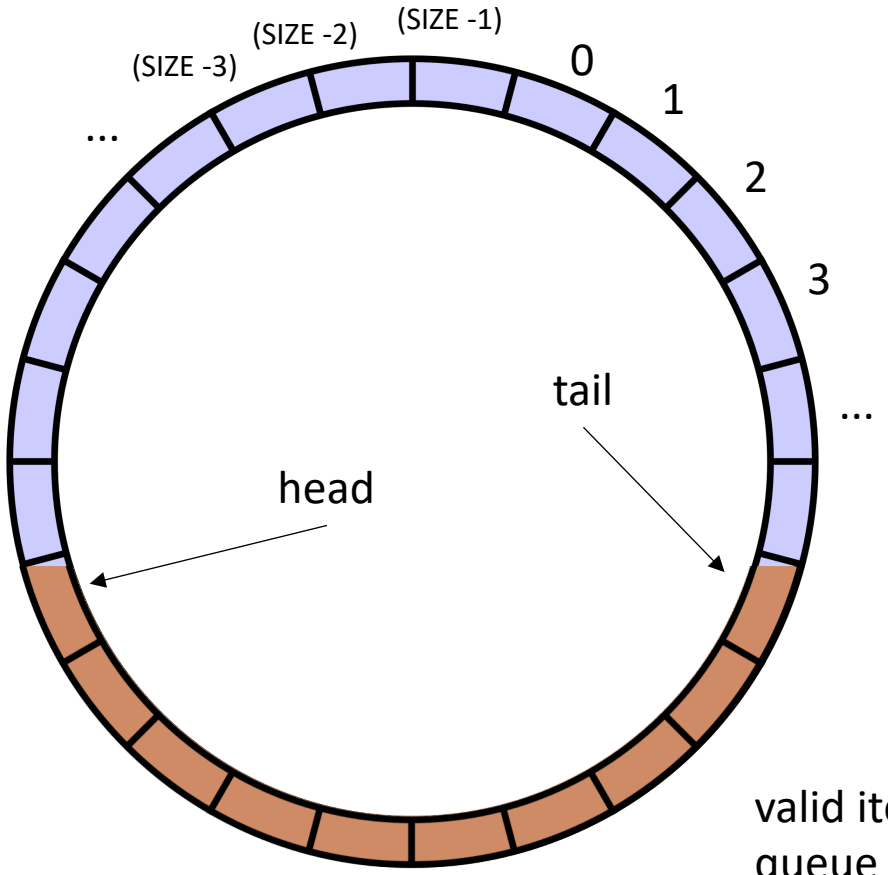
incrementing the head would make it empty!




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```

we need to wait for there to be room

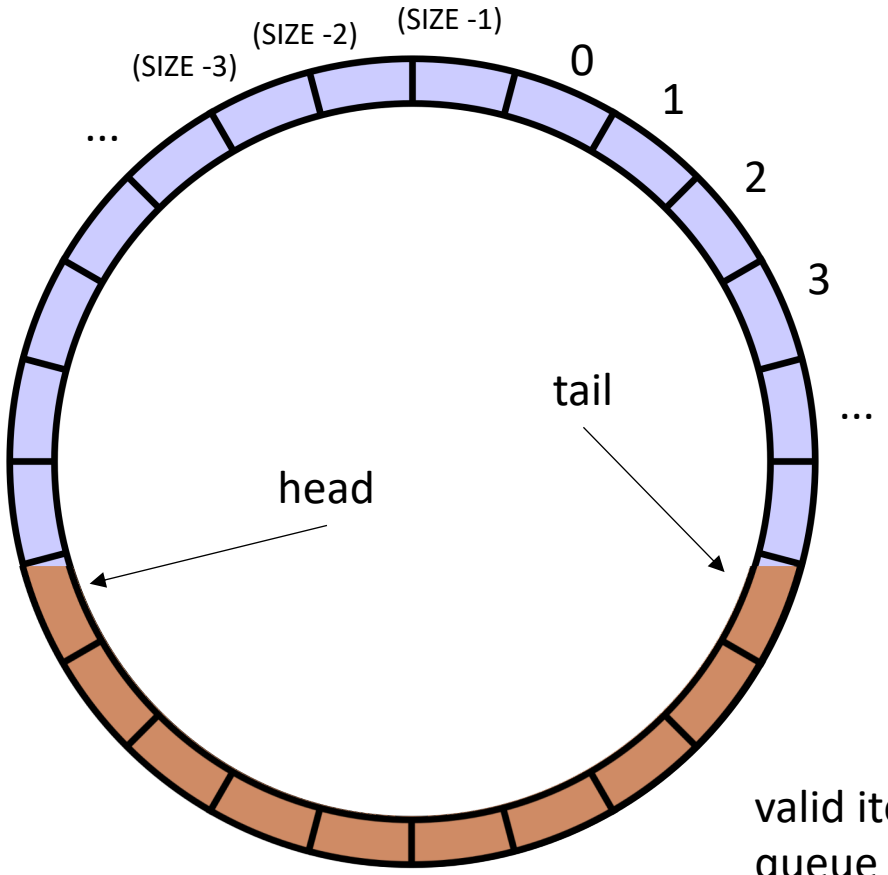
Other questions:



valid items in the queue

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```

Other questions:



valid items in the queue

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is not empty
      // get value at tail
      // increment tail
    }
}
```

Next week

- Workstealing!

- Good luck on the exam and HW!